



Oh, The Mockery: Mock/Stub Object Frameworks

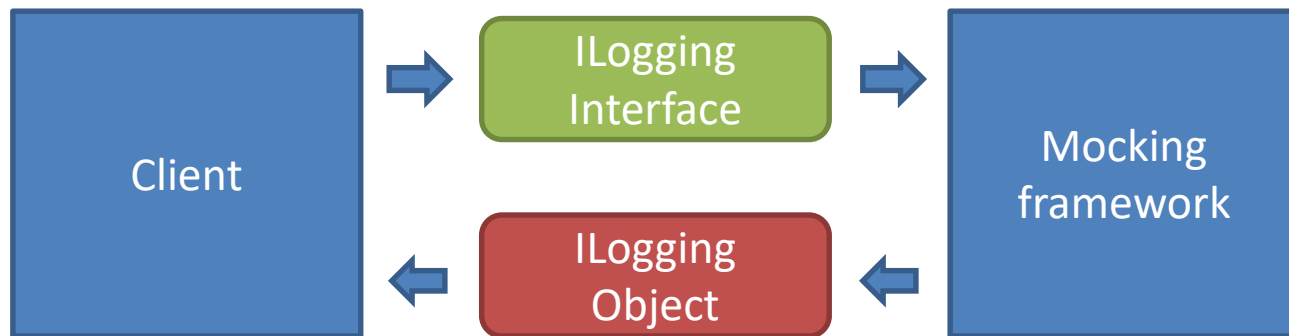
As we've seen in the previous section, we can write stub objects by hand.

However, when our interface has a few more methods, or we only care about a subset of those methods, creating and maintaining the stubs by hand may become tedious.

A **mock or stub framework** uses **dynamic code generation** in order to create test doubles automatically from any interface.

These can be **configured** to return results of our choosing, and can be either pre-programmed with expectations (**mocks**) or record what method calls are made for later inspection (**stubs**).

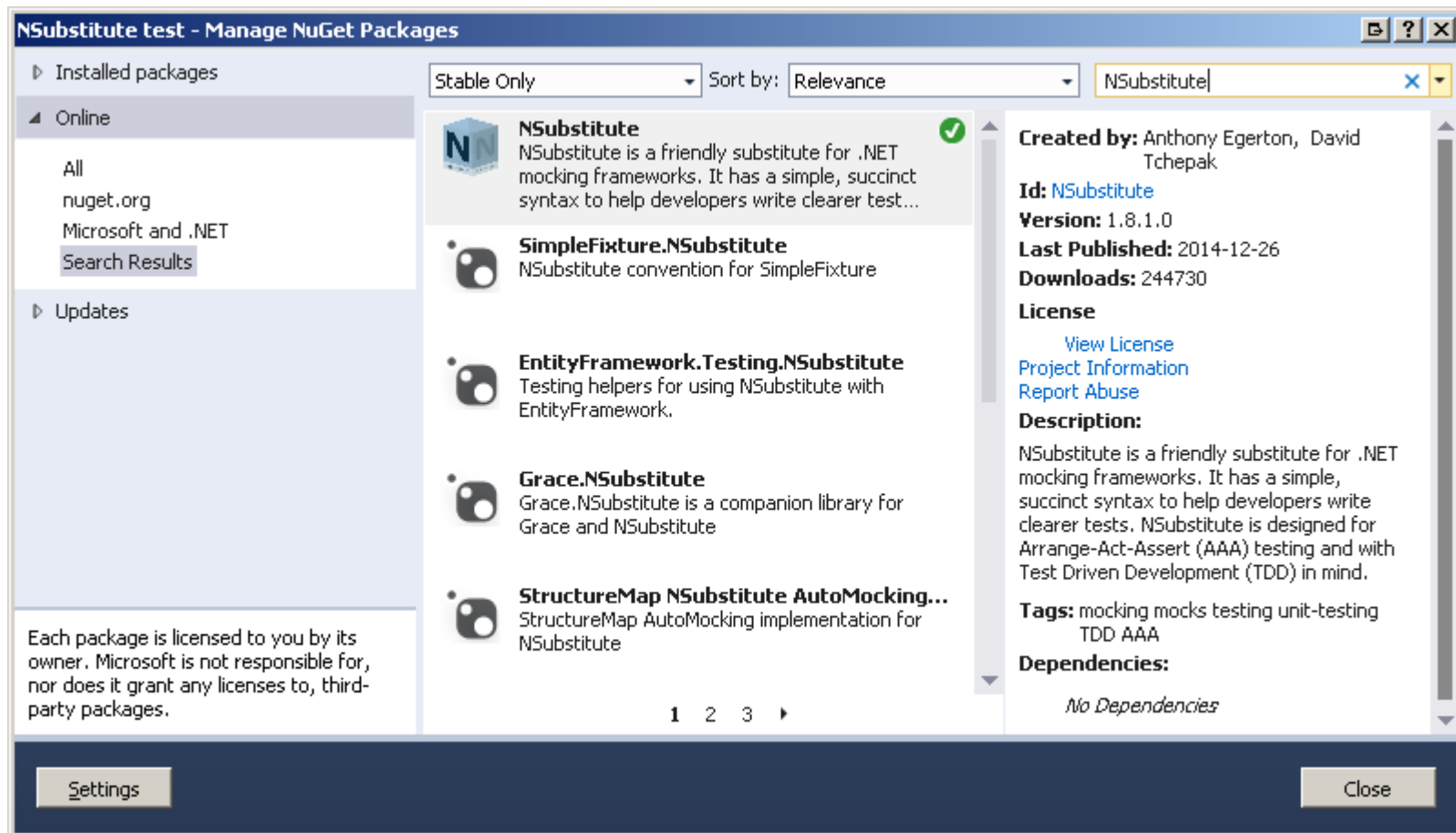
From a Interface, the mocking framework will generate and return a corresponding object.



Sample frameworks

Name	Description
NSubstitute	A friendly substitute for .NET mocking frameworks, we will be using this in this course. (http://nsubstitute.github.io/)
Rhino Mocks	Is a widely used mock and stub testing framework, developed over many years. It is well established, though has many old ways to do things amongst the new, modern ones. Using the modern, AAA approach, it can be rather clean, it is no longer being developed.
Moq	Is also widely used and certainly worth a look. It is built to take advantage of the .Net 3.5 and C# 3 features.
FakeItEasy	Provides a fluent interface and has good VB.Net & C# support. As the name suggests, it tries to make things easy.
TypeMock	Their commercial Isolator product probably has the best supporting for mocking classes (as opposed to interfaces, as we are doing). May be useful in a legacy application.
JustMock	Fast, flexible, fully featured mocking framework

NSubstitute can be installed using **NuGet**. This should also add a reference to it from your test project.



We use the **Substitute.For** method to automatically generate a stub objects for us. We now refer to them just using the interface.

```
[TestFixture]
public class NotifierTests
{
    private ISMSSender smsStub;
    private IPassengerInfoDAL passStub;
    private Notifier sut;

    [SetUp]
    public void Setup()
    {
        smsStub = Substitute.For<ISMSSender>();
        passStub = Substitute.For<IPassengerInfoDAL>();
        sut = new Notifier(smsStub, passStub);
    }
}
```

Our hand-written stub objects can now go away.

We arrange the return value for a method by configuring the generated stub object.

```
passStub.PassengersOnFlight("WP123").Returns(new List<Passenger>
{
    new Passenger {Name = "Dave", Mobile = "123456"},
    new Passenger {Name = "Lena", Mobile = "987654"}
});
```

Note that it will only give these results if the correct flight code is passed, meaning we can safely remove this line from our test:

```
Assert.AreEqual(passStub.flightCodeRequested, "WP123");
```

Rather than writing ad-hoc code to record the parameters passed to a method, Nsubstitute does the recording for us and exposes a uniform interface to make assertions.

We can check the **Send** method was called with the appropriate numbers using the **Received** extension method, using **Arg** constraints in place of the parameters.

```
smsStub.Received().Send(Arg.Is("987654"), Arg.Any<string>());  
smsStub.Received().Send(Arg.Is("123456"), Arg.Any<string>());
```



If we want to be really sure that no calls to **Send** were made besides the two we expected, we can also assert that no calls were made to other numbers.

```
smsStub.DidNotReceive().Send(  
    Arg.Is<string>(m => m != "123456" && m != "987654"),  
    Arg.Any<string>());
```

This time, the **Is** method of **Arg** is used. Rather than trying to build up a massive constraint library, the lambda lets you write code to check that the argument meets your conditions.

Our second test case checked that all the messages sent contained the flight number and did not mention a delay. With NSubstitute, this can be expressed as follows.

```
smsStub.Received(2).Send(Arg.Any<string>(),  
                           Arg.Is<string>(m => m.Contains("WP123") &&  
                                                !m.Contains("delay"))));
```



Call should be
Received twice

Here, an extra constraint is added. This indicates that two matching calls are expected.

In a sense, writing this test means we really need not worry about checking for no extra calls, as done on the previous slide, as this test covers it.

On the one hand, we **didn't have to hand-code** stub objects. That saves some work. On the other hand, they were written in a way that was a good fit for the things our tests were asserting. We **lost that specificity**.

The upshot is that our tests became **a little longer**. However, it's not all that big a difference. In our case, we roughly break even.

Exact results will vary, but it is a bit of **a waterbed situation**.

Our hand-coded stubs can be built with the tests we want to write in mind. On the other hand, if we need to test a wider range of things, they may be more effort to refactor.

Developers being able to **re-use knowledge** of a tool between projects/tests is also a win.

We will continue on the code-base from the previous exercise, but this time we are going to use mocks and DI to implement an email sender to send out confirmation mails whenever a booking gets made.

In order to unit test this behaviour, we will need to create yet another stub object to avoid actually sending mails every time we run our tests. In this case, we won't write the stub by hand, but use the **NSubstitute**-framework.