5 Isolation (mocking) frameworks

This chapter covers

- Understanding isolation frameworks
- Using NSubstitute to create stubs and mocks
- Exploring advanced use cases for mocks and stubs
- Avoiding common misuses of isolation frameworks

In the previous chapter, we looked at writing mocks and stubs manually and saw the challenges involved. In this chapter, we'll look at some elegant solutions for these problems in the form of an *isolation framework*—a reusable library that can create and configure fake objects *at runtime*. These objects are referred to as *dynamic stubs* and *dynamic mocks*.

We'll begin with an overview of isolation frameworks (or mocking frameworks—the word *mock* is too overloaded already) and what they can do. I call them isolation frameworks because they allow you to isolate the unit of work from its dependencies. We'll take a closer look at one specific framework: NSubstitute. You'll see how you can use it to test various things and to create stubs, mocks, and other interesting things.

But NSubstitute (NSub for short) isn't the point here. While using NSub, you'll see the specific values that its API promotes in your tests (readability, maintainability, robust long-lasting tests, and more) and find out what makes an isolation framework good and, alternatively, what can make it a drawback for your tests.

For that reason, later in this chapter, I'll contrast NSub with other frameworks available to .NET developers, compare their API decisions and how they affect test readability, maintainability, and robustness, and finish with a list of things you should watch out for when using such frameworks in your tests.

Let's start at the beginning: what are isolation frameworks?

5.1 Why use isolation frameworks?

I'll start with a basic definition that may sound a bit bland, but it needs to be generic in order to include the various isolation frameworks out there.

> **DEFINITION** An *isolation framework* is a set of programmable APIs that makes creating fake objects much simpler, faster, and shorter than hand-coding them.
>
> Isolation frameworks, when designed well, can save the developer from the need to write repetitive code to assert or simulate object interactions, and if they're designed *very* well, they can make tests last many years without making the developer come back to fix them on every little production code change.

Isolation frameworks exist for most languages that have a unit testing framework associated with them. For example, C++ has mockpp and other frameworks, and Java has jMock and PowerMock, among others. .NET has several well-known ones including Moq, FakeItEasy, NSubstitute, Typemock Isolator, and JustMock. There are also several other isolation frameworks that I don't use or teach anymore because they're either too old or too cumbersome, or they lack many features that the new frameworks have introduced. These include Rhino Mocks, NMock, EasyMock, NUnit.Mocks, and Moles. In Visual Studio 2012, Moles is included and named Microsoft Fakes—and I'd still stay away from it. More on these other tools in the appendix.

Using isolation frameworks instead of writing mocks and stubs manually, as in previous chapters, has several advantages that make developing more elegant and complex tests easier, faster, and less error prone.

The best way to understand the value of an isolation framework is to see a problem and its solution. One problem that might occur when using handwritten mocks and stubs is repetitive code.

Assume you have an interface a little more complicated than the ones shown so far:

```
public interface IComplicatedInterface
    {
        void Method1(string a, string b, bool c, int x, object o);
        void Method2(string b, bool c, int x, object o);
        void Method3(bool c, int x, object o);
    }
```

Creating a handwritten stub or mock for this interface may be time consuming, because you'd need to remember the parameters on a per-method basis, as this listing shows.

Listing 5.1 Implementing complicated interfaces with handwritten stubs

```
class MytestableComplicatedInterface:IComplicatedInterface
    {
        public string meth1_a;
        public string meth1_b,meth2_b;
        public bool meth1_c,meth2_c,meth3_c;
        public int meth1_x,meth2_x,meth3_x;
        public int meth1_0,meth2_0,meth3_0;

        public void Method1(string a,
                            string b, bool c,
                            int x, object o)
        {
            meth1_a = a;
            meth1_b = b;
            meth1_c = c;
            meth1_x = x;
            meth1_0 = 0;
        }

        public void Method2(string b, bool c, int x, object o)
        {
            meth2_b = b;
            meth2_c = c;
            meth2_x = x;
            meth2_0 = 0;
        }

        public void Method3(bool c, int x, object o)
        {
            meth3_c = c;
            meth3_x = x;
            meth3_0 = 0;
        }
    }
```

Manual cumbersome statements

Not only is this handwritten fake time consuming and cumbersome to write, what happens if you want to test that a method is called many times? (Remember in chapter 4, I introduced the word *fake* as anything that looks like a real thing but is not. Based on how it is used, it will be a mock or a stub.) Or what if you want it to return a specific value based on the parameters it receives or to remember all the values for all the method calls on the same method (the parameter history)? The code gets ugly fast.

Using an isolation framework, the code for doing this becomes trivial, readable, and much shorter, as you'll see when you create your first dynamic mock object.

5.2 Dynamically creating a fake object

Let's define *dynamic fake objects* and how they're different from regular, handwritten fakes.

> **DEFINITION** A *dynamic fake object* is any stub or mock that's created at runtime without needing to use a handwritten (hardcoded) implementation of that object.

Using dynamic fakes removes the need to hand-code classes that implement interfaces or derive from other classes, because the needed classes can be generated for the developer at runtime, in memory, and with a few simple lines of code.

Next, we'll look at NSubstitute and see how it can help you overcome some of the problems just discussed.

5.2.1 Introducing NSubstitute into your tests

In this chapter, I'll use NSubstitute (http://nsubstitute.github.com/), an isolation framework that's open source, freely downloadable, and installable through NuGet (available at http://nuget.org). I had a hard time deciding whether to use NSubstitute or FakeItEasy. They're both great, so you should look at both of them before choosing which one to go with. You'll see a comparison of frameworks in the next chapter and in the appendix, but I chose NSubstitute because it has better documentation and supports most of the values a good isolation framework should support. These values are listed in the next chapter.

In the interest of brevity (and ease of typing), I'll refer to NSubstitute from now on as NSub. NSub is simple and quick to use, with little overhead in learning how to use the API. I'll walk you through a few examples, and you can see how using a framework simplifies your life as a developer (sometimes). In the next chapter I go even deeper into some "meta" subjects concerning isolation frameworks, understanding how they work and figuring out why some frameworks can do things others can't. But first, back to work.

To start experimenting, create a class library that will act as your unit tests project, and add a reference to NSub by installing it via NuGet (choose Tools > Package Manager > Package Manager console > Install-Package NSubstitute).

NSub supports the *arrange-act-assert* model, which is consistent with the way you've been writing and asserting tests so far. The idea is to create the fakes and configure them in the *arrange* part of the test, *act* against the product under test, and verify that a fake was called in the *assert* part at the end.

NSub has a class called `Substitute`, which you'll use to generate fakes at runtime. This class has one method with a generic and nongeneric flavor, called `For(type)`, and it's the main way to introduce a fake object into your application when using NSub. You call this method with the type that you'd like to create a fake instance of.

This method then *dynamically* creates and returns a fake object that adheres to that type or interface at runtime. You don't need to implement that new object in real code.

Because NSub is a constrained framework, it works best with interfaces. For real classes, it will only work with nonsealed classes, and for those, it will only be able to fake virtual methods.

5.2.2 Replacing a handwritten fake object with a dynamic one

Let's look at a handwritten fake object used to check whether a call to the log was performed correctly. The following listing shows the test class and the handwritten fake you'd create if you weren't using an isolation framework.

Listing 5.2 Asserting against a handwritten fake object

```
[TestFixture]
class LogAnalyzerTests
{
    [Test]
    public void Analyze_TooShortFileName_CallLogger()
    {
        FakeLogger logger = new FakeLogger();

        LogAnalyzer analyzer = new LogAnalyzer(logger);

        analyzer.MinNameLength= 6;
        analyzer.Analyze("a.txt");

        StringAssert.Contains("too short",logger.LastError);
    }
}

class FakeLogger: ILogger
{
    public string LastError;

    public void LogError(string message)
    {
        LastError = message;
    }
}
```

Creating the fake

Using the fake as a mock object by asserting on it

The parts of the code in bold are the parts that will change when you start using dynamic mocks and stubs.

You'll now create a dynamic mock object and eventually replace the earlier test. The next listing shows how simple it is to fake `ILogger` and verify that it was called with a string.

Listing 5.3 Faking an object using NSub

```
[Test]
public void Analyze_TooShortFileName_CallLogger()
{
  ILogger logger = Substitute.For<ILogger>();
  LogAnalyzer analyzer = new LogAnalyzer(logger);

  analyzer.MinNameLength = 6;
  analyzer.Analyze("a.txt");

  logger.Received().LogError("Filename too short: a.txt");
}
```

1 Creates a mock object that you'll assert against at the end of the test

2 Sets expectation using NSub's API

A couple of lines rid you of the need to use a handwritten stub or mock, because they generate one dynamically ❶. The fake `ILogger` object instance is a dynamically generated object that implements the `ILogger` interface, but there's no implementation inside any of the `ILogger` methods.

From this moment until the last line of the test, all calls on that fake object are automatically recorded, or saved for later use, as in the last line of the test ❷.

In that last line, instead of a traditional assert call, you use a special API—an extension method that's provided by NSub's namespace. `ILogger` doesn't have any such method on its interface called `Received()`. This method is your way of asserting that a method call was invoked on your fake object (thus making it a mock object, conceptually).

The way `Received()` works seems almost like magic. It returns the same type of the object it was invoked on, but it really is used to state what will be asserted on.

If you'd just written in the last line of the test

```
logger.LogError("Filename too short: a.txt");
```

your fake object would treat that method call as one that was done during a production code run and would simply not do anything unless it was configured to do a special action for the method named `LogError`.

By calling `Received()` just before `LogError()`, you're letting NSub know that you really are *asking* its fake object whether or not that method got called. If it wasn't called, you expect an exception to be thrown from the last line of this test. As a readability hint, you're telling the reader of the test a fact: "Something *received* a method call, or this test would have failed."

If the `LogError` method wasn't called, you can expect an error with a message that looks close to the following in your failed test log:

```
NSubstitute.Exceptions.ReceivedCallsException : Expected to receive a call
     matching:
      LogError("Filename too short: a.txt")
Actually received no matching calls.
```

Arrange-act-assert

Notice how the way you use the isolation framework matches nicely with the structure of arrange-act-assert. You start by arranging a fake object, you act on the thing you're testing, and then you assert on something at the end of the test.

It wasn't always this easy, though.

In the olden days (around 2006) most of the open source isolation frameworks didn't support the idea of arrange-act-assert and instead used a concept called record-replay.

Record-replay was a nasty mechanism where you'd have to tell the isolation API that its fake object was in record mode, and then you'd have to call the methods on that object as you expected them to be called from production code.

(continued)

Then you'd have to tell the isolation API to switch into replay mode, and only *then* could you send your fake object into the heart of your production code.

An example can be seen on the Google testing blog: http://googletesting.blogspot.no/2009/01/tott-use-easymock.html.

Asserts, when using these tests, usually involved a simple call to a `verify()` or `verifyAll()` method on the isolation API, with the poor test reader having to go back and figure out what was really expected.

Compared to today's abilities to write tests that use the far more readable arrange-act-assert model, this tragedy cost many developers millions of combined hours in painstaking test reading, to figure out exactly where the test failed.

If you have the first edition of this book, you can see an example of record-replay when I showed Rhino Mocks in this chapter. Ah, good times! Now I stay away from Rhino Mocks, both because its API isn't as good as the new frameworks, and because its maintenance is in question by Oren Eini (http://Ayende.com). It seems Oren, who is known for being a supercoder in many ways, got a life and got married, and so he finally had to start choosing his battles. Rhino Mocks seems to be one of the battles he chose not to fight.

Now that you've seen how to use fakes as mocks, let's see how to use them as stubs, which simulate values in the system under test.

5.3 Simulating fake values

The next listing shows how you can return a value from a fake object when the interface method has a nonvoid return value. For this example, you'll add an `IFileNameRules` interface into the system (see NSubBasics.cs in the book's source code repository).

Listing 5.4 Returning a value from a fake object

```
[Test]
 public void Returns_ByDefault_WorksForHardCodedArgument()
 {
  IFileNameRules fakeRules =   Substitute.For<IFileNameRules>();

  fakeRules.IsValidLogFileName("strict.txt").Returns(true);

  Assert.IsTrue(fakeRules.IsValidLogFileName("strict.txt"));
 }
```

Forces method call to return fake value

What if you didn't care about the argument? It would certainly be a better maintainability tactic if you *always* returned a fake value no matter what, because then you don't care about internal production code changes, and your test would still pass, even if production code calls the method multiple times. It would also help readability, because currently the reader of the test doesn't know if the name of the file is

important. If you can improve their day by removing required information from their reading, they'll have an easier time with your code.

So let's use argument matchers:

```
[Test]
public void Returns_ByDefault_WorksForHardCodedArgument()
{
 IFileNameRules fakeRules =   Substitute.For<IFileNameRules>();

 fakeRules.IsValidLogFileName(Arg.Any<String>())
 .Returns(true);

 Assert.IsTrue(fakeRules.IsValidLogFileName("anything.txt"));
}
```

Ignore the argument value

Notice how you're using the `Arg` class to indicate that you don't care about the input that's required to make this fake value return. This is called an argument matcher, and it's widely used with isolation frameworks to control how arguments are treated, one by one.

What if you wanted to simulate an exception? Here's how to do that with NSub:

```
[Test]
 public void Returns_ArgAny_Throws()
 {
  IFileNameRules fakeRules = Substitute.For<IFileNameRules>();

  fakeRules.When(x =>
                    x.IsValidLogFileName(Arg.Any<string>()))
           .Do(context =>
              { throw new Exception("fake exception"); });

           Assert.Throws<Exception>(() =>
              fakeRules.IsValidLogFileName("anything"));

 }
```

A lambda expression is needed here

Notice how you use `Assert.Throws` to check that an exception is actually thrown.

I'm not crazy about the syntax hoops NSub is forcing you to use here. (This would be easier to do in FakeItEasy, in fact, but NSub has more docs, so I chose to use it here.)

Notice that you have to use a lambda expression here. In the `When` method call, the `x` argument signifies the fake object you're changing the behavior of. In the `Do` call, notice the CallInfo `context` argument. At runtime `context` will hold argument values and allow you to do wonderful things, but you don't need it for this example.

Now that you know how to simulate things, let's make things a bit more realistic and see what we come up with.

5.3.1 A mock, a stub, and a priest walk into a test

Let's combine two types of fake objects in the same scenario. One will be used as a stub and the other as a mock.

You'll use Analyzer2 in the book source code under chapter 5. It's a similar example to listing 4.2 in chapter 4, where I talked about `LogAnalyzer` using a `MailSender`

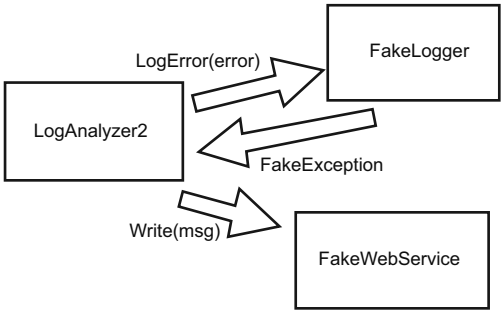

Figure 5.1 The logger will be stubbed out to simulate an exception, and a fake web service will be used as a mock to see if it was called correctly. The whole test will be about how `LogAnalyzer2` interacts with other objects.

class and a `WebService` class, but this time the requirement is that if the logger throws an exception, the web service is notified. This is shown in figure 5.1.

You want to make sure that if the logger throws an exception, `LogAnalyzer2` will notify `WebService` of the problem.

The next listing shows what the logic looks like with all the tests passing.

Listing 5.5 The method under test and a test that uses handwritten mocks and stubs

```
    [Test]
    public void Analyze_LoggerThrows_CallsWebService()          <-- The test
    {
       FakeWebService mockWebService = new FakeWebService();

       FakeLogger2 stubLogger = new FakeLogger2();
       stubLogger.WillThrow = new Exception("fake exception");

        var analyzer2 =
         new LogAnalyzer2(stubLogger, mockWebService);
             analyzer2.MinNameLength = 8;

        string tooShortFileName="abc.ext";
        analyzer2.Analyze(tooShortFileName);

        Assert.That(mockWebService.MessageToWebService,
                        Is.StringContaining("fake exception"));
    }
}

public class FakeWebService:IWebService                  <-- The fake web service you'll use as a mock
{
    public string MessageToWebService;

    public void Write(string message)
    {
        MessageToWebService = message;
    }
}

public class FakeLogger2:ILogger                         <-- The fake logger you'll use as a stub
{
    public Exception WillThrow = null;
    public string LoggerGotMessage = null;
```

```
    public void LogError(string message)
    {
        LoggerGotMessage = message;
        if (WillThrow != null)
        {
            throw WillThrow;
        }
    }
}

//---------- PRODUCTION CODE
public class LogAnalyzer2                          <-- The class under test
{
    private ILogger _logger;
    private IWebService _webService;

    public LogAnalyzer2(ILogger logger,IWebService webService)
    {
        _logger = logger;
        _webService = webService;
    }

    public int MinNameLength { get; set; }

    public void Analyze(string filename)
    {
        if (filename.Length<MinNameLength)
        {
            try
            {
                _logger.LogError(
                string.Format("Filename too short: {0}",filename));
            }
            catch (Exception e)
            {
                _webService.Write("Error From Logger: " + e);
            }
        }
    }
}

public interface IWebService
{
    void Write(string message);
}
```

The next listing shows what the test might look like if you'd used NSubstitute.

Listing 5.6 Converting the previous test into one that uses NSubstitute

```
[Test]
public void Analyze_LoggerThrows_CallsWebService()
{
    var mockWebService = Substitute.For<IWebService>();
    var stubLogger = Substitute.For<ILogger>();
    stubLogger.When(                               <-- Simulates exception on any input
```

```
        logger => logger.LogError(Arg.Any<string>()))
        .Do(info => { throw new Exception("fake exception");});

    var analyzer =
       new LogAnalyzer2(stubLogger, mockWebService);

    analyzer.MinNameLength = 10;
    analyzer.Analyze("Short.txt");

    mockWebService.Received()
     .Write(Arg.Is<string>(s => s.Contains("fake exception")));
}
```

Checks that the mock web service was called with a string containing "fake exception"

The nice thing about this test is that it requires no handwritten fakes, but notice how it's already starting to take a toll on the readability for the test reader. Those lambdas aren't very friendly, to my taste, but they're one of the small evils you need to learn to live with in C#, because those are what allow you to avoid using strings for method names. That makes your tests easier to refactor if a method name changes later on.

Notice that argument-matching constraints can be used both in the simulation part, where you configure the stub, and during the assert part, where you check to see if the mock was called.

There several possible argument-matching constraints in NSubstitute, and the website has a nice overview of them. Because this book isn't meant as a guide to NSub (that's why God created online documentation, after all), if you're interested in finding out more about this API, go to http://nsubstitute.github.com/help/argument-matchers/.

COMPARING OBJECTS AND PROPERTIES AGAINST EACH OTHER

What happens when you expect an object with certain properties to be sent as an argument? For example, what if you'd sent in an `ErrorInfo` object with `severity` and `message` properties, as a call to the `webservice.Write`?

```
[Test]
public void
Analyze_LoggerThrows_CallsWebServiceWithNSubObject()
{
    var mockWebService = Substitute.For<IWebService>();
    var stubLogger = Substitute.For<ILogger>();
    stubLogger.When(
        logger => logger.LogError(Arg.Any<string>()))
        .Do(info => { throw new Exception("fake exception");});

    var analyzer =
       new LogAnalyzer3(stubLogger, mockWebService);

    analyzer.MinNameLength = 10;
    analyzer.Analyze("Short.txt");

    mockWebService.Received()
     .Write(Arg.Is<ErrorInfo>(info => info.Severity == 1000
           && info.Message.Contains("fake exception")));
}
```

Strongly typed argument matcher to the object type you expect

Simple C# "and" to create a more complex expectation on your object

Notice how you can simply use plain-vanilla C# to create compound matchers on the same argument. You want the info being sent in as an argument to have a specific severity *and* a specific message.

Also notice how this impacts readability. As a general rule of thumb, I notice that the more I use isolation frameworks, the less readable the test code turns out, but sometimes it's acceptable enough to use them. This would be a borderline case. For example, if I reach a case where I have more than a single lambda expression in an assert, I question whether using a handwritten fake would have been more readable.

But if you're going to test things in the simplest way, you could compare two objects and simply test the readability. You could create and compare an `expected` object with all the expected properties against the actual object being sent in, as shown here.

Listing 5.7 Comparing full objects

```
[Test]
public void
Analyze_LoggerThrows_CallsWebServiceWithNSubObjectCompare()
{
    var mockWebService = Substitute.For<IWebService>();
    var stubLogger = Substitute.For<ILogger>();
    stubLogger.When(
        logger => logger.LogError(Arg.Any<string>()))
        .Do(info => { throw new Exception("fake exception");});

    var analyzer =
       new LogAnalyzer3(stubLogger, mockWebService);

    analyzer.MinNameLength = 10;
    analyzer.Analyze("Short.txt");

  var expected = new ErrorInfo(1000, "fake exception");
  mockWebService.Received().Write(expected);
}
```

Create the object you expect to receive

Assert that you got exactly the same object (essentially assert.equals())

Testing full objects only works when the following are true:

- It's easy to create the object with the expected properties.
- You want to test *all* the properties of the object in question.
- You know the exact values of each property, fully.
- The `Equals()` method is implemented correctly on the two objects being compared. (It's usually bad practice to rely on the out-of-the-box implementation of `object.Equals()`. If `Equals()` is not implemented, then this test will always fail, because by default `Equals()` will return `false`.)

Also, a note about robustness of the test: Because you won't be able to use argument matchers to ask if a string contains some value in one of the properties when using this technique, your tests are just a little less robust for future changes.

Also, every time a string in an expected property changes in the future, even if it is just one extra whitespace at the beginning or end, your test will fail and you'll have to change it to match the new string. The art here is deciding how much readability you want to give up for robustness over time. For me, perhaps not comparing

a full object but testing a few properties on it with argument matchers could be borderline acceptable, for the added robustness over time. I *hate* changing tests for the wrong reasons.

5.4 Testing for event-related activities

Events are a two-way street, and you can test them in two different directions:

- Testing that someone is listening to an event
- Testing that someone is triggering an event

5.4.1 Testing an event listener

The first scenario we'll tackle is one that I see many developers implement poorly as a test: checking if an object registered to an event of another object.

Many developers choose the less-maintainable and more-overspecified way of checking whether an object's internal state registered to receive an event from another object.

This implementation isn't something I'd recommend doing in real tests. Registering to an event is an internal private code behavior. It doesn't do anything as an end result, except change state in the system so it behaves differently.

It's better to implement this check by seeing the listener object doing something in response to the event being raised. If the listener wasn't registered to the event, then no visible public behavior will be taken, as shown in the following listing.

Listing 5.8 Event-related code and how to trigger it

```
class Presenter
{
    private readonly IView _view;

    public Presenter(IView view)
    {
        _view = view;
        this._view.Loaded += OnLoaded;
    }

    private void OnLoaded()
    {
        _view.Render("Hello World");
    }
}

public interface IView
{
    event Action Loaded;
    void Render(string text);
}

//------ TESTS
[TestFixture]
public class EventRelatedTests
```

```
{
    [Test]
    public void ctor_WhenViewIsLoaded_CallsViewRender()
    {
        var mockView = Substitute.For<IView>();

        Presenter p = new Presenter(mockView);
        mockView.Loaded += Raise.Event<Action>();

        mockView.Received()
         .Render(Arg.Is<string>(s => s.Contains("Hello World")));
    }
}
```

Trigger the event with NSubstitute

Check that the view was called

Notice the following:

- The mock is also a stub (you simulate an event).
- To trigger an event, you have to awkwardly register to it in the test. This is only to satisfy the compiler, because event-related properties are treated differently and are heavily guarded by the compiler. Events can only be directly invoked by their declaring class/struct.

Here's another scenario, where you have two dependencies: a logger and a view. The following listing shows a test that makes sure `Presenter` writes to a log upon getting an error event from your stub.

Listing 5.9 Simulating an event along with a separate mock

```
[Test]
public void ctor_WhenViewhasError_CallsLogger()
{
    var stubView = Substitute.For<IView>();
    var mockLogger = Substitute.For<ILogger>();

    Presenter p = new Presenter(stubView, mockLogger);
    stubView.ErrorOccured +=
         Raise.Event<Action<string>>("fake error");

    mockLogger.Received()
      .LogError(Arg.Is<string>(s => s.Contains("fake error")));
}
```

❶ **Simulate the error**

❷ **Uses mock to check log call**

Notice that you use a stub ❶ to trigger the event and a mock ❷ to check that the service was written to.

Now, let's take a look at the opposite end of the testing scenario. Instead of testing the listener, you'd like to make sure that the event source triggers the event at the right time. The next section shows how you can do that.

5.4.2 Testing whether an event was triggered

A simple way to test the event is by manually registering to it inside the test method using an anonymous delegate. The next listing shows a simple example.

Listing 5.10 Using an anonymous delegate to register to an event

```
[Test]
public void EventFiringManual()
{
      bool loadFired = false;
      SomeView view = new SomeView();
      view.Load+=delegate
                     {
                       loadFired = true;
                     };

      view.DoSomethingThatEventuallyFiresThisEvent();

      Assert.IsTrue(loadFired);
}
```

The delegate simply records whether or not the event was fired. I chose to use a delegate and not a lambda because I think it's more readable. You could also have parameters in the delegate to record the values, and they could later be asserted as well.

Next, we'll take a look at isolation frameworks for .NET.

5.5 Current isolation frameworks for .NET

NSub is certainly not the only isolation framework around. In an informal poll held in August 2012, I asked my blog readers, "Which isolation framework do you use?" See figure 5.2 for the results.

Moq, which in the previous edition of this book was a newcomer in a poll I did then, is now the leader, with Rhino Mocks trailing a bit and losing ground (basically because it's no longer being actively developed). Also changed from the first edition, note that there are many contenders—double the amount, actually. This tells you something about the maturity of the community in terms of recognizing the need for testing and isolation, and I think this is great to see.

FakeItEasy, which may have not even been a blink in its creator's eyes when the first edition of this book came out, is a strong contender for the things that I like in NSubstitute, and I highly recommend that you try it. Those areas (values, really) are listed in the next chapter, when we dive even deeper into the makings of isolation frameworks.

I personally don't use Moq, because of bad error messages and "mock" is used too much in the API. It is confusing since you use mocks also to create stubs.

It's usually a good idea to pick one and stick with it as much as possible, for the sake of readability and to lower the learning curve for team members.

In the book's appendix, I cover each of these frameworks in more depth and explain why I like or dislike it. Go there for a reference list on these tools.

Let's recap the advantages of using isolation frameworks over handwritten mocks. Then we'll discuss things to watch for when using isolation frameworks.

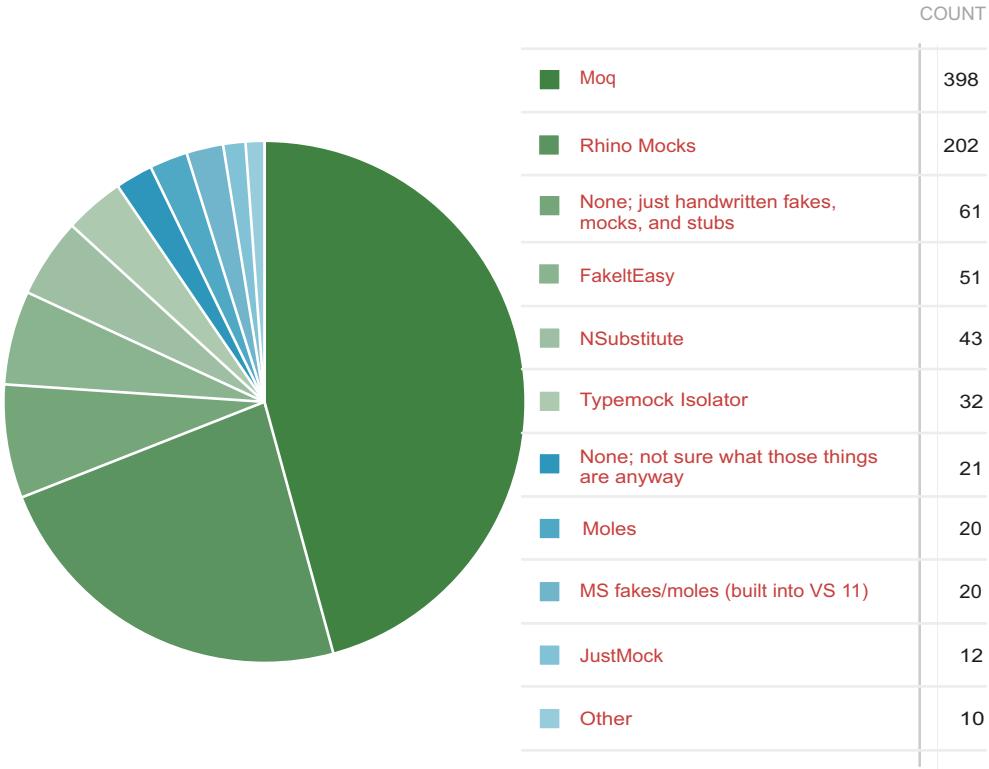

Figure 5.2 Isolation framework usage among my blog readers

Why method strings are bad inside tests

In many frameworks outside the .NET world, it's common to use strings to describe which methods you're about to change the behavior of. Why is this not great?

If you were to change the name of a method in production, any tests using the method in a string would still compile and would only break at runtime, throwing an exception indicating that a method could not be found.

With strongly typed method names (thanks to lambda expressions and delegates), changing the name of a method wouldn't be a problem, because the method is used directly in the test. Any method changes would keep the test from compiling, and you'd know immediately that there was a problem with the test.

With automated refactoring tools like those in Visual Studio, renaming a method is easier, but most refactorings will still ignore strings in the source code. (ReSharper for .NET is an exception. It also corrects strings, but that's only a partial solution that may prove problematic in some scenarios.)

5.6 Advantages and traps of isolation frameworks

From what we've covered in this chapter, you can see distinct advantages to using isolation frameworks:

- *Easier parameter verification*—Using handwritten mocks to test that a method was given the correct parameter values can be a tedious process, requiring time and patience. Most isolation frameworks make checking the values of parameters passed into methods a trivial process even if there are many parameters.
- *Easier verification of multiple method calls*—With manually written mocks, it can be difficult to check that multiple method calls on the same method were made correctly with each having appropriate different parameter values. As you'll see later, this is a trivial process with isolation frameworks.
- *Easier fakes creation*—Isolation frameworks can be used for creating both mocks and stubs more easily.

5.6.1 Traps to avoid when using isolation frameworks

Although there are many advantages to using isolation frameworks, there are possible dangers, such as overusing an isolation framework when a manual mock object would suffice, making tests unreadable because of overusing mocks in a test, or not separating tests well enough.

Here's a list of things to watch out for:

- Unreadable test code
- Verifying the wrong things
- Having more than one mock per test
- Overspecifying the tests

Let's look at each of these in depth.

5.6.2 Unreadable test code

Using a mock in a test already makes the test a little less readable, but still readable enough that an outsider can look at it and understand what's going on. Having many mocks, or many expectations, in a single test can ruin the readability of the test so it's hard to maintain or even to understand what's being tested.

If you find that your test becomes unreadable or hard to follow, consider removing some mocks or some mock expectations or separating the test into several smaller tests that are more readable.

5.6.3 Verifying the wrong things

Mock objects allow you to verify that methods were called on your interfaces, but that doesn't necessarily mean that you're testing the right thing. Testing that an object subscribed to an event doesn't tell you anything about the functionality of that object. Testing that when the event is raised something meaningful happens is a better way to test that object.

5.6.4 Having more than one mock per test

It's considered good practice to test only one concern per test. Testing more than one concern can lead to confusion and problems maintaining the test. Having two mocks in a test is the same as testing several end results of the same unit of work. If you can't name your test because it does too many things, it's time to separate it into more than one test.

5.6.5 Overspecifying the tests

Avoid mock objects if you can. Tests will always be more readable and maintainable when you don't assert that an object was called. Yes, there are times when you can use only mock objects, but that shouldn't happen often.

If more than 5% of your tests have mock objects (not stubs), you might be overspecifying things, instead of testing state changes or value results. In those 5% that use mock objects, you can still overdo it.

If your test has too many expectations (`x.received().X()` and `X.received().Y()` and so on), it may become very fragile, breaking on the slightest of production code changes, even though the overall functionality still works.

Testing interactions is a double-edged sword: test it too much, and you start to lose sight of the big picture—the overall functionality; test it too little, and you'll miss the important interactions between objects.

Here are some ways to balance this effect:

- *Use nonstrict mocks when you can (strict and nonstrict mocks are explained in the next chapter).* The test will break less often because of unexpected method calls. This helps when the private methods in the production code keep changing.
- *Use stubs instead of mocks when you can.* If you have more than 5% of your tests with mock objects, you might be overdoing it. Stubs can be everywhere. Mocks, not so much. You only need to test one scenario at a time. The more mocks you have, the more verifications will take place at the end of the test, but usually only one will be the important one. The rest will be noise against the current test scenario.
- *Avoid using stubs as mocks if humanly possible.* Use a stub only for faking return values into the program under test or to throw exceptions. Don't verify that methods were called on stubs. Use a mock only for verifying that some method was called on it, but don't use it to return values into your program under test. Most of the time, you can avoid a mock that's also a stub but not always (as you saw earlier in this chapter, regarding events).

5.7 Summary

Isolation frameworks are pretty cool, and you should learn to use them at will. But it's important to lean toward return-value or state-based testing (as opposed to interaction testing) whenever you can, so that your tests assume as little as possible about internal implementation details. Mocks should be used only when there's no other

way to test the implementation, because they eventually lead to tests that are harder to maintain if you're not careful.

If more than 5% of your tests have mock objects (not stubs), you might be overspecifying things.

Learn how to use the advanced features of an isolation framework such as NSub, and you can pretty much make sure that anything happens or doesn't happen in your tests. All you need is for your code to be testable.

You can also shoot yourself in the foot by creating overspecified tests that aren't readable or will likely break. The art lies in knowing when to use dynamic versus handwritten mocks. My guideline is that when the code using the isolation framework starts to look ugly, it's a sign that you may want to simplify things. Use a handwritten mock, or test a different result that proves your point but is easier to test.

When all else fails and your code is hard to test, you have three choices: use a super framework like Typemock Isolator (explained in the next chapter), change the design, or quit your job.

Isolation frameworks can help make your testing life much easier and your tests more readable and maintainable. But it's also important to know when they might hinder your development more than they help. In legacy situations, for example, you might want to consider using a different framework based on its abilities. It's all about picking the right tool for the job, so be sure to look at the big picture when considering how to approach a specific problem in testing.

In the next chapter, we'll dig deeper into isolation frameworks and see how their design and underlying implementation affect their abilities.