4 Interaction testing using mock objects

This chapter covers

- Defining interaction testing
- Understanding mock objects
- Differentiating fakes, mocks, and stubs
- Exploring mock object best practices

In the previous chapter, you solved the problem of testing code that depends on other objects to run correctly. You used stubs to make sure that the code under test received all the inputs it needed so that you could test its logic independently.

Also, so far, you've only written tests that work against the first two of the three types of end results a unit of work can have: returning a value and changing the state of the system.

In this chapter, we'll look at how you test the third type of end result—a call to a third-party object. You'll check whether an object calls other objects correctly. The object being called may not return any result or save any state, but it has complex logic that needs to result in correct calls to other objects that aren't under your control or aren't part of the unit of work under test. Using the approach you've employed so far won't do here, because there's no externalized API that you can use to check if something has changed in the object under test.

How do you test that your object interacts with other objects correctly? You use mock objects.

The first thing we need to do is define interaction testing and how it's different from the testing you've done so far—value-based and state-based testing.

4.1 Value-based vs. state-based vs. interaction testing

I defined the three types of end results units of work can generate in chapter 1. Now I'll define interaction testing, which deals with the third kind of result: calling a third party. Value-based testing checks the value returned from a function. State-based testing is about checking for noticeable behavior changes in the system under test, after changing its state.

> **DEFINITION** *Interaction testing* is testing how an object sends messages (calls methods) to other objects. You use interaction testing when calling another object is the end result of a specific unit of work.

You can also think of interaction testing as being action-driven testing. *Action-driven* testing means that you test a particular action an object takes (such as sending a message to another object).

Always choose to use interaction testing only as the last option. This is very important. It's preferable to see if you can use the first two types (value or state) of end-result tests of units of work, because so many things become much more complicated by having interaction tests, as you'll see in this chapter. But sometimes, as is the case of a third-party call to a logger, interactions between objects are the end result. That's when you need to test the interaction itself.

Please note that not everyone agrees with the point of view that mocks should only be used when there are no other ways to test the software under test. In *Growing Object-Oriented Software, Guided by Tests*, Steve Freeman and Nat Pryce advocate what many call "the London school of TDD," which, for design purposes, uses mocks and stubs as a way to merge the design of the software. I'm not fully disagreeing with them that that's a valid way to go about designing your code. But this book is *not* about design, and from a pure maintainability perspective, in my tests using mocks creates more trouble than not using them. That has been my experience, but I'm always learning something new. It's possible that in the next edition of this book I'll have turned 180 degrees on this subject.

Interaction testing, in one form or another, has existed since the first days of unit testing. Back then, there weren't any names or patterns for it, but people still needed to know if one object called another object correctly. Most of the time it was either overdone or done badly, though, and resulted in unmaintainable and unreadable test code. That's why I always recommend tests of the other two types of end results.

To understand some of the pros and cons of interaction testing, let's look at an example. Say you have a watering system, and you've configured the system on when

to water the tree in your yard: how many times a day and what quantity of water each time. Here are two ways to test that it's working correctly:

- *State-based integration test* (yes, integration and not unit test)—Run the system for 12 hours, during which it should water the tree multiple times. At the end of that time, check the state of the tree being irrigated. Is the land moist enough, is the tree doing well, are its leaves green, and so on. It may be quite a difficult test to perform, but assuming you can do it, you can find out if your watering system works. I call this an integration test because it's very, very slooow and running it involves the whole environment around the watering system.
- *Interaction testing*—At the end of the irrigation hose, set up a device that records how much water flows through the device and at what times. At the end of the day, check that the device has been called the right number of times, with the correct quantity of water each time, and don't worry about checking the tree. In fact, you don't even need a tree to check that the system works. You can go further and modify the system clock on the irrigation unit (making it a stub), so that the system thinks that the time to irrigate has arrived, and it will irrigate whenever you choose. That way, you don't have to wait (for 12 hours in this example) to find out whether it works.

As you can see, in this case, having an interaction test can make your life much simpler.

But sometimes state-based testing is the best way to go because interaction testing is too difficult to pull off.

That's the case with crash-test dummies: a car is crashed into a standing target at a specific speed, and after the crash, both car and dummies' states are checked to determine the outcomes. Running this sort of test as an interaction test in a lab can be too complicated, and a real-world state-based test is called for. (People are working on simulating crashes with computers, but it's still not close to testing the real thing.)

Now, back to the irrigation system. What is that device that records the irrigation information? It's a fake water hose, a stub, you could say. But it's a smarter breed of stub—a stub that records the calls made to it, and you use it to define if your test passed or not. That's partly what a mock object is. The clock that you replace with a fake clock? That's a stub, because it just makes happy noises and simulates time so that you can test a different part of the system more comfortably.

> **DEFINITION** A *mock object* is a fake object in the system that decides whether the unit test has passed or failed. It does so by verifying whether the object under test called the fake object as expected. There's usually no more than one mock per test.

A mock object may not sound much different from a stub, but the differences are large enough to warrant discussion and special syntax in various frameworks, as you'll see in chapter 5. Let's look at exactly what the differences are.

Now that I've covered the idea of fakes, mocks, and stubs, it's time for a formal definition of the concept of fakes.

DEFINITION A *fake* is a generic term that can be used to describe either a stub or a mock object (handwritten or otherwise), because they both look like the real object. Whether a fake is a stub or a mock depends on how it's used in the current test. If it's used to check an interaction (asserted against), it's a *mock object*. Otherwise, it's a *stub*.

Let's dive more deeply to see the distinction between the two types of fakes.

4.2 The difference between mocks and stubs

Stubs replace an object so that you can test another object without problems. Figure 4.1 shows the interaction between the stub and the class under test.

The distinction between mocks and stubs is important because a lot of today's tools and frameworks (as well as articles) use these terms to describe different things. It's also important because when you review other people's tests, understanding that you have more than one mock object is an important skill to master (more on that later). There's a lot of confusion about what each term means, and many people seem to use them interchangeably. Once you understand the differences, you can evaluate the world of tools, frameworks, and APIs more carefully and understand more clearly what each does.

At first glance, the difference between mocks and stubs may seem small or nonexistent. The distinction is subtle but important, because many of the mock object frameworks that you'll deal with in the next chapters use these terms to describe different behaviors in the framework. The basic difference is that stubs can't fail tests. Mocks can.

The easiest way to tell you're dealing with a stub is to notice that the stub can never fail the test. The asserts that the test uses are always against the class under test.

On the other hand, the test will use a mock object to verify whether or not the test failed. Figure 4.2 shows the interaction between a test and a mock object. Notice that the assert is performed on the mock.

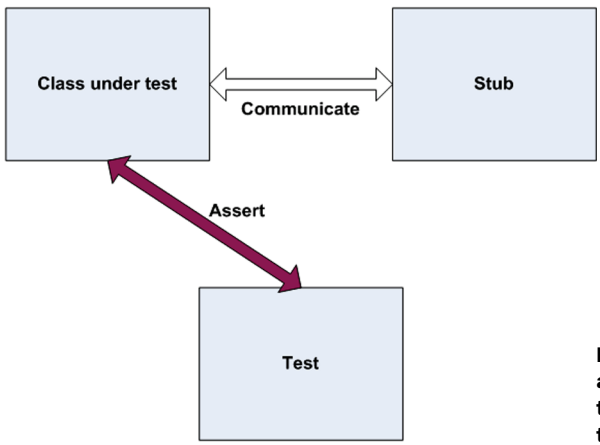

Figure 4.1 When using a stub, the assert is performed on the class under test. The stub aids in making sure the test runs smoothly.

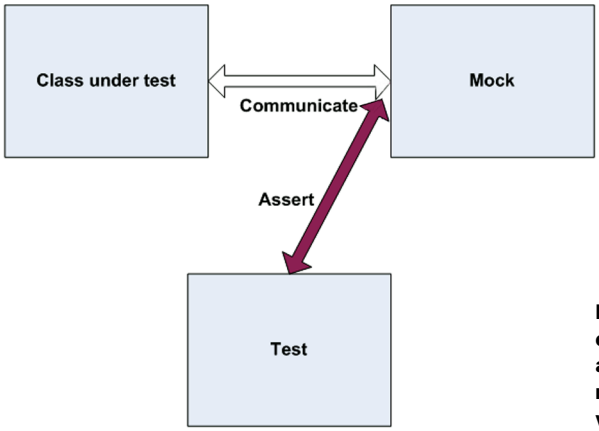

Figure 4.2 The class under test communicates with the mock object, and all communication is recorded in the mock. The test uses the mock object to verify that the test passes.

Again, the mock object is the object you use to see if the test failed or not. Let's look at these ideas in action by building your own mock object.

4.3 A simple handwritten mock example

Creating and using a mock object is much like using a stub, except that a mock will do a little more than a stub: it will save the history of communication, which will later be verified in the form of *expectations*.

Let's add a new requirement to your `LogAnalyzer` class. This time, it will have to interact with an external web service that will receive an error message whenever the `LogAnalyzer` encounters a filename whose length is too short.

Unfortunately, the web service you'd like to test against is still not fully functional, and even if it were, it would take too long to use it as part of your tests. Because of that, you'll refactor your design and create a new interface for which you can later create a mock object. The interface will have the methods you'll need to call on your web service and nothing else.

Figure 4.3 shows how your mock, implemented as `FakeWebService`, will fit into the test.

First off, you'll extract a simple interface that you can use in your code under test, instead of talking directly to the web service:

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

This interface will serve you when you want to create stubs as well as mocks. It will let you avoid an external dependency you have no control over.

Next, you'll create the mock object itself. It may look like a stub, but it contains one extra bit of code that makes it a mock object:

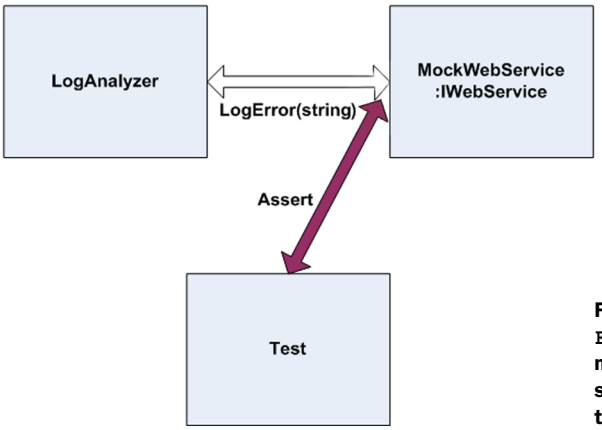

Figure 4.3 Your test will create a `FakeWebService` to record messages that `LogAnalyzer` will send. It will then assert against the `FakeWebService`.

```
public class FakeWebService:IWebService
{
    public string LastError;

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

This handwritten class implements an interface, as a stub does, but it saves some state for later, so that your test can then assert and verify that your mock was called correctly. It's still not a mock object. It will only become one when you *use it as one* in your test.

> **NOTE** According to *xUnit Test Patterns: Refactoring Test Code* by Gerard Meszaros, this would be called a Test Spy.

The following listing shows what the test might look like.

Listing 4.1 Testing the `LogAnalyzer` with a mock object

```
[Test]
 public void Analyze_TooShortFileName_CallsWebService()
  {
     FakeWebService mockService = new FakeWebService();
     LogAnalyzer log = new LogAnalyzer(mockService);
     string tooShortFileName="abc.ext";

     log.Analyze(tooShortFileName);

     StringAssert.Contains("Filename too short:abc.ext",
                           mockService.LastError);
 }

 public class LogAnalyzer
 {
    private IWebService service;
```

Asserts against a mock object

```
    public LogAnalyzer(IWebService service)
    {
        this.service = service;
    }

    public void Analyze(string fileName)
    {
        if(fileName.Length<8)
        {
            service.LogError("Filename too short:"
               + fileName);
        }
    }
}
```

Logs error in production code

Notice how the assert is performed against the mock object and not against the `LogAnalyzer` class? That's because you're testing the interaction between `LogAnalyzer` and the web service. You use the same DI techniques from chapter 3, but this time the mock object (used instead of a stub) also makes or breaks the test.

Also notice that you aren't writing the tests directly inside the mock object code. There are a couple of reasons for this:

- You'd like to be able to reuse the mock object in other test cases, with other asserts on the message.
- If the assert were put inside the handwritten fake class, whoever reads the test would have no idea what you're asserting. You'd be hiding essential information from the test code, which hinders the readability and maintainability of the test.

In your tests, you might find that you need to replace more than one object. We'll look at combining mocks and stubs next. As you'll see, it's perfectly OK to have multiple stubs in a single test, but more than a single mock can mean trouble, because you're testing more than one thing.

4.4 Using a mock and a stub together

Let's consider a more elaborate problem. This time `LogAnalyzer` not only needs to talk to a web service, but if the web service throws an error, `LogAnalyzer` has to log the error to a different external dependency, sending it by email to the web service administrator, as shown in figure 4.4.

Here's the logic you need to test inside `LogAnalyzer`:

```
if(fileName.Length<8)
{
   try
   {
     service.LogError("Filename too short:" + fileName);
   }
   catch (Exception e)
   {
       email.SendEmail("a","subject",e.Message);
   }
}
```

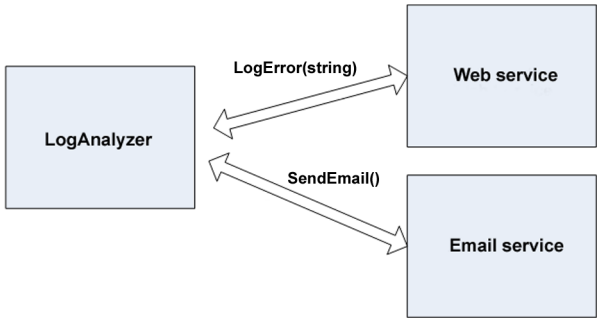

Figure 4.4 `LogAnalyzer` has two external dependencies: web service and email service. You need to test `LogAnalyzer`'s logic when calling them.

Notice that there's logic here that only applies to interacting with external objects; there's no value being returned or system state changed. How do you test that `LogAnalyzer` calls the email service correctly when the web service throws an exception?

Here are the questions you're faced with:

- How can you replace the web service?
- How can you simulate an exception from the web service so that you can test the call to the email service?
- How will you know that the email service was called correctly, or at all?

You can deal with the first two questions by using a stub for the web service. To solve the third problem, you can use a mock object for the email service.

In your test, you'll have two fakes. One will be the email service mock, which you'll use to verify that the correct parameters were sent to the email service. The other will be a stub that you'll use to simulate an exception thrown from the web service. It's a stub because you won't be using the web service fake to verify the test result, only to make sure the test runs correctly. The email service is a mock because you'll assert against it that it was called correctly. Figure 4.5 shows this visually.

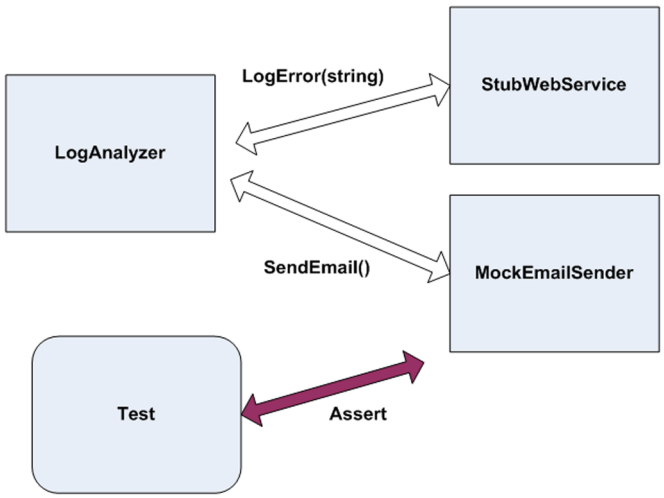

Figure 4.5 The web service will be stubbed out to simulate an exception; then the email sender will be mocked to see if it was called correctly. The whole test will be about how `LogAnalyzer` interacts with other objects.

The next listing shows the code that implements figure 4.5.

Listing 4.2 Testing the `LogAnalyzer` with a mock and a stub

```
public interface IEmailService
    {
        void SendEmail(string to, string subject, string body);
    }

public class LogAnalyzer2
    {
       public LogAnalyzer2(IWebService service, IEmailService email)
        {
             Email = email,
             Service = service;
        }
        public IWebService Service
        {
            get ;
            set ;
        }

        public IEmailService Email
        {
            get ;
            set ;
        }

        public void Analyze(string fileName)
        {
            if(fileName.Length<8)
            {
                try
                {
                  Service.LogError("Filename too short:" + fileName);
                }
                catch (Exception e)
                {
                    Email.SendEmail("someone@somewhere.com",
                                    "can't log",e.Message);
                }
            }
        }
    }

    [TestFixture]
     public class LogAnalyzer2Tests
     {
        [Test]
         public void Analyze_WebServiceThrows_SendsEmail()
         {
            FakeWebService stubService = new FakeWebService();
            stubService.ToThrow=  new Exception("fake exception");

            FakeEmailService mockEmail = new FakeEmailService();

            LogAnalyzer2 log = new LogAnalyzer2(stubService,mockEmail);
```

```
        string tooShortFileName="abc.ext";
        log.Analyze(tooShortFileName);

        StringAssert.Contains("someone@somewhere.com",mockEmail.To);
        StringAssert.Contains("fake exception",mockEmail.Body);
        StringAssert.Contains("can't log",mockEmail.Subject);
    }
}

public class FakeWebService:IWebService
{
    public Exception ToThrow;
    public void LogError(string message)
    {
        if(ToThrow!=null)
        {
            throw ToThrow;
        }
    }
}

public class FakeEmailService:IEmailService
{
    public string To;
    public string Subject;
    public string Body;

    public void SendEmail(string to,
           string subject,
           string body)
    {
        To = to;
        Subject = subject;
        Body = body;
    }
}
```

This code raises some interesting points:

- Having several asserts can sometimes be a problem, because the first time an assert fails in your test, it actually throws a special type of exception that is caught by the test runner. That also means no other lines below the line that just failed will be executed. In this current case, it's OK, because if one assert fails, you don't care about the others because they're all related to the same object, and they're part of the same "feature."
- If you cared about the other asserts being run even if the first one failed, it would be a good indication to you to break this test into multiple tests. Alternatively, perhaps you could just create a new `EmailInfo` object and have the three attributes put on it, and then in your test create an expected version of this object with all correct properties. This would then be one assert.

Here's how it would look:

```
class EmailInfo
{
    public string Body;
    public string To;
    public string Subject;
}

[Test]
public void Analyze_WebServiceThrows_SendsEmail()
{
    FakeWebService stubService = new FakeWebService();
    stubService.ToThrow=  new Exception("fake exception");

    FakeEmailService mockEmail = new FakeEmailService();

    LogAnalyzer2 log = new LogAnalyzer2(stubService,mockEmail);

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

    EmailInfo expectedEmail = new EmailInfo {
                                        Body = "fake exception",
                                        To = "someone@somewhere.com",
                                        Subject = "can't log" }

    Assert.AreEqual(expectedEmail, mockEmail.email);

}

public class FakeEmailService:IEmailService
{
    public EmailInfo email = null;
    public void SendEmail(EmailInfo emailInfo)
    {
        email = emailInfo;
    }
}
```

Creating an expected object

Asserting on all properties together with an expected object

If you have a reference to the *actual* object you expect as an end result, you might be able to use this:

```
Assert.AreSame(expectedEmail, mockEmail.email);
```

One important thing to consider is how many mocks and stubs you can use in a test.

4.5 One mock per test

In a test where you test only one thing (which is how I recommend you write tests), there should be no more than one mock object. All other fake objects will act as stubs. Having more than one mock per test usually means you're testing more than one thing, and this can lead to complicated or brittle tests. (Look for more on this in chapter 8.)

If you follow this guideline, when you get to more complicated tests, you can always ask yourself, "Which one is my mock object?" Once you've identified it, you can leave the others as stubs and not have any assertions against them. (If you do find assertions against fakes that are clearly used as stubs, be wary. It's the mark of overspecification.)

Overspecification is the act of specifying too many things that should happen that your test shouldn't care about; for example, that stubs were called.

These extra specifications can make your test fail for all the wrong reasons: You change your production code to work differently, but even though the end result of your code is still good, your test will start screaming at you, "I've failed! You told me this method will be called and it wasn't! Waaah!"

Then you'll have to constantly change your tests to suit the internal implementation of your code—and then you'll eventually get tired of doing that and ask yourself "Why am I doing this again?" and then you'll start to delete those tests.

Game over.

Specify only one of the three end results of a unit of work, or you'll end up in hell, many kittens will die, and angels will fall from the sky. I warned you.

Next, we'll deal with a more complex scenario: using a stub to return a fake (mock or stub) that will be used by the application.

4.6 Fake chains: stubs that produce mocks or other stubs

One of the most common scamming techniques online these days follows a simple path. A fake email is sent to a massive number of recipients. The fake email is from a fake bank or online service claiming that the potential customer needs to have a balance checked or to change some account details on the online site.

All the links in the email point to a fake site. It looks exactly like the real thing, but its only purpose is to collect data from innocent customers of that business. In essence, you have a fake email that brings you to a fake website. This simple chain of lies is known as a phishing attack, and it's more lucrative than you'd imagine.

Why does this chain of lies matter to you? Sometimes you want to have a fake object return (from a method or property) another fake component, producing your own little chain of stubs ending with a mock object deep in the bowels of the system, so that you can end up collecting some data during your test. You can create a stub that leads to a mock object that records data.

The design of many systems under test allows for complex object chains to be created. It's not uncommon to find code like this:

```
IServiceFactory factory = GetServiceFactory();
IService service = factory.GetService();
```

Or like this:

```
String connstring =
GlobalUtil.Configuration.DBConfiguration.ConnectionString;
```

Suppose you wanted to replace the connection string with one of your own during a test. You could set up the `Configuration` property of the `GlobalUtil` object to be a stub object. Then, you could set the `DBConfiguration` property on that object to be another stub object, and so on, finally returning a fake object that you'll use as a mock, or a stub of the connection string.

It's a powerful technique, but you need to ask yourself whether it might not be better to refactor your code to do something like this:

```
String connstring =GetConnectionString();
Protected virtual string GetConnectionString()
{
    Return GlobalUtil.Configuration.DBConfiguration.ConnectionString;
}
```

You could then override the virtual method as described in chapter 3 (section 3.4.5). This can make the code easier to read and maintain, and it doesn't require adding new interfaces to insert two more stubs into the system.

> **TIP** Another good way to avoid call chains is to create special wrapper classes around the API that simplify using and testing it. For more about this method, see *Working Effectively with Legacy Code* by Michael Feathers. The pattern is called Adapt Parameter in that book.

Handwritten mocks and stubs have benefits, but they also have their share of problems. Let's take a look at them.

4.7 The problems with handwritten mocks and stubs

There are several issues that crop up when using manual mocks and stubs:

- It takes time to write the mocks and stubs.
- It's difficult to write stubs and mocks for classes and interfaces that have many methods, properties, and events.
- To save state for multiple calls of a mock method, you need to write a lot of boilerplate code within the handwritten fakes.
- If you want to verify that all parameters on a method call were sent correctly by the caller, you'll need to write multiple asserts. That's a drag.
- It's hard to reuse mock and stub code for other tests. The basic stuff works, but once you get into more than two or three methods on the interface, everything starts getting tedious to maintain.
- Is there a place for a fake that is both a mock and a stub? Very rarely. And I mean maybe once or twice in a project. I've only seen this a couple of times in the past couple of years myself.

A mock might double as a stub when you need to use a mock object that has a function that returns a value. To satisfy the compiler (oh, static languages, you mock us right back, don't you?—pun intended), you'll also need to return some fake value from the fake object, or the code won't run (or even compile). In that case, your

mock is doubling as a stub, and I'd argue that if you're returning a value back into the system from your mock object, you might be testing the wrong end result to begin with. I usually look for end results that are calls to a third-party system that don't return anything. I look for void methods as much as possible. Sometimes the design of the system requires that the method that calls the third party also is a function (happens a lot in C++ to denote errors). That's the one case where I'd allow something to be both a mock and a stub. Here's an example:

```
public interface IComNotificationService
{
  int SendNotification(string info);
}
```

In this code, if the unit of work's end result is to call `SendNotification`, you'd use a mock to prove that method got called, but to satisfy the compiler you'd also have to tell it to return some value you don't care about for this test.

These problems are inherent in manually written mocks and stubs. Fortunately, there are other ways to create mocks and stubs, as you'll see in the next chapter.

4.8 Summary

This chapter covered the distinction between stubs and mock objects. A mock object is like a stub, but it also helps you to assert something in your test. A stub can never fail your test and is strictly there to simulate various situations. This distinction is important because many of the mock object frameworks you'll see in the next chapter have these definitions engrained in them, and you'll need to know when to use which.

Combining stubs and mocks in the same test is a powerful technique, but you must take care to have no more than one mock in each test. The rest of the fake objects should be stubs that can't break your test. Following this practice can lead to more maintainable tests that break less often when internal code changes.

Stubs that produce other stubs or mocks can be a powerful way to inject fake dependencies into code that uses other objects to get its data. It's a great technique to use with factory classes and methods. You can even have stubs that return other stubs that return other stubs and so on, but at some point you'll wonder if it's all worth it. In that case, take a look at the techniques described in chapter 3 for injecting stubs into your design. (The next chapter discusses how some isolation frameworks allow you to create full fake call chains in one line of code—recursive fakes to the rescue!)

One of the most common problems encountered by people who write tests is using mocks too much in their tests (overspecification). You should rarely verify calls to fake objects that are used both as mocks and as stubs in the same test. You can have multiple stubs in a test, because a class may have multiple dependencies. Just make sure your test remains readable. Structure your code nicely so the reader of the test understands what's going on.

You may find that writing manual mocks and stubs is inconvenient for large interfaces or for complicated interaction-testing scenarios. It is, and there are better ways

to do this, as you'll see in the next chapter. But often you'll find that handwritten mocks and stubs still beat frameworks for simplicity and readability. The art lies in when you use which tool.

The next chapter deals with isolation (mocking) frameworks, which allow you to automatically create, at runtime, stubs or mock objects and use them with at least the same power as manual mocks and stubs, if not much, much more.