
Mocklis Documentation

Release 1.0.0

Esbjörn Redmo

Jul 07, 2019

Contents:

1	Introduction	3
1.1	Fake implementations of interfaces	3
1.2	Can be given specific behaviour	5
1.3	Intellisense friendly	5
1.4	Used as dependencies	6
1.5	Verify interactions	7
1.6	Without reflection	8
2	Installation	9
2.1	The quick version	9
2.2	The slightly longer version	10
3	Getting Started	11
3.1	A first mock	11
4	Using Mocklis	17
4.1	Generating Mocklis classes	17
4.2	Adding steps	19
4.3	Work with type parameters	20
4.4	Debugging tests	21
4.5	Refactoring tests	22
4.6	What Mocklis can't do	23
5	Reference	25
5.1	Standard Steps	25
5.2	Verifications	30
5.3	Mocklis Code Generation	31
5.4	Experimental Stuff	38
6	Extending Mocklis	39
6.1	Writing new steps	39
6.2	Writing new verifications	41
6.3	Writing a new logging context	43
7	Frequently Asked Questions	45
7.1	ValueTuple	45
7.2	“Missing” mock	46



Mocklis is a source code generator for .net (currently C#) which creates test doubles from interfaces, along with an api for providing these test doubles with behaviour.

Mocklis is a mocking library for .net (currently C#) that

- creates fake implementations of interfaces
- that can be given specific behaviour
- in an intellisense-friendly way
- to be used as dependencies in components we want to test
- letting us verify that they are correctly interacted with
- without any use of reflection.

Let's go over these points one by one.

1.1 Fake implementations of interfaces

With Mocklis you take an interface that defines a dependency for a component we wish to test, for instance this `IConnection` interface (with some details such as `Message` and `MessageEventArgs` removed for brevity):

```
public interface IConnection
{
    string ConnectionId { get; }
    event EventHandler<MessageEventArgs> Receive;
    Task Send(Message message);
}
```

Then you create an empty class implementing this interface, and decorate it with the `MocklisClass` attribute.

```
[MocklisClass]
public class MockConnection : IConnection
{
}
```

This will of course not compile in its current form. However, the presence of the `MocklisClass` attribute enables a code fix in Visual Studio.

```
namespace MyProject.Tests
{
    [MocklisClass]
    public class MockConnection : IConnection
    {
        // The contents of this class were created by the Mocklis code-gen
        // Any changes you make will be overwritten if the contents are re
    }
}
```

Update Mocklis Class

MocklisAnalyzer Mocklis code can be regenerated

```
{
    // The contents of this class were created by the Mocklis code-gen
    // Any changes you make will be overwritten if the contents are re

    public MockConnection()
    {
        ConnectionId = new PropertyMock<string>(this, "MockConnection"
        Receive = new EventMock<EventHandler<MessageEventArgs>>(this,
        Send = new FuncMethodMock<Message, Task>(this, "MockConnection"
    }

    public PropertyMock<string> ConnectionId { get; }

    string IConnection.ConnectionId => ConnectionId.Value;
}
```

The code fix replaces the contents of the class as follows:

```
[MocklisClass]
public class MockConnection : IConnection
{
    // The contents of this class were created by the Mocklis code-generator.
    // Any changes you make will be overwritten if the contents are re-generated.

    public MockConnection()
    {
        ConnectionId = new PropertyMock<string>(this, "MockConnection", "IConnection",
        "ConnectionId", "ConnectionId", Strictness.Lenient);
        Receive = new EventMock<EventHandler<MessageEventArgs>>(this, "MockConnection",
        "IConnection", "Receive", "Receive", Strictness.Lenient);
        Send = new FuncMethodMock<Message, Task>(this, "MockConnection", "IConnection",
        "Send", "Send", Strictness.Lenient);
    }

    public PropertyMock<string> ConnectionId { get; }

    string IConnection.ConnectionId => ConnectionId.Value;

    public EventMock<EventHandler<MessageEventArgs>> Receive { get; }

    event EventHandler<MessageEventArgs> IConnection.Receive {
        add => Receive.Add(value);
        remove => Receive.Remove(value);
    }

    public FuncMethodMock<Message, Task> Send { get; }

    Task IConnection.Send(Message message) => Send.Call(message);
}
```

You can see that the `IConnection` interface has been explicitly implemented, where the `IConnection.ConnectionId` property gets its value from a *mock property* with the same name. The `IConnection.Receive`

event forwards adds and removes to another *mock property*, and the `IConnection.Send` method forwards calls to yet another *mock property*.

Note that the *mock properties* are generally properties even if the members they support aren't: the `IConnection.Send` method is paired with a *mock property* of type `FuncMethodMock`, the `IConnection.Receive` event is paired with a *mock property* of type `EventMock` and so forth.

The practical upshot is that the `IConnection` interface is now implemented by the class, so that instances of the class can be used in places where the `IConnection` interface is expected.

1.2 Can be given specific behaviour

If we just use the class that was written for us in a real test it might not work as expected. The default behaviour for a newly constructed mock is to do as little as possible and return default values wherever asked, which is probably not what you wanted. The good news is that you can control this on a case by case basis using so-called *steps*.

```
[Fact]
public void ServiceCanCountMessages()
{
    // Arrange
    var mockConnection = new MockConnection();
    mockConnection.ConnectionId.Return("TestConnectionId");
    mockConnection.Receive.Stored(out var registeredEvents);

    var service = new Service(mockConnection);

    // Act
    registeredEvents.Raise(mockConnection, new MessageEventArgs(new Message("Test")));

    // Assert
    Assert.Equal("TestConnectionId", service.ConId);
    Assert.Equal(1, service.ReceiveMessageCount);
}
```












In this example, two steps were used. The `Return` step simply returns a value whenever the mock is used, and the `Stored` step tracks values written to the mock. In this case we also obtained a reference to the store, which tracked attached event handlers letting us raise an event on them for testing purposes. (Exercise for the reader: implement the `Service` class so that the test passes...)

This is of course just an introduction; see the [Reference](#) for a complete list of steps and other constructs used to control how *Mocklis Classes* work.

1.3 Intellisense friendly

Intellisense is a great feature of modern code editors, and Mocklis is written to make the most of it. Your *Mocklis class* exposes *mock properties* for members of implemented interfaces. These *mock properties* have extension methods for all of the different steps that they support, which allows Visual Studio will list the available steps through intellisense.










```
// Arrange
var mockConnection = new MockConnection();
mockConnection.Send.
```

 InstanceRecordBeforeCall	ICanHaveNextMethodStep<TParam,TResult>	
 Join		void
 JoinPoint	ICanHaveNextMethodStep<TParam,TResult>	
 Missing		void
 RecordAfterCall	ICanHaveNextMethodStep<TParam,TResult>	
 RecordBeforeCall	ICanHaveNextMethodStep<TParam,TResult>	
 Return		void
 ReturnEach	ICanHaveNextMethodStep<TParam,TResult>	
 ReturnOnce	ICanHaveNextMethodStep<TParam,TResult>	
 Throw		void
 Times	ICanHaveNextMethodStep<TParam,TResult>	

Thanks to the extension method approach this list would also include any bespoke steps that have been added, whether defined in your own solution or in third party packages.

When mocking out method calls, all arguments are combined into a named value tuple (unless there's exactly one in which case that one is used), which means that we get intellisense for using those parameters as well.

```
var mock = new MockSample();
mock.MonthlyPayment.Func(a => a.
```

 interestRate	double
 loanSize	double
 numberOfMonths	int
 CompareTo	int
 GetHashCode	int
 Equals	bool
 ToString	string
 GetType	Type
 ToTuple (using System)	Tuple<T1,T2,T3>

1.4 Used as dependencies

Since *Mocklis classes* implement interfaces explicitly, we don't risk a name clash with the *mock properties* (and indeed if possible, the *mock properties* will be given the same name as the interface member it's paired with), and we can use the *Mocklis class* instance directly wherever the interface is expected.

Mocklis classes can also implement more than one interface in cases where the component it acts as a stand-in for would implement more than one interface. Common cases include where a class would implement a service interface and `IDisposable`, or an interface with property accessors and `INotifyPropertyChanged`. If you need to mock out an enumerable, your *Mocklis class* can mock both `IEnumerable<T>` and `IEnumerator<T>` at the same time.

However, this also means that *Mocklis classes* can not create mocks for virtual members of an (abstract) base class, as these can not be explicitly implemented.

1.5 Verify interactions

There are a number of ways in which you can verify that the ‘component under test’ makes the right calls to your mocked dependency. There are a number of ways to do this using steps: simple cases:

- If you have a method you don’t expect to be called, you can use a `Throw` step to throw an exception which will hopefully bubble up through your code and fail the test.
- If you have a property, event or indexer you can use a `Stored` step and manually check that the right value was stored.
- If you have a method then you can use a `Func` or `Action` step and let that set a flag which you can later manually assert.
- You can use a `Record` step to record all interactions and check that the right interactions happened.

Mocklis also has a set of verification classes and interfaces that can be used to add checks to your *mock properties* and to verify the contents of `Stored` steps in a declarative way. You create a `VerificationGroup`, pass it to checks and verification steps, and assert everything in one go.

Take for instance this, somewhat contrived, test:

```
[Fact]
public void TestIndex()
{
    // Arrange
    var vg = new VerificationGroup("Checks for indexer");
    var mockIndex = new MockIndex();
    mockIndex.Item
        .ExpectedUsage(vg, null, 1, 3)
        .StoredAsDictionary()
        .CurrentValuesCheck(vg, null, new[]
        {
            new KeyValuePair<int, string>(1, "one"),
            new KeyValuePair<int, string>(2, "two"),
            new KeyValuePair<int, string>(3, "three")
        });

    var index = (IIndex) mockIndex;

    // Act
    index[1] = "one";
    index[2] = "two";
    index[3] = "three";

    // Assert
    vg.Assert(includeSuccessfulVerifications: true);
}
```

This test will fail with the following output:

```
Mocklis.Verification.VerificationFailedException : Verification Failed.

FAILED: Verification Group 'Checks for indexer':
FAILED: Usage Count: Expected 1 get(s); received 0 get(s).
Passed: Usage Count: Expected 3 set(s); received 3 set(s).
FAILED: Values check:
Passed: Key '1'; Expected 'one'; Current Value is 'one'
```

(continues on next page)

(continued from previous page)

```
Passed:      Key '2'; Expected 'two'; Current Value is 'two'  
FAILED:      Key '3'; Expected 'thre'; Current Value is 'three'
```

Note that all verifications are checked - it will not stop at the first failure. By default the assertion will not show the Passed verifications (although the exception itself has a `VerificationResult` property, so you can always get to it). If you want to include all verifications in the exception message you need to pass true for the `includeSuccessfulVerifications` parameter, as was done in the sample above. Without it you would only see the lines that failed.

The *null* values in the example are placeholders for strings that would be added to the relevant lines in the result to help finding the culprit if the assertion failed.

1.6 Without reflection

Maybe this point should have gone in first. Mocklis does not use reflection to find out information about mocked interfaces, and it does not use emit or dynamic proxies to add implementations on the fly. Furthermore the mock instance and the object used to ‘program’ the mock are the same thing. There are pros and cons with this approach:

1.6.1 Pros

- What you see is what you get. No code is hidden from view, and you can freely set break points and inspect variables as you’re debugging your tests.
- You can easily extend Mocklis with your own steps, with whatever bespoke behaviour you might need.
- Running your tests is significantly faster than it would have been with on-the-fly generated dynamic proxies.

1.6.2 Cons

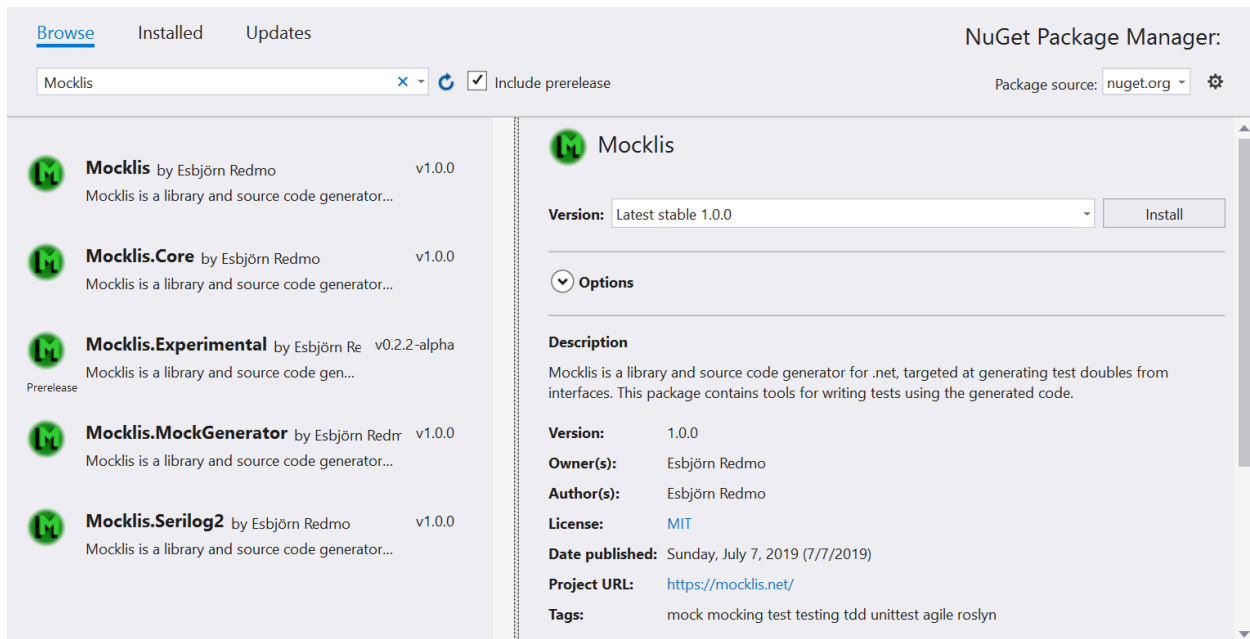
- Your project will include code for mocked interfaces, although that code can be reused by all tests using the interface.
- The code in question has to be written, although the code generator bundled with Mocklis does this for you.
- The design only really works for interfaces and not for mocking members of virtual base classes.

2.1 The quick version

Add the `Mocklis` nuget package to your test project.

This can be done with the NuGet browser in Visual Studio: Search for ‘Mocklis’ while on the Browse tab, and you should see the Mocklis packages and be able to add `Mocklis` to your project.

You’ll need to tick the checkbox that says ‘Include prerelease’ to see the `Mocklis.Experimental` package.



The screenshot shows the NuGet Package Manager interface. At the top, there are tabs for 'Browse', 'Installed', and 'Updates'. The search bar contains 'Mocklis'. Below the search bar, there is a checkbox for 'Include prerelease' which is checked. The package source is set to 'nuget.org'. The search results list five packages:

- Mocklis** by Esbjörn Redmo v1.0.0
- Mocklis.Core** by Esbjörn Redmo v1.0.0
- Mocklis.Experimental** by Esbjörn Re v0.2.2-alpha (Prerelease)
- Mocklis.MockGenerator** by Esbjörn Redrr v1.0.0
- Mocklis.Serilog2** by Esbjörn Redmo v1.0.0

The details for the **Mocklis** package are shown on the right:

- Version:** Latest stable 1.0.0
- Options:** (expanded)
- Description:** Mocklis is a library and source code generator for .net, targeted at generating test doubles from interfaces. This package contains tools for writing tests using the generated code.
- Version:** 1.0.0
- Owner(s):** Esbjörn Redmo
- Author(s):** Esbjörn Redmo
- License:** MIT
- Date published:** Sunday, July 7, 2019 (7/7/2019)
- Project URL:** <https://mocklis.net/>
- Tags:** mock mocking test testing tdd unittest agile roslyn

2.2 The slightly longer version

There are two things that you'll need to get hold of to run Mocklis.

Firstly there is a code generator that builds test double classes from interface definitions. The recommended way is to use a Roslyn Analyzer + Code Fix supplied in the form of the NuGet package `Mocklis.MockGenerator`. (If you're building Mocklis from sources - kudos if you do - there is an embryo to a command-line version. This is just the absolute bare minimum needed to load a solution and update all `MocklisClasses` within.)

Then there is a library of pluggable 'steps' which provide bite-sized behaviours to the test doubles, along with some supporting code. This library is spread over a number of assemblies, most notably `Mocklis.Core` which contains the minimum amount of code required to build the test doubles, and `Mocklis` which you use in your tests to add behaviour. There's also `Mocklis.Experimental` for steps whose design is still under development (read: steps that simply haven't been axed yet...) and `Mocklis.Serilog2` which contains a logging provider for Serilog 2.x.

Note that `Mocklis.Experimental` is going to stay a perpetual 0.x pre-release, so that it can have breaking changes without violating the semantic versioning rules.

3.1 A first mock

Mocking is the art of creating fake but controllable replacements for dependencies to code that we wish to test. However, for a simple walk-through of the functionality we'll write a normal console application instead hoping that not too much is lost in the transition.

Let's say we're writing a component that reads numbers from standard input, sends them off to a web service for some calculation and writes the result to standard output.

The first step is to create two interfaces. One for interacting with the console, and one for the web service:

```
public interface IConsole
{
    string ReadLine();
    void WriteLine(string s);
}

public interface IService
{
    int Calculate(params int[] values);
}
```

And then we have our code that uses these two interfaces. Let's add a constructor to our `Program` class, along with fields for the dependencies we'll use.

```
public class Program
{
    public static void Main()
    {
    }

    private readonly IConsole _console;
```

(continues on next page)

(continued from previous page)

```
private readonly IService _service;

public Program(IConsole console, IService service)
{
    _console = console;
    _service = service;
}
}
```

We will at some point write proper implementations of these interfaces, but for now we want to just mock them out.

Add two new classes, `MockConsole` and `MockService`. Let them implement their corresponding interface but otherwise remain empty. Reference the `Mocklis` NuGet package from your project and add the `MocklisClass` attribute to both classes.

For this walkthrough we need to add the ‘Strict’ parameter to the attributes - this is purely because we want to get exceptions for missing configurations to guide us to write more mocks. In your real life cases you may wish to have all mocks return default values instead of throwing exceptions in which case leave out the *Strict* parameter.

```
[MocklisClass(Strict = true)]
public class MockConsole : IConsole
{
}

[MocklisClass(Strict = true)]
public class MockService : IService
{
}
```

Now you can use the *Update Mocklis Class* code fix to create implementations for these interfaces. If you look at the `MocklisClass` attribute you’ll see that the first characters have an underline. This is Visual Studio hinting that there is a code fix available. Move your mouse over that area, and Visual Studio will provide you with a light-bulb. Click on the dropdown arrow and you’ll see a list of suggestions. Pick ‘Update Mocklis Class’ from the list, which will create an implementation of the class. Do this for both classes.

Instantiate these mocks in the static `Main`. Pass the instances to the constructor, create a non-static `Run` method and call it once the program instance has been created:

```
public static void Main()
{
    var mockConsole = new MockConsole();
    var mockService = new MockService();

    var program = new Program(mockConsole, mockService);
    program.Run();
}

public void Run()
{
}
```

Note that you didn’t have to cast `mockConsole` to `IConsole`, or `mockService` to `IService`. As long as the parameters accepting the mocked instances are of an implemented interface type, C# will perform an implicit cast.

Now we want to have a play with the interfaces. Let’s say we read numbers off standard input until we get an empty string, pass them all to the service, and then write the return value back to the console.


```

public void Run()
{
    var values = new List<int>();
    for (;;)
    {
        string s = _console.ReadLine();
        if (string.IsNullOrEmpty(s))
        {
            break;
        }
        values.Add(int.Parse(s));
    }

    var result = _service.Calculate(values.ToArray());
    _console.WriteLine(result.ToString());
}

```

If we try to run this we'll fall over with a `MockMissingException`:

```

Mocklis.Core.MockMissingException: No mock implementation found for Method 'IConsole.
↪ReadLine'. Add one using 'ReadLine' on your 'MockConsole' instance.

```

Let's fix this with some mocking. First we want to return some strings from the mocked console. Let's say the strings "8", "13", "21", and an empty string. We should also add logging so we can follow what's going on. Update `Main` as follows:

```

public static void Main()
{
    var mockConsole = new MockConsole();
    var mockService = new MockService();

    mockConsole.ReadLine.Log().ReturnEach("8", "13", "21", string.Empty);

    var program = new Program(mockConsole, mockService);
    program.Run();
}

```

Running the program now should give us the following output, most of it coming from the `Log` step.

```

Calling '[MockConsole] IConsole.ReadLine'
Returned from '[MockConsole] IConsole.ReadLine' with result: '8'
Calling '[MockConsole] IConsole.ReadLine'
Returned from '[MockConsole] IConsole.ReadLine' with result: '13'
Calling '[MockConsole] IConsole.ReadLine'
Returned from '[MockConsole] IConsole.ReadLine' with result: '21'
Calling '[MockConsole] IConsole.ReadLine'
Returned from '[MockConsole] IConsole.ReadLine' with result: ''
Mocklis.Core.MockMissingException: No mock implementation found for Method 'IService.
↪Calculate'. Add one using 'Calculate' on your 'MockService' instance.

```

Apparently we're missing a mock for the `IService.Calculate` interface member. Let's add that. In fact, let's just pretend that the service adds up anything that is sent to it.

```

public static void Main()
{
    var mockConsole = new MockConsole();
    var mockService = new MockService();

```

(continues on next page)

(continued from previous page)

```

mockConsole.ReadLine.Log().ReturnEach("8", "13", "21", string.Empty);
mockService.Calculate.Log().Func(m => m.Sum());

var program = new Program(mockConsole, mockService);
program.Run();
}

```

Which should now give us the following when we run the program:

```

Calling '[MockConsole] IConsole.ReadLine'
Returned from '[MockConsole] IConsole.ReadLine' with result: '8'
Calling '[MockConsole] IConsole.ReadLine'
Returned from '[MockConsole] IConsole.ReadLine' with result: '13'
Calling '[MockConsole] IConsole.ReadLine'
Returned from '[MockConsole] IConsole.ReadLine' with result: '21'
Calling '[MockConsole] IConsole.ReadLine'
Returned from '[MockConsole] IConsole.ReadLine' with result: ''
Calling '[MockService] IService.Calculate' with parameter: 'System.Int32[]'
Returned from '[MockService] IService.Calculate' with result: '42'
Mocklis.Core.MockMissingException: No mock implementation found for Method 'IConsole.
->WriteLine'. Add one using 'WriteLine' on your 'MockConsole' instance.

```

Ok - so we're still missing mocking out the `WriteLine` method. Let's do so, add logging (as for the other ones) and also recording. A `Record` step will remember everything that was passed through it, and make it available using its out parameter as an `IReadOnlyList`.

Record steps, like `Log` steps, kind of expect you to chain further steps. They don't make any decisions on their own. In this case we didn't provide a further step so a default behaviour kicks in which is to accept any input and provide default values for any output required. This is normally the default behaviour as well for mocks that don't have any steps defined at all, but we overrode that behaviour with the `Strict = true` switch in the `MocklisClass` attribute. If we want to throw for an incomplete mock as well (such as only providing a `Log` or `Record` without a subsequent step) you can set the `VeryStrict = true` attribute switch. If you'd done so, you would have needed to add a `Dummy` step after the `RecordBeforeCall` step.

Let's also write out the first recorded value (in fact the only recorded value) to the real console so we can see the full thing end-to-end.

```

public static void Main()
{
    var mockConsole = new MockConsole();
    var mockService = new MockService();

    mockConsole.ReadLine.Log().ReturnEach("8", "13", "21", string.Empty);
    mockConsole.WriteLine.Log().RecordBeforeCall(out var consoleOut);
    mockService.Calculate.Log().Func(m => m.Sum());

    var program = new Program(mockConsole, mockService);
    program.Run();

    Console.WriteLine("The value 'written' to console was " + consoleOut[0]);
}

```

The parameter to `RecordBeforeCall` returns a list with the recorded values, which by default is just a list of the values passed to the method. You may want to store a subset of these or do some calculation on some values (or if they are mutable, get the current values before they're changed) in which case you can add a selector func as a second parameter.

The program now completes without any exceptions, with the following output:

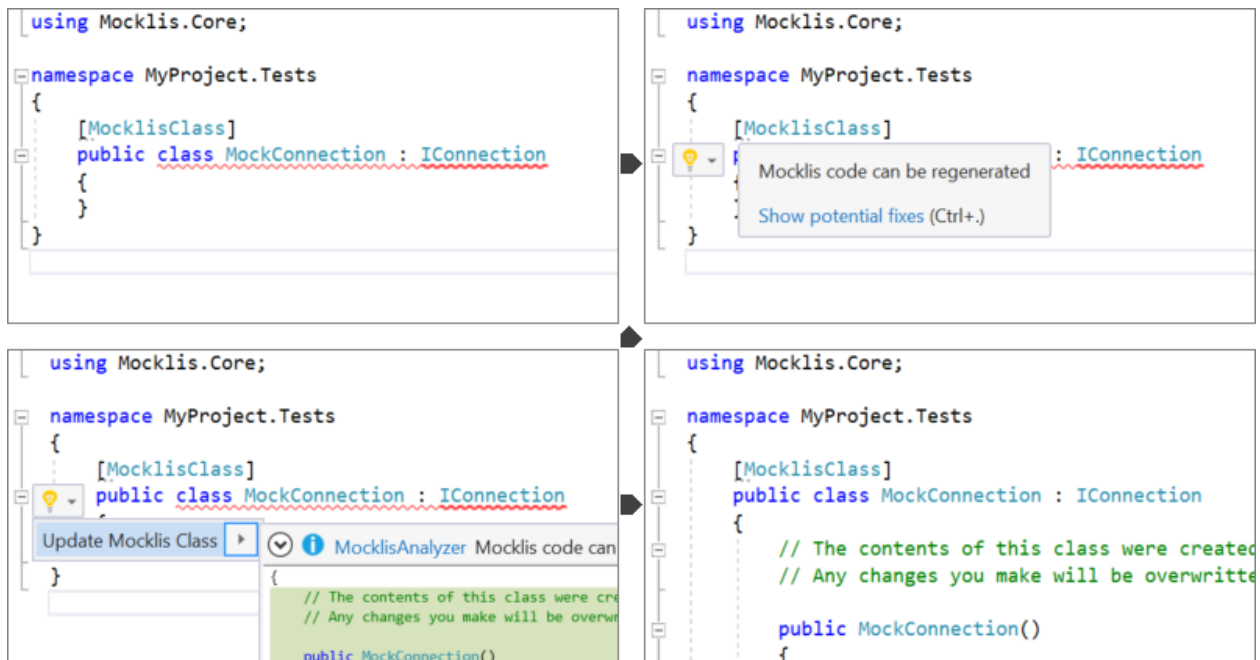
```
Calling '[MockConsole] IConsole.ReadLine'  
Returned from '[MockConsole] IConsole.ReadLine' with result: '8'  
Calling '[MockConsole] IConsole.ReadLine'  
Returned from '[MockConsole] IConsole.ReadLine' with result: '13'  
Calling '[MockConsole] IConsole.ReadLine'  
Returned from '[MockConsole] IConsole.ReadLine' with result: '21'  
Calling '[MockConsole] IConsole.ReadLine'  
Returned from '[MockConsole] IConsole.ReadLine' with result: ''  
Calling '[MockService] IService.Calculate' with parameter: 'System.Int32[]'  
Returned from '[MockService] IService.Calculate' with result: '42'  
Calling '[MockConsole] IConsole.WriteLine' with parameter: '42'  
Returned from '[MockConsole] IConsole.WriteLine'  
The value 'written' to console was 42
```

And with that we have written our first program with mocked interfaces using Mocklis. Of course normally we don't work with mocking outside of unit tests, so this was for illustration only. But it should have given you some idea of what you can use Mocklis for.

4.1 Generating Mocklis classes

The Mocklis code generator takes a class which implements one or more interfaces and replaces the contents of that class with valid implementations of the interface members along with hooks through which you can control the behaviour of those members on a case by case basis. It does this by means of an analyzer/code-fix pair. Any class that is decorated with the `MocklisClass` attribute will make the code-fix available.

The code fix appears in Visual Studio as an underline of the first couple of characters in the attribute. Hover over these with the mouse, click on the arrow next to the light-bulb that appears and select 'Update Mocklis Class'.



If you find it a bit tricky to find the right spot to hover over, you can change the 'Severity' of the rule in your project

as follows: In the solution explorer, expand References, Analyzers and Mocklis.MockGenerator. Right-click on the MocklisAnalyzer, and choose 'Set Rule Severity'. Choose 'Warning' and your entire MocklisClass will be underlined with a green squiggly line and you can update the class from anywhere.

The code generator will make use of any using statements it can find to shorten names of types. It will not add any of its own, so if you generate a class and find that you end up with a lot of fully qualified types you can just add the relevant using statements, and then re-generate the class.

The details of the code generation can be found in the [Reference](#) section.

4.1.1 Attribute parameters

The generated code can be customised to a degree using properties on the `MocklisClass` attribute. The first two we'll look at deal with how a mock should act when it's got an incomplete or missing configuration, and the other two deal with generating code for methods returning values by reference.

One caveat: you can only set these properties to *true* or *false*. Anything else, like expressions, references to constants etc. will be interpreted as the default value for the parameter. Mocklis reads the parameters from code that has been parsed and analyzed, but it's not running code. While it may be theoretically possible to do the required evaluations from the semantic model this has not been attempted.

Strictness parameters

Normally when you call a member on a mock that doesn't have any steps assigned to it, it will do the least amount of work possible and return the least it can to the caller. In other words do nothing and return *default* where needed.

This may or may not be what you want. Another approach is that you should assign steps for all members that you expect to be used; if any of your unconfigured members are being called that's a sign that your code is behaving in ways that you didn't expect. To get mocks that throw an exception whenever this happens you add a `Strict=true` parameter to the `MocklisClass` attribute.

However we can be more draconian than that. Some steps will forward to further steps. If you want to throw not only on a completely unconfigured step, but also when a step is missing a step to forward to, you can add the `VeryStrict=true` parameter to the `MocklisClass` attribute.

The exception thrown is a `MockMissingException`, and you can ask to have it thrown regardless of strictness level by adding a `Missing` step. Likewise if you want to mimic the lenient behaviour, you can add a `Dummy` step.

For an example where the `Strict=true` parameter is used, see the [Getting Started](#) section.

Return by reference

C#7 supports returning values by reference. However the `ref` keyword is not part of the returned type, and we cannot therefore just introduce a `Mocklis mock` where the return type is `ref int` rather than `int`. So we can cheat by pretending that the call is not by reference, and then we wrap it in an object at the last minute to fulfil the contract specified with the `ref` keyword in the interface.

Now, there are (at least) two reasons for returning values by reference. One is that we wish to modify the original value through the reference, the other is to improve performance where the returned type is larger than the size of a pointer. In the latter case we normally mark them as `ref readonly` to signal that you cannot make modifications through the reference. At least since C# 7.2 when this feature was introduced.

In the former case we usually need the reference to point to something known - so that the update has an effect that is visible to other code. In the second it doesn't matter if the reference is to a newly created wrapper object that's only there to fulfil a contract. Therefore Mocklis' default behaviour differs between `ref` and `ref readonly`; for `ref readonly` we cheerfully wrap and for `ref` we use a *virtual method* fallback: Instead of normal mocks, we

define virtual methods for all accessors of the mocked member and instead of adding steps you need to subclass the `MocklisClass` class and add bespoke implementations there. The base implementation for these methods is to always throw a `MockMissingException` regardless of strictness level. This is for technical reasons, the fallback is used in other cases where it turns out to be remarkably tricky to produce code that does nothing, returns default values and at the same time compiles...

But anyway.

You may want to use the fallback for `ref readonly` returns, and you might want to use the object wrapper solution for `ref` returns. The `MockReturnsByRefReadOnly` parameter defaults to true (meaning to use *mock properties* and the wrapper solution), but you can set it to false if you like, which means to use virtual methods to control the behaviour of `ref readonly` returns. Likewise the `MockReturnsByRef` parameter defaults to false (meaning to use the fallback), but you can set it to true if you like, changing the code created to deal with `ref` returns to use mocks and the wrapper solution.

In the current version the parameters will affect all members in the class and there is no means by which we can chose what members to use what strategy for. There are no plans to change this but if you really need it please raise an issue on GitHub.

4.2 Adding steps

If you just need an object that implements an interface to pass to a constructor or method then you don't need to add any steps at all - just an instance will do.

If the instance is used, but you don't really care about what it does or returns you will get away with not doing any configuration, as long as you've created a lenient mocklis class, which is after all the default.

In other cases you'll need to add configuration via steps. To add steps you can get a lot of help from the intellisense feature of your code editor. Given a variable that contains an instance of a *Mocklis class*, you can type that variable and a dot to get a list of mock properties to choose from. Select one, and type another dot and it will give you a list of all the valid steps you can add at this point.

Steps can be also chained together for more advanced cases. If a step can forward on call to other steps, type that dot directly after it (without ending the expression with a semi-colon) and intellisense will give you a list of valid steps to choose from.

Let's say that you have mocked an `int` property, where the first time you call it expect the value 120, the second time you expect the value 210, and for any calls after that it should throw a `FileNotFoundException`. The following would do the trick:

```
var mock = new MockSample();
mock.TotalLinesOfCode
    .ReturnOnce(120)
    .ReturnOnce(210)
    .Throw(() => new FileNotFoundException());
```

The `ReturnOnce` steps can forward on calls, while the `Throw` step will always throw an exception and as such cannot chain in a further step. The extension methods used to add steps to mocks are written in such a way that you will get full intellisense and the ability to add steps to `TotalLinesOfCode` and `ReturnOnce`, but will not allow you to add anything to `Throw` (that is to say the `Throw` extension method returns `void`).

Since all steps are added through extension methods on the step type interface, any steps that you create yourself will automatically be available through intellisense.

More details can be found in the [Reference](#) section.

4.3 Work with type parameters

Roslyn, the code analysis and compilation framework that the Mocklis code generator uses, makes some things that look simple very difficult. Fine-tuning layout of code springs to mind. It also makes some things that seem insanely difficult almost trivial. Using type parameters is one such case.

There are two places where you can declare new type parameters, one is in the declaration of a class, struct or interface, and the other is when defining a method.

4.3.1 Type parameters on interfaces and classes

Mocklis can mock interfaces with type parameters, and indeed *Mocklis classes* can themselves be generic. You just need to make sure all types are closed when instantiating the class.

```
public interface IValueReader<out T>
{
    T Value { get; }
}

[MocklisClass]
public class MockValueReader<T> : IValueReader<T>
{
    // implementation removed for brevity
}

// usage:
var mock = new MockValueReader<string>();
mock.Value.Return("Hello world!");
```

Note that the steps remain strongly typed to the choice of type parameters; ‘mock.Value.Return(15)’ wouldn’t have compiled.

If a *MocklisClass* implements more than one interface, either directly or through other interfaces, each will be given its own implementation. A *MocklisClass* implementing `IEnumerable<string>`, which in turn extends `IEnumerable`, will have two different `GetEnumerator` methods; one from each interface, and they will need to be mocked out separately.

For a more extreme example run the code generator on the following class:

```
[MocklisClass]
public class MyDictionary<TKey> : IDictionary<TKey, string>
{
}
```

It will happily expand out all the interfaces necessary for the implementation (such as `ICollection<KeyValuePair<TKey, string>>`), and leave you with a *Mocklis class* you can instantiate with any key type you wish in your tests.

A corner case to be aware of is that you cannot implement interfaces that could unify for some combinations of actual types but not others. The following is an invalid declaration, but not because `IEnumerable` is declared twice. That is perfectly ok. The issue is that if `T` is substituted with ‘int’ then the interface declarations would unify, and otherwise they would remain separate. *That* is invalid.

```
public class Incompatible<T> : IEnumerable<T>, IEnumerable<int>
{
}
```


4.3.2 Type parameters on methods

For type parameters introduced on methods, Mocklis generates code with a slightly different syntax. Let's say you have the following in your interface:

```
public interface ITypeParameters
{
    TOut Test<TIn, TOut>(TIn input) where TOut : struct;
}
```

Now Mocklis will generate a bit more code than normally:

```
[MocklisClass]
public class TypeParameters : ITypeParameters
{
    // The contents of this class were created by the Mocklis code-generator.
    // Any changes you make will be overwritten if the contents are re-generated.

    private readonly TypedMockProvider _test = new TypedMockProvider();

    public FuncMethodMock<TIn, TOut> Test<TIn, TOut>() where TOut : struct
    {
        var key = new[] { typeof(TIn), typeof(TOut) };
        return (FuncMethodMock<TIn, TOut>)_test.GetOrAdd(key, keyString => new
        ↪FuncMethodMock<TIn, TOut>(this, "TypeParameters", "ITypeParameters", "Test" +
        ↪keyString, "Test" + keyString + "()", Strictness.Lenient));
    }

    TOut ITypeParameters.Test<TIn, TOut>(TIn input) => Test<TIn, TOut>().Call(input);
}
```

The difference is that the *mock property* has been replaced with a generic *mock factory method*, and this in turn requires a slightly different syntax when adding steps; where your 'normal' tests used to look like this:

```
var t = new TypeParameters;
t.Test.Return(15); // mock property
```

You'll now write:

```
var t = new TypeParameters;
t.Test<string, int>().Func(int.Parse); // mock factory method
t.Test<int, int>().Func(a => a*2); // mock factory method
```

Your mocks are made 'per type combination', and if you're trying to use the mock with an un-mocked set of type parameters the result depends on the strictness level of your mock. There is no easy way to define a mock 'for all possible combinations of types', so Mocklis doesn't support this. Note however that Mocklis passed on the type constraints to your factory method so you won't be able to add steps to an invalid type combination.

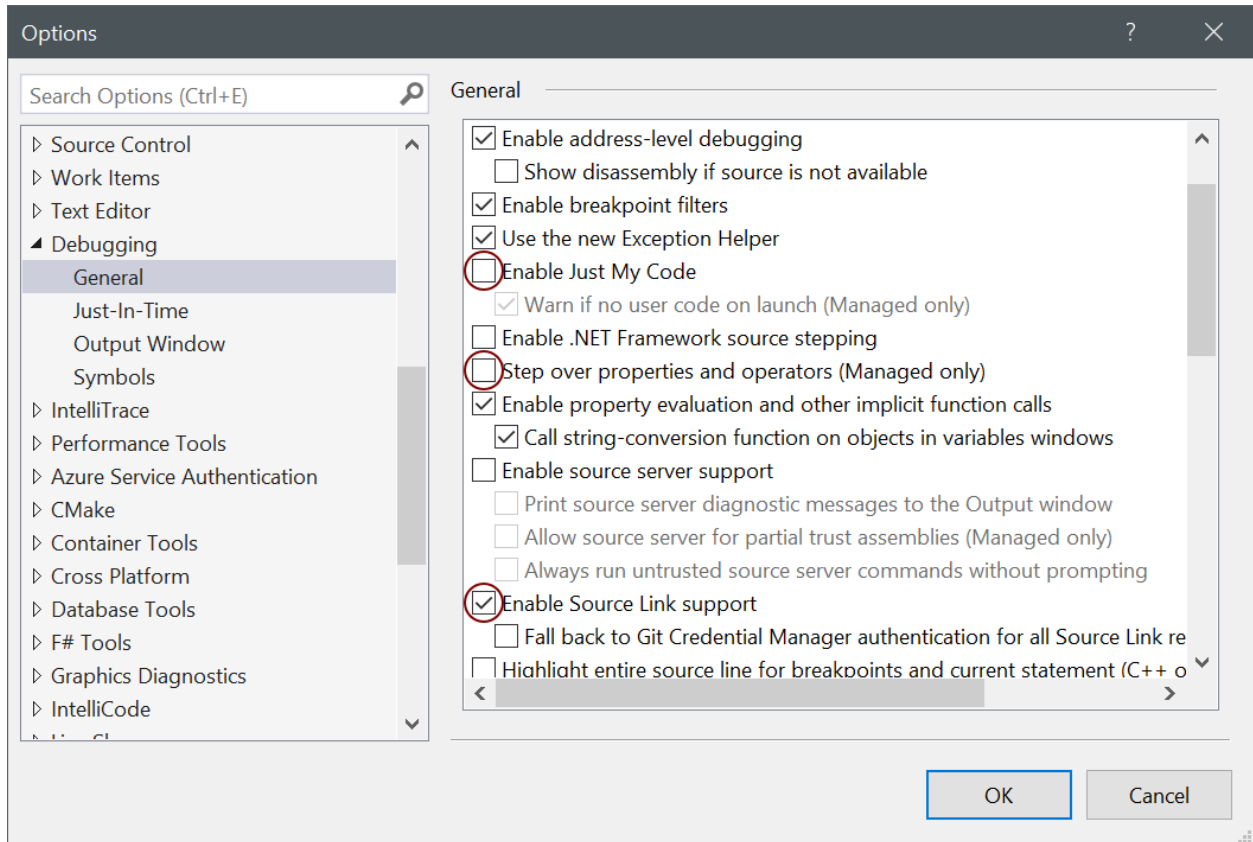
4.4 Debugging tests

There are two approaches to debugging tests that have been taken into account for Mocklis. One is something akin to good old *print*-style debugging where a print statement would log any calls to a specific piece of code. Mocklis has a specific `Log` step that does roughly this.

Then, given that Mocklis generates source code you can easily set breakpoints as you would in any other code, step through code and watch variables as you normally do. The Mocklis libraries themselves are source linked debug

builds, which means that you can get Visual Studio to let you step into the Mocklis source code and set breakpoints in the Mocklis sources almost as easily as you can do that in your own code.

To make this work you'll need to switch off *Just My Code* and enable *Source Links*. Both from the *OptionsDebugging* section in Visual Studio. It's also useful to clear *Step over properties and operators*.



Now you can set breakpoints in your tests and use *step into* (F11) to drill into the Mocklis source code once you're in debug mode. You can set breakpoints in any source file opened in this fashion. If you need to set a breakpoint in a file you don't have opened but you know the name of the member, you can add this with the *New* dropdown in the Breakpoints window, or *New Breakpoint* from the Debug menu.

4.5 Refactoring tests

As the number of tests in your solution grows, it becomes increasingly important to make the test code itself streamlined and easy to work with. The classes you write with Mocklis are just normal code so most of the techniques you use for your normal development work equally well. In some cases you use patterns in your code base that would benefit from having steps tailored for them, we'll go over how to do this in the section on *Extending Mocklis*. Find a few other techniques listed below.

4.5.1 Sharing setup logic

It's a simple thing, but one that is easy to overlook. Since your *Mocklis classes* are just normal classes with source code you can write methods that operate on them. If you have a similar mock setup needed for a number of your tests, you can refactor that logic into a method of its own, or define extension methods on the *Mocklis class*.

4.5.2 Inheritance

The Mocklis code generator will not impose a base class for your *Mocklis classes*, nor will it prevent you from inheriting from them.

The only real restriction is that the *Mocklis classes* must not be partial (as that introduces a whole new level of corner case cacaphony), or static (as you cannot implement an interface ‘statically’ on a class).

But in short the class hierarchy is yours for making the most of; if you want to create a common ancestor for all your mocks you can certainly do so, and if you want to override a *Mocklis class* to centralise configuration or add new functionality just go ahead. Mocklis will create constructors as necessary, all of which will be protected if the *Mocklis class* is abstract and public otherwise.

You can also have *Mocklis classes* inherit from other *Mocklis classes* which lets you mock new interfaces for an existing *Mocklis class*. This could be useful if some of your tests require the mocked out dependency to also be disposable for instance... If you do use the `MocklisClass` attribute at more than one level of the class hierarchy you need to generate the code in the right order, from base class to derived class, otherwise you could get unresolved name clashes.

4.5.3 Invoking Mocks directly

Strictly speaking not something that helps you refactor tests, but still a technique that is useful to know when writing code that interacts with *Mocklis classes*: The *mock properties* that are added to your *Mocklis classes* will let you make the same calls to them as the explicitly implemented interface members would.

The different *MethodMock* classes (*ActionMethodMock* and *FuncMethodMock*) expose a *Call* method. The *PropertyMock* gives you access to a *Value* property, and the *IndexerMock* has an *indexer* defined so you can use it directly as an *indexer*.

It would be nice if the ‘EventMock’ could have an event, but it seems it is not possible to declare an interface with a type from a type variable, regardless of whether it’s restricted to a ‘Delegate’ type. However we have an ‘Add’ and a ‘Remove’ method that will let you do the same thing.

This can be particularly useful when unit testing steps themselves, but it can come in handy for writing normal tests as well.

```
[Fact]
public void SetThroughMock ()
{
    var mock = new MockSample ();
    var stored = mock.TotalLinesOfCode.Stored(0);

    // Write through the mock property
    mock.TotalLinesOfCode.Value = 99;

    // Assert through the stored step
    Assert.Equal(99, stored.Value);
}
```

4.6 What Mocklis can’t do

As with any framework, there have been trade-offs in the design.

Let’s start with the biggest one: Mocklis deals with interfaces only, the reason being that only interface members can be explicitly implemented. This makes things quite a bit easier for us - we don’t need to worry too much about naming clashes (that is to say the code generator does worry greatly about this, but the resulting code will be much less likely

to have them). Then it may be that we want to use the same mocked class for more than one interface, and have the mock handle identical members on different interfaces in different ways.

So if you want to mock members of an abstract base class you can't - unless you're happy to manually write code to create *mock properties* and call them from your overridden members, and either do away with the ability to call 'base' or pass on the base call as another property as a lambda.

Then there are the so-called restricted types, comprised of a handful of core .net classes and ref structs. (The handful of classes are `System.RuntimeArgumentHandle`, `System.ArgIterator`, and `System.TypedReference`, and your ref structs are things like `Span<T>`.) These cannot be cast to object, and cannot be used as type parameters. As Mocklis uses type parameters to fit interface members into one of the four standard forms, these types can not be used by normal Mocklis mocks.

Mocklis will still implement these interface members explicitly, but instead of forwarding calls on to a *mock property* (or *mock factory method*) it will create *virtual methods* whose default implementation is to throw a `MockMissingException`. If you want to create bespoke behaviour you'll have to subclass, and override. This is exactly the same trick as is used by default for some of the *ref returns* cases mentioned earlier.

Having said all of this, Mocklis should be able to provide something that compiles from any interface or (valid combination of) interfaces. In most cases this should result in *mock properties* that you can use steps with. It should also avoid any name clashes, be it clashes with the name of the *Mocklis class* itself, any members defined in base classes, or clashes in type parameter names. If you do come up with a way of tripping up the code generator, please flag this on GitHub so it can be dealt with.

The reference section consists of four parts: standard steps, verifications, code generation and experimental stuff.

5.1 Standard Steps

Find below a discussion of all the steps in the standard Mocklis package that you can use. The list is, perhaps unfortunately, in alphabetic order. This means that some of the more common steps are listed towards the end, but it makes a little bit more sense as a reference section in this way.

5.1.1 Conditional steps

The conditional steps are steps that either branch out or cut short the invocation of a mocked member based on some condition.

The `If` steps branch to a different chain of steps if a given condition holds, with the option of joining the original remaining steps. In their basic form the decision is based on just information passed to the step, parameters to a method or key to an indexer. The `InstanceIf` step lets you make the decision based on the state of the whole mocked instance, and the `IfAdd`, `IfRemove`, `IfGet` and `IfSet` versions branch only for those actions of an event, property or indexer.

Here's a mock setup for an indexer. Note the underlying name used for the indexer, which is used since it would not be well-formed C# to name a property *this[]*. The underlying name is usually *Item* - which is the reason why it's not possible (unless the indexer name has been changed via the `IndexerNameAttribute`) to have a method named `Item` in a class with an indexer.

```
var mock = new MockSample();
mock.Item
    .If(g => g % 2 == 0, null, b => b.Return("Even"))
    .Return("Odd");
```

The `null` argument is for a parameter that decides when to use the if-branch when writing to the indexer.

If steps provide you with a *joinpoint* representing the non-if branch (called 'ElseBranch'). In the following sample we have a property, where both the getter and setter are connected to the same `Stored` step. Only calls to the setter are logged to the console, however.

```
var mock = new MockSample();
mock.TotalLinesOfCode
    .IfSet(b => b.Log().Join(b.ElseBranch))
    .Stored(200);
```

The last sample for a conditional step uses the `OnlySetIfChanged` conditional which only exists for properties and indexers. When an attempt to 'set' a value is made, the step will first try to 'get' the value, check if it's actually changed, and only 'set' the new value if it has.

```
var mock = new MockSampleWithNotifyPropertyChanged();
mock.PropertyChanged
    .Stored(out var pch);
mock.TotalLinesOfCode
    .OnlySetIfChanged()
    .RaisePropertyChangedEvent(pch)
    .Stored();
```

The combination of the `OnlySetIfChanged`, `RaisePropertyChangedEvent` and `Stored` steps is so common that there is a shorthand: `StoredWithChangeNotification`

```
var mock = new MockSampleWithNotifyPropertyChanged();
mock.PropertyChanged
    .Stored(out var pch);
mock.TotalLinesOfCode
    .StoredWithChangeNotification(pch);
```

5.1.2 Dummy steps

The Dummy steps will do as little as possible without throwing an exception. For a property or indexer, the step will do nothing for a setter, and return a default value for a getter. For an event, adding or removing an event handler do absolutely nothing, and for a method, it will not do anything with the parameters, and return default values for anything that needs returning, including out and ref parameters.

Note also that Dummy steps are final - you cannot add anything to follow them.

5.1.3 Join steps

We've already met the `Join` step in the sample code for `If` above, where it allows us to take any step (with the right form - that is member type and type parameters) and use as the next step. The missing piece is a method to designate a step as such a target, which is where the `JoinPoint` comes in.

Let's say that we want to connect two properties to the same `Stored` step. The solution is to add a `JoinPoint` step just before the `Stored` step.

```
var mockDishes = new MockDishes();
mockDishes.Vichyssoise.JoinPoint(out var soup).Stored();
mockDishes.Revenge.Join(soup);

IDishes dishes = mockDishes;
```

(continues on next page)

(continued from previous page)

```
dishes.Vichyssoise = "Best served cold";
Console.WriteLine(dishes.Revenge);
```

Note that any step would do for a `Join`, as long as we can get hold of it. The following would work equally well, taking the `Stored` step itself and using that as a join point:

```
var mockDishes = new MockDishes();
mockDishes.Vichyssoise.Stored(out var soup);
mockDishes.Revenge.Join(soup);
```

5.1.4 Lambda steps

These steps are constructed with either an `Action` or a `Func`, and when they are called the `Action` or `Func` will be run. In the case of `Func` the result of the call will be returned.

The names always contain the word `Action` or the word `Func`, but they are further qualified for non-method steps. Property and indexer steps are called `GetFunc` and `SetAction` while event steps are called `AddAction` and `RemoveAction`.

The lambda steps (and some of the other steps) have ‘instance’ versions where the current instance of the mock is passed as an additional parameter. This parameter is always untyped (well, passed as object), so you’ll need to cast it to one of the mocked interfaces (or the mocking class itself) for it to be of any use. These steps have the names of their non-instance counterparts prefixed with the word `Instance` (so that `InstanceSetAction` would exist as a property step to give an example).

Here’s an example where a `Send` method takes a message of some reference type and returns a `Task`:

```
var mockConnection = new MockConnection();
mockConnection.Send.Func(m => m == null
    ? Task.FromException(new ArgumentNullException())
    : Task.CompletedTask);
```

5.1.5 Log steps

Log steps are your quintessential debugging steps. They won’t do anything except write out anything that passes through them, by default to the console although this can be tailored to your specific needs.

Therefore you can just add in a `.Log()` if you need to figure out what happens with a given mock. Note that they are best added early in a mock step chain if you want to get a faithful representation of what’s being called from the code you are testing, as steps can short-circuit calls or make calls of their own down the chain.

The *Getting Started* makes extensive use of Log steps.

If you’re working with Xunit as your test framework, you probably know that you cannot write to the `Console` and expect the strings written to be part of the test output, and that instead your test class accepts an `ITestOutputHelper` on the constructor. The recommended approach is to have your test class implement the `ILogContextProvider` interface.

```
public class Tests : ILogContextProvider
{
    public ILogContext LogContext { get; }

    protected Tests(ITestOutputHelper testOutputHelper)
    {
```

(continues on next page)

(continued from previous page)

```

    LogContext = new WriteLineLogContext(testOutputHelper.WriteLine);
}
}

```

Then you can replace all your calls to `Log()` with `Log(this)`, and the lines will be written to the `ITestOutputHelper`.

There is also a specific `LogContext` for Serilog 2.x if you add the `Mocklis.Serilog2` NuGet package. Create a `SerilogContext` with an `ILogger` that has a test-framework compatible sink, and set `LogContext` to that as in the example above.

5.1.6 Miscellaneous steps

Stuff that couldn't really be placed in an existing category, and would have constituted a 'one-step-only' category if pushed...

Currently this (possibly expanding) category contains just the `RaisePropertyChangedEvent` step you saw in the last example of the Conditional steps category.

5.1.7 Missing steps

When one of these steps is invoked, it will throw a `MockMissingException` with information about the *mock property* itself.

The exception thrown could look something like this:

```
Mocklis.Core.MockMissingException: No mock implementation found for getting value of Property 'ISample.TotalLinesOfCode'. Add one using 'TotalLinesOfCode' on your 'MockSample' instance.
```

5.1.8 Record steps

These steps will keep track of all the calls that have been made to them, so that you can assert in your tests that the right interactions have happened.

Each of the record steps will cater for one type of interaction only (method call, indexer get, indexer set, property get, property set, event add or event remove), and it will take a `Func` that transforms whatever is seen by the step to something that you want to store. They also provide the 'ledger' with recorded data as an out parameter.

There is currently no mechanism for letting record steps share these 'ledgers' with one another.

```

[Fact]
public void RecordAddedEventHandlers ()
{
    // Arrange
    var mockSamples = new MockSampleWithNotifyPropertyChanged();
    mockSamples.PropertyChanged.RecordBeforeAdd(out var handlingTypes, h => h.Target?.
↪GetType());

    // Act
    ((INotifyPropertyChanged)mockSamples).PropertyChanged += OnPropertyChanged;

    // Assert
    Assert.Equal(new[] { typeof(RecordSamples) }, handlingTypes);
}

```


5.1.9 Repetition steps

The `Times` steps look a little like conditional steps in that they add a separate step chain that can be taken. They differ from the `if`-step in that they cannot join back to the normal path, and that the separate path will only be used a given number of times.

In the current version a `get` or a `set` both count as a usage from the same pool for property and indexer mocks, as do `adds` and `removes` for an event mock.

For a sample see the next section, return steps.

5.1.10 Return steps

Arguably the most important step of them all. The `Return` step, only useable in cases where some sort of return value is expected, will simply return a value.

There are three versions, one that just returns a given value once, and passes calls on to subsequent steps on later calls, one that returns items from a list one by one, and one that returns the same value over and over.

Here's code that shows how to use these, and the repetition step:

```
var mock = new MockSample();
mock.GuessTheSequence
    .Times(2, m => m.Return(1))
    .ReturnOnce(int.MaxValue) // should really be infinity for this sequence
    .ReturnEach(5, 6)
    .Return(3);

var systemUnderTest = (ISample)mock;

Assert.Equal(1, systemUnderTest.GuessTheSequence);
Assert.Equal(1, systemUnderTest.GuessTheSequence);
Assert.Equal(int.MaxValue, systemUnderTest.GuessTheSequence);
Assert.Equal(5, systemUnderTest.GuessTheSequence);
Assert.Equal(6, systemUnderTest.GuessTheSequence);
Assert.Equal(3, systemUnderTest.GuessTheSequence);
Assert.Equal(3, systemUnderTest.GuessTheSequence);
Assert.Equal(3, systemUnderTest.GuessTheSequence);
Assert.Equal(3, systemUnderTest.GuessTheSequence);
```

5.1.11 Stored steps

If the `Return` steps are the most used steps, the `Stored` steps are definitely the first runners up. These steps are defined for properties, playing backing field to the mocked property. They are also defined for indexers, where the backing structure is a dictionary which has the default return value for all non-set keys.

When creating a `Stored` step for a property you can give it an initial value, and for both properties and indexers you can use verifications to check that the stored value has been set correctly by the components that are under test.

`Stored` steps are also used with events where the steps act as storage for added event handlers. If you have a reference to the `Stored` step you can raise events on these handlers, simply by calling `Invoke` on the stored value. Alternatively, if your handler type is a generic `EventHandler<>` or one of a handful of very common event handler types including `PropertyChangedEventHandler` and the basic `EventHandler`, the `Mocklis` library provides you with `Raise` extension methods. These can be found in the `Mocklis.Verification` namespace.

```
[Fact]
public void RaiseEvent ()
{
    var mock = new MockSample ();
    mock.MyEvent.Stored<EventArgs> (out var eventStep);
    bool hasBeenCalled = false;

    ISample sample = mock;
    sample.MyEvent += (s, e) => hasBeenCalled = true;

    eventStep.Raise (null, EventArgs.Empty);
    // equivalent: eventStep.EventHandler?.Invoke (null, EventArgs.Empty);
    Assert.True (hasBeenCalled);
}
```

For indexers the step is called `StoredAsDictionary` as it holds different values for different keys. It will return a default value rather than throw if an empty slot is read from.

5.1.12 Throw steps

Super easy - with these steps you provide a `Func` that creates an exception. When called, the step will call this `Func` and throw the exception it returns.

5.1.13 Verification steps

Verification steps are steps that track some condition that can be checked and asserted against.

`ExpectedUsage` steps take a verification group as a parameter, along with the number of time they expect the mocked member to be called (which are tracked individually for getters, setters, adds, removes and plain method calls).

To get access to all steps and checks (see next section) for verifications you need to have the namespace `Mocklis.Verification` in scope via a using statement at the top of your file.

5.2 Verifications

If steps provide a means of creating behaviour for the system under test, verifications provide a means of checking that those behaviours have been used in the right way by the system under test.

Verifications come in two flavours. As normal steps they check data as it passes through them:

```
var vg = new VerificationGroup ();
var mock = new MockSample ();
mock.DoStuff
    .ExpectedUsage (vg, "DoStuff", 1);
...
vg.Assert ();
```

... and also as 'checks' that verify some condition of an existing step:

```
[Fact]
public void JustChecks ()
{
    var vg = new VerificationGroup ();
    var mock = new MockSample ();
    mock.TotalLinesOfCode
        .Stored (50)
        .CurrentValueCheck (vg, "TLC", 60);

    ISample sample = mock;
    sample.TotalLinesOfCode = 60;

    vg.Assert ();
}
```

These are the only verifications in the framework at the moment. The expected usage steps work for all different member types, and track the different access methods independently. The current value checks exist for properties and indexers only, where the latter takes a list of key-value pairs to check.

To check that verifications have been met, call `Assert` on the top-most verification group, as done in the last example.

5.3 Mocklis Code Generation

An interface in C# can contain four different types of members: events, methods, properties and indexers. However each of them is just syntactic sugar over one or two method calls. In Mocklis we represent each with a generic interface that encapsulates these method calls.

```
public interface IEventStep<in THandler> where THandler : Delegate
{
    void Add(IMockInfo mockInfo, THandler value);
    void Remove(IMockInfo mockInfo, THandler value);
}

public interface IIndexerStep<in TKey, TValue>
{
    TValue Get(IMockInfo mockInfo, TKey key);
    void Set(IMockInfo mockInfo, TKey key, TValue value);
}

public interface IMethodStep<in TParam, out TResult>
{
    TResult Call(IMockInfo mockInfo, TParam param);
}

public interface IPropertyStep<TValue>
{
    TValue Get(IMockInfo mockInfo);
    void Set(IMockInfo mockInfo, TValue value);
}
```

Ignoring the `IMockInfo` parameter for the moment, these represent what the different member types do, except they are modelled as if indexers always have *one* key, methods always *one* parameter and *one* result. Thanks to the value tuple feature we can pretend that multiple values are one.

Now it's just a question of transforming any interface member into one of these four standard forms, and this is done by the code generated by Mocklis.

In the next few sections we'll go over the different things that Mocklis can generate for us. There are a number of corner cases in particular to do with naming, and we won't go over all of them here. For a reasonably complete set of cases see the `Mocklis.MockGenerator.Tests` project in the Mocklis source code.

5.3.1 Event mocks

The simplest (as in has the fewest special cases) thing to implement is events. An event has to be of a delegate type and is always represented by an Add/Remove pair of methods, which is exactly what the `IEventStep` interface models.

Let's say that we have an interface with an event.

```
public interface ITestClass
{
    event EventHandler MyEvent;
}
```

The generated code for such an interface consists of three parts. The explicitly implemented event itself just forwards adds and removes to an `EventMock`, which is itself created in the constructor and exposed as a property.

```
[MocklisClass]
public class TestClass : ITestClass
{
    // The contents of this class were created by the Mocklis code-generator.
    // Any changes you make will be overwritten if the contents are re-generated.

    public TestClass()
    {
        MyEvent = new EventMock<EventHandler>(this, "TestClass", "ITestClass",
↪"MyEvent", "MyEvent", Strictness.Lenient);
    }

    public EventMock<EventHandler> MyEvent { get; }

    event EventHandler ITestClass.MyEvent { add => MyEvent.Add(value); remove =>
↪MyEvent.Remove(value); }
}
```

That's really all there is to it. A common question is how to raise events. The fact that an event itself has little to do with raising events is a common C# 'gotcha'. The event is only about combining event handlers through the add and remove accessors. To raise an event you need to call `Invoke` on the resulting combined handler. In Mocklis you need to add a `Stored` step to an event in order to correctly remember and combine handlers so you have something to raise the event on. Then you can use the handler exposed by the `Stored` step. See the documentation for a `Stored` step above for a complete example.

Events can be generic-ish. For some reason it's not possible to have an event of type parameter type, even if that type parameter is constrained to delegate type. But you can have an event of a generic delegate type, such as `EventHandler<T>`.

5.3.2 Property mocks

Like an event, a property has two accessor methods. In this case one to get a value, and one to set a value. Unlike an event you do not need to use both of them, as a property can be readonly or writeonly. The generated *mock property* doesn't make a distinction, and the generated code for an interface with three string properties with different access looks like this:

```
[MocklisClass]
class TestClass : ITestClass
{
    // The contents of this class were created by the Mocklis code-generator.
    // Any changes you make will be overwritten if the contents are re-generated.

    public TestClass()
    {
        ReadOnly = new PropertyMock<string>(this, "TestClass", "ITestClass", "ReadOnly
↪", "ReadOnly", Strictness.Lenient);
        WriteOnly = new PropertyMock<string>(this, "TestClass", "ITestClass",
↪"WriteOnly", "WriteOnly", Strictness.Lenient);
        ReadWrite = new PropertyMock<string>(this, "TestClass", "ITestClass",
↪"ReadWrite", "ReadWrite", Strictness.Lenient);
    }

    public PropertyMock<string> ReadOnly { get; }

    string ITestClass.ReadOnly => ReadOnly.Value;

    public PropertyMock<string> WriteOnly { get; }

    string ITestClass.WriteOnly { set => WriteOnly.Value = value; }

    public PropertyMock<string> ReadWrite { get; }

    string ITestClass.ReadWrite { get => ReadWrite.Value; set => ReadWrite.Value =
↪value; }
}
```

Properties can be generic. Properties can also be of *restricted type*, in which case the generated code will fall back to virtual methods and you'll need to subclass and override to add behaviour rather than using steps.

```
public interface ITestClass
{
    Span<string> SpanProperty { get; set; }
}

[MocklisClass]
class TestClass : ITestClass
{
    // The contents of this class were created by the Mocklis code-generator.
    // Any changes you make will be overwritten if the contents are re-generated.

    protected virtual Span<string> SpanProperty()
    {
        throw new MockMissingException(MockType.VirtualPropertyGet, "TestClass",
↪"ITestClass", "SpanProperty", "SpanProperty");
    }

    protected virtual void SpanProperty(Span<string> value)
    {
        throw new MockMissingException(MockType.VirtualPropertySet, "TestClass",
↪"ITestClass", "SpanProperty", "SpanProperty");
    }

    Span<string> ITestClass.SpanProperty { get => SpanProperty(); set =>
↪SpanProperty(value); }
}
```

(continues on next page)

```
}

```

5.3.3 Indexer mocks

Indexers are, loosely speaking, properties with a parameter list, so most of the discussion for properties goes for indexers as well. Even though indexers are declared with the *this* keyword, they have an internal name (that can be changed with the *IndexerName* attribute), and the default name is *Item*.

In mocklis an indexer has a getter with *one* parameter type, and a return type, and a setter with *one* parameter type and a value type, but indexers can have more than one parameter. Here Mocklis uses *ValueTypes* to turn multiple types into one.

```
public interface ITestClass
{
    string this[int row, int col] { get; set; }
}

[MocklisClass]
class TestClass : ITestClass
{
    // The contents of this class were created by the Mocklis code-generator.
    // Any changes you make will be overwritten if the contents are re-generated.

    public TestClass()
    {
        Item = new IndexerMock<(int row, int col), string>(this, "TestClass",
↪ "ITestClass", "this[]", "Item", Strictness.Lenient);
    }

    public IndexerMock<(int row, int col), string> Item { get; }

    string ITestClass.this[int row, int col] { get => Item[(row, col)]; set =>
↪ Item[(row, col)] = value; }
}

```

Note that the *mock property* uses the internal name of the indexer, it's not possible to expose a *this[]* property. Otherwise anything that goes for a property also goes for an indexer.

5.3.4 Method mocks

We're discussing methods last because these have the largest number of different cases, even though the *IMethodStep* interface only has one member. As for the indexer, condensing multiple parameters into one is done using *ValueTuples*. We cannot encode whether a parameter is *in*, *out* or *ref* in the value tuple, so instead an *out* parameter is added to the return type, and a *ref* parameter is added both as a normal parameter and as part of the return type. This means that we can have multiple return types, and again these are combined into a single *ValueTuple*.

The canonical example is the *TryParse*. Notice that the mock takes a string and returns a (bool, int) pair. By naming the individual types in the *ValueTuple* we get intellisense, and the name *returnValue* is given to the value returned from the mocked method.

```
public interface ITestClass
{
    bool TryParse(string text, out int result);
}

```

(continues on next page)

(continued from previous page)

```

}

[MocklisClass]
class TestClass : ITestClass
{
    // The contents of this class were created by the Mocklis code-generator.
    // Any changes you make will be overwritten if the contents are re-generated.

    public TestClass()
    {
        TryParse = new FuncMethodMock<string, (bool returnValue, int result)>(this,
↪"TestClass", "ITestClass", "TryParse", "TryParse", Strictness.Lenient);
    }

    public FuncMethodMock<string, (bool returnValue, int result)> TryParse { get; }

    bool ITestClass.TryParse(string text, out int result)
    {
        var tmp = TryParse.Call(text);
        result = tmp.result;
        return tmp.returnValue;
    }
}

```

Method mocks can also return values by reference, and like *out* or *ref* parameters, the information that the value is to be returned *by ref* isn't part of the return type and thus cannot be encoded in the type parameters of the mock itself. Mocklis can handle this by treating the mock as if it was a normal *non-ref* method, and then wrap the return value in an object so that we can return a reference at the last minute. Granted, this doesn't provide the performance benefit that is sometimes looked for when using *ref* return values, but for tests this is usually good enough.

```

public interface ITestClass<in TKey, out TValue> where TValue : new()
{
    ref readonly int GetRef();
}

[MocklisClass]
class TestClass<T, U> : ITestClass<T, U> where U : new()
{
    // The contents of this class were created by the Mocklis code-generator.
    // Any changes you make will be overwritten if the contents are re-generated.

    public TestClass()
    {
        GetRef = new FuncMethodMock<int>(this, "TestClass", "ITestClass", "GetRef",
↪"GetRef", Strictness.Lenient);
    }

    public FuncMethodMock<int> GetRef { get; }

    ref readonly int ITestClass<T, U>.GetRef() => ref ByRef<int>.Wrap(GetRef.Call());
}

```

The default is to treat *ref readonly* returns in this manner, while using the virtual method fallback for *ref* returns. This can be controlled with attribute parameters as discussed in the [Using Mocklis](#) section.

Since method calls can have zero (or more) parameters and a void (or non-void) return type, we end up with four different types of methods: nothing->nothing, nothing->something, something->nothing and something->something.

To keep the mock class a little more readable there are four different method mock types (in the example above *FuncMethodMock* was used) that all implement the *ICanHaveNextMethodStep* interface. There are also cases where the steps themselves come in different flavours depending on whether there are parameters and/or return types. The trick used by Mocklis is to represent a missing type with *ValueTuple*, but this also means that there might be more than one valid step to use.

As for properties and indexers, methods can use type parameters introduced by the interface they're defined in. But methods can also introduce type parameters of their own. Since these type parameters won't be closed we cannot create a mock property directly with them - we would need to have individual properties of all possible combinations of types. This is clearly impractical, so instead we create them as needed and store them in a dictionary keyed on the actual types used for each instance.

The generated code therefore contains a mock factory method instead of a mock property.

```
public interface ITestClass
{
    string Write<T>(T param);
}

[MocklisClass]
class TestClass : ITestClass
{
    // The contents of this class were created by the Mocklis code-generator.
    // Any changes you make will be overwritten if the contents are re-generated.

    private readonly TypedMockProvider _write = new TypedMockProvider();

    public FuncMethodMock<T, string> Write<T>()
    {
        var key = new[] { typeof(T) };
        return (FuncMethodMock<T, string>)_write.GetOrAdd(key, keyString => new_
↪FuncMethodMock<T, string>(this, "TestClass", "ITestClass", "Write" + keyString,
↪"Write" + keyString + "()", Strictness.Lenient));
    }

    string ITestClass.Write<T>(T param) => Write<T>().Call(param);
}
```

5.3.5 Constructors

The mocks are initialised in the constructor. For a *MocklisClass* that doesn't derive from another class (that is to say that derives directly from *object*) a default constructor will be added if there are mocks to initialise. The constructors are *protected* if the *MocklisClass* is declared as abstract, and *public* otherwise.

If the *MocklisClass* does derive from another class, all public and protected constructors from that base class will be given a corresponding constructor in the *MocklisClass*, passing on parameters as necessary.

If you look at the constructors in the examples given, each of the *mock properties* take a couple of parameters, a reference to the mock instance itself, and a couple of strings with the name of the mock class, the names of the interface and member, and the name of the *mock property* (which often but not always is the same as the name of the member). It also takes the strictness used when creating the *MocklisClass* so that it can react correctly in the cases where the configuration is missing or incomplete. This is exactly what can be found in the *IMockInfo* interface that is on every call on every I-membertype-Step interface. Steps can take advantage of this information if they want to; indeed the *Missing* step picks up the information from this parameter to provide the best possible exception message for the user.

5.3.6 Name clashes

There are cases where the generated code for mocks would clash with either each other or with identifiers already declared in base classes. In these cases Mocklis will add a numerical suffix to the introduced identifier. To take a very simple example, `IEnumerable<T>` derives from `IEnumerable`, and both have a `GetEnumerator` method. The generated code looks like the following, and unfortunately you have to know which method you want to add a step to and use the corresponding name.

```
[MocklisClass]
class TestClass : IEnumerable<int>
{
    // The contents of this class were created by the Mocklis code-generator.
    // Any changes you make will be overwritten if the contents are re-generated.

    public TestClass()
    {
        GetEnumerator = new FuncMethodMock<IEnumerator<int>>(this, "TestClass",
↪ "IEnumerable", "GetEnumerator", "GetEnumerator", Strictness.Lenient);
        GetEnumerator0 = new FuncMethodMock<System.Collections.IEnumerator>(this,
↪ "TestClass", "IEnumerable", "GetEnumerator", "GetEnumerator0", Strictness.Lenient);
    }

    public FuncMethodMock<IEnumerator<int>> GetEnumerator { get; }

    IEnumerator<int> IEnumerable<int>.GetEnumerator() => GetEnumerator.Call();

    public FuncMethodMock<System.Collections.IEnumerator> GetEnumerator0 { get; }

    System.Collections.IEnumerator System.Collections.IEnumerable.GetEnumerator() =>
↪ GetEnumerator0.Call();
}

```

Name clashes can also appear in the type parameter names, and in the types used to create a *ValueTuple*. When creating a *ValueTuple* we could clash with the default names *Item1*, *Item2* and so forth, so Mocklis will rename these as well when needed.

5.3.7 Type parameter substitutions

The type parameter names declared by an interface, and the type parameter names used when referencing that interface do not need to be the same. Mocklis does substitutions where necessary, and sorts out situations where there would be a name clash.

Here's a simple example - the interface calls the key `TKey`, but the class uses `T`, therefore all instances of `TKey` in the interface have been replaced with `T` in the class, and the same goes for `TValue` and `U`.

```
public interface ITestClass<in TKey, out TValue>
{
    TValue GetValue(TKey key);
}

[MocklisClass]
class TestClass<T, U> : ITestClass<T, U>
{
    // The contents of this class were created by the Mocklis code-generator.
    // Any changes you make will be overwritten if the contents are re-generated.

```

(continues on next page)

(continued from previous page)

```

public TestClass()
{
    GetValue = new FuncMethodMock<T, U>(this, "TestClass", "ITestClass", "GetValue
↔", "GetValue", Strictness.Lenient);
}

public FuncMethodMock<T, U> GetValue { get; }

U ITestClass<T, U>.GetValue(T key) => GetValue.Call(key);
}

```

5.4 Experimental Stuff

Mocklis has a project & associated NuGet package for experimental things: `Mocklis.Experimental`. It is meant for things that are in a bit of flux and may either graduate to the main `Mocklis` package, or be found wanting and deleted. Think of it as an incubation space for new functionality. It will not follow the versioning of the other `Mocklis` NuGet packages, but will stay perpetually pre-release.

At the moment it only contains the `Gate` step.

5.4.1 Gate steps

The idea behind the `Gate` step is that it will complete a `Task` (as in `Task Parallel Library`), when the step is called. The `Task` can then be used to drive other things happening in the step, effectively forcing a strict ordering of events in the face of many threads running.

The syntax is still very experimental - it currently only exists for 'Method' mocks, and might well be killed off altogether...

```

public async Task SuccessfulPing()
{
    // Arrange
    var mockConnection = new MockConnection();
    mockConnection.Send
        .Gate(out var sendGate)
        .Return(Task.CompletedTask);
    mockConnection.Receive
        .Stored<MessageEventArgs>(out var messageReceive);
    var pingService = new PingService(mockConnection);

    // Act
    var ping = pingService.Ping();
    await sendGate;
    messageReceive.Raise(mockConnection, new MessageEventArgs(new Message(
↔"PingResponse")));
    var pingResult = await ping;

    // Assert
    Assert.True(pingResult);
}

```

6.1 Writing new steps

The best way to learn about writing steps is to look at the source code for existing steps. But in the interest of documentation, here's a sample.

Disclaimer: This is a silly example. You would never write test code that depends on the time of day. It was chosen because you can be absolutely certain that this step won't ever clash with anything in the Mocklis libraries themselves. (We sincerely hope...)

6.1.1 Phase 1: Write a step

If you are writing a 'final' step, implement the `I-memberType-Step` interface. You just need to implement this interface and you're done.

If you are writing a non-final step, consider (as in: it is very strongly recommended) subclassing the `memberType-StepWithNext` class, and override the `I-memberType-Step` members as you see fit. Otherwise you need to implement strictness checks yourself, and the base class will give you overridable members to plug in your specific functionality where the base implementation will forward to a next step. Let's say we're writing a step to nudge our overworked developers to go home by starting to throw exceptions after 5 o'clock.

Let's also say we're writing this for a property. We'll end up with something like this:

```
public class EndOfDayPropertyStep<TValue> : PropertyStepWithNext<TValue>
{
    private readonly int _cutOffHour;

    public EndOfDayPropertyStep(int cutOffHour)
    {
        _cutOffHour = cutOffHour;
    }

    private void ThrowIfLate()

```

(continues on next page)

(continued from previous page)

```

{
    if (DateTime.Now.Hour >= _cutOffHour)
    {
        throw new Exception("It's late - start considering calling it a day.");
    }
}

public override TValue Get(IMockInfo mockInfo)
{
    ThrowIfLate();
    return base.Get(mockInfo);
}

public override void Set(IMockInfo mockInfo, TValue value)
{
    ThrowIfLate();
    base.Set(mockInfo, value);
}
}

```

6.1.2 Phase 2: Write an extension method

If you wanted to use the new step as is, you would have to create an instance of it and feed to the `SetNextStep` method of the previous step. To enable the fluent syntax you'll need to add the step as an extension method on the `IPropertyStepCaller` interface.

```

public static class EndOfDayStepExtensions
{
    public static IPropertyStepCaller<TValue> EndOfDay<TValue>(
        this IPropertyStepCaller<TValue> caller,
        int? cutOffHour = null)
    {
        return caller.SetNextStep(new EndOfDayPropertyStep<TValue>(cutOffHour ?? 17));
    }
}

```

Notice the naming convention: The `EndOfDayPropertyStep` is added as an extension method named `EndOfDay`, taking an `IPropertyStepCaller` as its 'this' parameter. An `EndOfDayMethodStep` would be added as an extension method also named `EndOfDay`. There is no risk of a naming clash, as the parameter types will differ.

Now you can use your new step:

```

var mock = new MockSample();
mock.TotalLinesOfCode
    .EndOfDay()
    .Return(50);

```

With the obvious (well - depending on what time it is) result:

```

System.Exception: It's late - start considering calling it a day.

```

6.1.3 Phase 3: Generalise

The last phase is to look at your newly created step and consider whether it can be used in other situations. You should extend the step to the different member types if possible.

In some cases the way a step works could depend on the complete state of the mock instance. In these cases you should add new steps with the same name as your existing ones, but prefixed with 'Instance'. For this version you pass on the `IMockInfo.Instance` to the construct you have that uses the instance. Look at the existing `Lambda` steps for the quintessential implementation, however the `Record` and `If` steps also have instance versions.

If you work with steps for methods, you might need to consider having different versions depending on whether your methods take parameters or not, and whether they return things or not. For the `lambda` steps there are two `FuncMethodStep` classes, and two `ActionMethodStep` classes.

```
public class FuncMethodStep<TParam, TResult> : IMethodStep<TParam, TResult>
{
}

public class FuncMethodStep<TResult> : IMethodStep<ValueTuple, TResult>
{
}

public class ActionMethodStep<TParam> : IMethodStep<TParam, ValueTuple>
{
}

public class ActionMethodStep : IMethodStep<ValueTuple, ValueTuple>
{
}
```

Note how the ones that don't funnel data constrict either `TParam` and/or `TResult` to be of type `ValueTuple` (read: *void* or *unit* depending on how you were brought up). While more than one of these might be eligible for use in a given scenario, the design goal is that there should always be one that doesn't require the user to pass manually created `ValueTuple` instances.

6.2 Writing new verifications

The idea behind Mocklis' verifications is to create a tree of binary checks that can be verified in one go. When verified, a read-only data structure is created, that contains information about all the verifications and whether they were successful or not.

A verification implements the `IVerifiable` interface:

```
public interface IVerifiable
{
    IEnumerable<VerificationResult> Verify();
}
```

... where a truncated version of the `VerificationResult` struct is as follows:

```
public struct VerificationResult
{
    public string Description { get; }
    public IReadOnlyList<VerificationResult> SubResults { get; }
    public bool Success { get; }
```

(continues on next page)

(continued from previous page)

```

public VerificationResult(string description, bool success)
{
    Description = description;
    SubResults = Array.Empty<VerificationResult>();
    Success = success;
}

public VerificationResult(string description, IEnumerable<VerificationResult>
↳subResults)
{
    Description = description;
    if (subResults is ReadOnlyCollection<VerificationResult> readOnlyCollection)
    {
        SubResults = readOnlyCollection;
    }
    else
    {
        SubResults =
            new ReadOnlyCollection<VerificationResult>(
                subResults?.ToArray() ?? Array.Empty<VerificationResult>());
    }

    Success = SubResults.All(sr => sr.Success);
}
}

```

The first constructor is for leaf nodes, and the second is for branch nodes. Note that if any leaf node fails, all branch nodes up to the root from that leaf node will have failed as well. Therefore if the root succeeds, we can be sure that all leaf nodes will have as well.

Verifications can either be written as steps. These steps implement the `IVerifiable` interface, and the extension method takes a `VerificationGroup` as a parameter and attach the created step to that group.

Let's say we're creating a `Method` step to check that the method has indeed been called. Subclass `MethodStepWithNext`, override `Call` to set a flag that it has been called, and implement `IVerifiable` to return a `VerificationResult`.

```

public override TResult Call(IMockInfo mockInfo, TParam param)
{
    _hasBeenCalled = true;
    return base.Call(mockInfo, param);
}

public IEnumerable<VerificationResult> Verify()
{
    var text = "Method should be called, " +
        (_hasBeenCalled ? "and it has." : "but it hasn't.");
    yield return new VerificationResult(text, _hasBeenCalled);
}

```

Then we add the step to the verification group in its extension method:

```

public static IMethodStepCaller<TParam, TResult> HasBeenCalled<TParam, TResult>(
    this IMethodStepCaller<TParam, TResult> caller,
    VerificationGroup collector)
{
    var step = new HasBeenCalledMethodStep<TParam, TResult>();
}

```

(continues on next page)

(continued from previous page)

```

        collector.Add(step);
        return caller.SetNextStep(step);
    }

```

But we may want to check some condition without it being a step in its own right. All the `Stored` steps (which would be property, indexer and event) implement an interface to directly access what is being stored. An implementation of `IVerifiable` that is not a step in its own right is called a ‘check’, and writing one is straightforward:

Create a class, have it implement `IVerifiable`. Let the constructor take as input anything it needs to verify that the condition for the verification has been met. In the case of the `CurrentValuePropertyCheck` that checks that a `Stored` property step has the right value this includes:

- The `IStoredProperty` to check the value of.
- A string that allows us to give the verification a name to identify it by. This is generally a recommended thing to do.
- The expected value.
- An equality comparer to check that the value is right, where the default null will be replaced with `EqualityComparer.Default`.

Then the `Verify` method checks the condition and returns one or more verification results.

The extension method is slightly different from the one used for steps. For one thing there is no chaining going on through a `SetNextStep` method. Just use the interface exposed as a ‘this’ parameter, add a `VerificationGroup`, use the former to create the check instance and the latter to make the check available from the group. Then it can just return the access interface again if we want to attach more checks.

Something like this:

```

public static IStoredProperty<TValue> CurrentValueCheck<TValue>(
    this IStoredProperty<TValue> property,
    VerificationGroup collector,
    string name,
    TValue expectedValue,
    IEqualityComparer<TValue> comparer = null)
{
    collector.Add(new CurrentValuePropertyCheck<TValue>(property, name, expectedValue,
    ← comparer));
    return property;
}

```

6.3 Writing a new logging context

This has got to be a very rare occurrence. Given that the `Log` steps are mainly there to aid in debugging your mocks, the default behaviour to just write log statements to the console is normally good enough.

If you need to write them to somewhere else, such as to an xUnit `ITestOutputHelper`, you can pass an `Action<string>` to the `WriteLineLogContext` constructor and pass that to the `Log` steps.

However if you need to do more advanced stuff, such as logging mock interactions as structured data you can create a bespoke implementation of the `ILogContext` interface. This interface has individual methods for all different logging calls made by Mocklis. Implementing it should be a straightforward, if boring, exercise, and you can always look at the source code for the `Mocklis.Serilog2` for an example of how it can be done.

Frequently Asked Questions

7.1 ValueTuple

When I'm creating mocks for methods I often come across mock steps that take "ValueTuple" as parameters, but I didn't have anything like that in my original interfaces - what's going on?

Mocklis strives to map members of interfaces into one of four formats (for events, indexers, methods and properties respectively). In the case of a method the format is simply an ability to call the method with one parameter and one return type. If you have many parameters, they will be grouped together as named members of a value tuple, and in the case where you don't have any parameters or a void return type, the 'empty' ValueTuple will be used. It is essentially just a struct with no members, and is the closest thing to a *void* type that the .net framework contains.

For the lambda steps (Func and Action, and the instance versions thereof) there are overloads that require the parameters to be of type ValueTuple, and the action versions require the return type to be a ValueTuple as well. This is simply to provide a nicer and terser syntax, but there is no way to prevent intellisense to suggest the fuller version as well. To make it a little bit more concrete, consider the following interface:

```
public interface IMisc
{
    void SayHello();
    int ScaleByTheAnswer(int p);
}

[MocklisClass]
public class Misc : IMisc
{
    . . .
}
```

When we are creating a mock for the SayHello method, we have four different (but ultimately equivalent) ways to go about it:

```
var misc = new Misc();
```

(continues on next page)

(continued from previous page)

```

misc.SayHello.Action(() =>
{
    Console.WriteLine("Hello");
});

misc.SayHello.Action<ValueTuple>(_ =>
{
    Console.WriteLine("Hello");
});

misc.SayHello.Func<ValueTuple>(() =>
{
    Console.WriteLine("Hello");
    return ValueTuple.Create();
});

misc.SayHello.Func<ValueTuple, ValueTuple>(_ =>
{
    Console.WriteLine("Hello");
    return ValueTuple.Create();
});

```

Of course the compiler infers the type parameters - they're just there for clarity. On the other hand for the `ScaleByTheAnswer` method, we only have one overload that works, since this method both accepts and returns stuff.

```

var misc = new Misc();

misc.ScaleByTheAnswer.Func<int, int>(i =>
{
    return i * 42;
});

```

Again the compiler infers the type parameters, and you can of course shorten the lambda considerably. But in short: you may be presented with steps that use `ValueTuple`, in which case it generally pays to look for versions of those steps that don't.

7.2 “Missing” mock

When my mock is called, it throws a ‘MockMissingException’. But I’m absolutely certain that I did provide a mock implementation. What’s going on?

Update: This now only happens when the `MocklisClass` attribute was declared with a `VeryStrict = true` parameter. The new behaviour (since version 0.2.0-alpha) for both lenient and strict (as in not ‘very strict’) mocks is that all steps assume an implicit `Dummy` step for all further extension points. In ‘very strict’ mode there is an implicit `Missing` step instead and an exception will be thrown.

You can think of the ‘very strict’ mode as ‘treat warnings as errors’. It’s a bit of a pain but it can help find issues with your mocks.

The solution is to chain a next step that does what you want the mock to do, be it a `Dummy` step, a `Return` step or anything else.

With an interface borrowed from the previous faq entry, here is a case which would throw the exception when used:

```
var misc = new Misc();  
misc.ScaleByTheAnswer.Log();
```

The `Log` step will log the call, and then forward to the ‘default’ next step which (perhaps surprisingly) throws. Provide a next step as follows and it doesn’t throw:

```
var misc = new Misc();  
misc.ScaleByTheAnswer.Log().Dummy();
```

And of course it doesn’t have to be `Dummy()`; - looking at the name of the method an appropriate mock might be `.Func(i => i * 42);...`

CHAPTER 8

Indices and tables

- `genindex`
- `modindex`
- `search`