6 Digging deeper into isolation frameworks

This chapter covers

- Working with constrained versus unconstrained frameworks
- Understanding how unconstrained profiler-based frameworks work
- Defining the values of a good isolation framework

In the previous chapter, we used NSubstitute to create fakes. In this chapter, we'll step back to look at the bigger picture of isolation frameworks both in .NET and outside it. The world of isolation frameworks is vast, and there are many different things to consider when you choose one.

Let's start with a simple question: Why do some frameworks have more abilities than others? For example, some frameworks are able to fake static methods, and some aren't. Some are even able to fake objects that haven't yet been created, and others are blissfully unaware of such abilities. What gives?

6.1 Constrained and unconstrained frameworks

Isolation frameworks in .NET (and in Java, C++, and other static languages) fall into two basic groups based on their abilities to do specific things in the programming language. I call these two archetypes *unconstrained* and *constrained*.

6.1.1 Constrained frameworks

Constrained frameworks in .NET include Rhino Mocks, Moq, NMock, EasyMock, NSubstitute, and FakeItEasy. In Java, jMock and EasyMock are examples of constrained frameworks.

I call them *constrained* because there are some things these frameworks aren't able to fake. What they can or can't fake changes depending on the platform they run on and how they use that platform.

In .NET, constrained frameworks are unable to fake static methods, nonvirtual methods, nonpublic methods, and more.

What's the reason for that? Constrained isolation frameworks work in the same way that you use handwritten fakes: they generate code and compile it at runtime, so they're constrained by the compiler and intermediate language (IL) abilities. In Java, the compiler and the resulting bytecode are the equivalent. In C++, constrained frameworks are constrained by the C++ language and its abilities.

Constrained frameworks usually work by generating code at runtime that inherits and overrides interfaces or base classes, just as you did in the previous chapter, only you did it before running the code. That means that these isolation frameworks also have the same requirements for compiling: the code you want to fake has to be public and inheritable (nonsealed), has to have a public constructor, or should be an interface. For base classes, methods you'd like to override need to be virtual.

All this means that if you're using constrained frameworks, you're basically bound by the same compiler rules as regular code. Static methods, private methods, sealed classes, classes with private constructors, and so on are out of the equation when using such a framework.

6.1.2 Unconstrained frameworks

Unconstrained frameworks in .NET include Typemock Isolator, JustMock, and Moles (a.k.a. MS Fakes). In Java, PowerMock and JMockit are examples of unconstrained frameworks. In C++, Isolator++ and Hippo Mocks are examples of such frameworks. Unconstrained frameworks don't generate and compile code at runtime that inherits from other code. They usually use other means to get what they need, and the way they achieve what they need changes based on the platform.

Before we jump into how these work in .NET, I should mention that this chapter goes a bit deep. It's not really about the art of unit testing, but it allows you to understand why some things are the way they are and to make better-informed decisions about your unit test's design, and to take action based on that knowledge.

In .NET, all unconstrained frameworks are profiler-based. That means they use a set of unmanaged APIs called the *profiling APIs* that are wrapped around the running instance of the CLR—the Common Language Runtime—in .NET. You can read more about them at http://msdn.microsoft.com/en-us/library/bb384493.aspx#profiling_api. These APIs provide events on anything that happens during CLR code execution, and even on events that happen before .NET IL code gets compiled in memory into binary code. Some of these events also allow you to change and inject new IL-based code to be compiled in memory, thus adding new functionality to existing code. A lot of tooling out there, from the ANTS profiler to memory profilers, already uses the profiling APIs. Typemock Isolator was the first framework, more than seven years ago, to understand the potential of profiler APIs and their use to change the behavior of "fake" objects.

Because profiling events happen on all the code, including static methods, private constructors, and even third-party code that doesn't belong to you, like SharePoint, these unconstrained frameworks in .NET can effectively inject and change the behavior of any code they wish, in any class, in any library, even if it wasn't compiled by you. The options are limitless. I'll discuss in detail the differences between the profiler-based frameworks in the appendix.

In .NET, to enable profiling, and for your code to run tests written using a framework that uses the profiling APIs, you need to activate the environment variables for the executable process that runs the tests. By default, they're not active, so .NET code isn't profiled unless you opt in. Set `Cor_Enable_Profiling=0x1` and `COR_PROFILER=SOME_GUID` for the profiler you want hooked up to the process running the tests. (Yes, there can only be one profiler attached at a time.)

Frameworks such as Moles, Typemock, and JustMock all have special add-ins to Visual Studio that enable these environment variables and allow your tests to run. These tools usually have a special command-line executable that runs your other command-line tasks with these two environment variables enabled.

If you try to run your tests without a profiler enabled, you might see weird errors in the output window of the test runner. Be warned. The isolation framework you use might report that nothing was recorded or that no tests were run, for example.

Using unconstrained isolation frameworks has some advantages:

- You can write unit tests for previously untestable code, because you can fake things around the unit of work and isolate it, without needing to touch and refactor the code. Later, when you have tests, you can start refactoring.
- You can fake third-party systems that you can't control and that are potentially very hard to test with, such as if your objects have to inherit from the base class of a third-party product that contains many dependencies at a lower level (SharePoint, CRM, Entity Framework, or Silverlight, to name a few).
- You can choose your own level of design, rather than be forced into specific patterns. Design isn't created by a tool; it's a people issue. If you don't know what you're doing, a tool won't help you anyway. I talk more about this in chapter 11.

Using unconstrained isolation frameworks also has some cons:

- If you don't pay close attention, you can fake your way into a corner by faking things that aren't needed, instead of looking at the unit of work at a higher level.
- If you don't pay close attention, some tests can become unmaintainable because you're faking APIs that you don't own. This can happen, but not as often as you might think. From my experience, if you fake a low-enough level of an API in a framework, it's very unlikely to change in the future. The deeper an API is, the more likely many things are built on top of it, and the less likely it is to change.

Next, we'll look at what allows unconstrained frameworks to do these amazing feats.

6.1.3 How profiler-based unconstrained frameworks work

This section applies only to the .NET platform and the CLR, because that's where the profiling APIs live, and it should only matter to readers who care about exact and minute details. It isn't important to know this to write good unit tests, but it's good for extra bonus points if you ever want to build a competitor to these frameworks. Different techniques are employed in Java—or C++ for that matter.

In .NET, tools like Typemock Isolator will write native code in C++ that will attach to the CLR Profiler API's COM interface and register to a handful of special event-hook callbacks. Typemock actually owns a patent on this (you can find it at http://bit.ly/typemockpatent), which they don't seem to enforce, or we wouldn't have had competitors like JustMock and Moles entering the ring.

`JitCompilationStarted`, in conjunction with `SetILFunctionBody`, both members of the `ICorProfilerCallback2` COM interface, allow you to get and change, at runtime, the IL code that's about to be executed *before* it gets turned into binary code. You can change this IL code so that it includes custom IL code of your own. Tools like Typemock will insert IL headers before and after each method that they can get their hands on. These headers are basically logic code that calls out to managed C# code and checks to see if someone has set a special behavior on this method. Think of this process as generating global, aspect-oriented, crosscutting checks on all methods in your code about how to behave. The injected IL headers will also have calls to managed code hooks (written in C#, usually, where the real heart of the isolation framework logic lies) based on what behavior was set by the user of the framework API (such as "throw an exception" or "return a fake value").

Just-in-time (JIT) compilation happens in .NET for everything (unless it was pre-JITted using NGen.exe). This includes all code, not just your own, and even the .NET framework itself, SharePoint, or other libraries.

That means that a framework such as Typemock can inject IL behavior code into any code it likes, even if it's part of the .NET framework. You can add these headers before and after each method, even if you didn't write the code for them, and that's why these frameworks can be a godsend for legacy code that you don't have the power to refactor.

NOTE The profiling APIs aren't very well documented (on purpose?). But if you Google `JitCompilationStarted` and `SetILFunctionBody`, you should find many references and anecdotes to guide you on your quest to build your own unconstrained isolation framework in .NET. Prepare for a long arduous journey, and learn C++. Take along a bottle of whiskey.

FRAMEWORKS EXPOSE DIFFERENT PROFILER ABILITIES

Potentially, *all* profiler-based isolation frameworks have the same underlying abilities. But in real life, the major frameworks in .NET aren't the same in their abilities. Each of the big three profiler-based frameworks—JustMock, Typemock, and MS Fakes (Moles)—implements some subset of the full abilities.

NOTE I'm using the names Typemock and Typemock Isolator interchangeably, because that's the current way of referring to the Isolator product.

Typemock, having been around the longest, supports almost any code that would today seem untestable when doing tests with legacy code, including future objects, static constructors, and other weird creatures. It lacks in only the area of faking APIs from mscorlib.dll; that's the library that contains essential APIs like DateTime, System.String, and System.IO namespaces. In that specific DLL (and only that one), Typemock chose to implement only a handful of APIs instead of all of them.

Technically, Typemock could have chosen to allow faking of types from this whole library, but performance issues made that unrealistic. Imagine faking all strings in your system to return some fake values. Multiply the number of times each string is used in the underlying basic API of the .NET framework with a check or two for each call inside the Typemock API to check whether or not to fake this action, and you have yourself a performance nightmare.

Other than some core types of the .NET framework, Typemock supports just about anything you can throw at it.

MS Fakes has an advantage over Typemock Isolator. It was written and developed inside Microsoft, initially as an addition to another tool called Pex (described in the appendix). Because it was developed in-house, Microsoft developers had more insight into the largely undocumented profiling APIs, so they've built in support for some types that even Typemock Isolator doesn't allow faking. On the other hand, the API for MS Fakes doesn't contain most of the legacy code-related functionality found in Isolator or JustMock that you might expect from a framework with such abilities. The API mainly allows you to replace public methods (static and nonstatic) with delegates of your own, but it doesn't permit nonpublic method faking out of the box with the API.

In terms of its API and what it can fake, JustMock is getting quite close to the abilities of Typemock Isolator, but it still lacks some things relating to legacy code, such as faking static constructors and private methods. Mostly this is because of how long it's been alive. MS Fakes and JustMock are now maybe three years old. Typemock has a three- or four-year head start on them.

For now, what's important to realize is that when you choose an isolation framework to use, you're also selecting a basic set of abilities or constraints.

NOTE Profiler-based frameworks do carry some performance penalty. They add calls to your code at each step of the way, so it runs more slowly. You might only start to notice it after you've added a few hundred tests, but it's noticeable and it's there. I've found that for the big plus they offer in being able to fake and test legacy code, that's a small penalty to pay.

6.2 Values of good isolation frameworks

In .NET (and somewhat in Java), a new generation of isolation frameworks has started to rise in the past couple of years. These isolation frameworks shed some of the weight that the older, more established frameworks had been carrying and made huge strides in the areas of readability, usability, and simplicity. Most importantly, they support test robustness over time, with features I'll list shortly.

These new isolation frameworks include Typemock Isolator (although it's getting a bit long in the tooth), NSubstitute, and FakeItEasy. The first is an unconstrained framework, and the other two are constrained frameworks, yet they still bring interesting things to the table regardless of their underlying constraints.

Unfortunately, in languages such as Ruby, Python, JavaScript, and others, isolation frameworks still don't support most of these values of readability and usability. It might be the lack of maturity of the frameworks themselves, but it could be that the unit-testing culture in those languages hasn't yet arrived at the same conclusions the .NET unit-testing geeks have come to. Then again, we could all be doing it wrong, and the way things are isolated in Ruby is the way to go. Anyway, where was I?

Good isolation frameworks have what I call *the big two values*:

- Future-proofing
- Usability

Here are some features that support these values in the newer frameworks:

- Recursive fakes
- Ignored arguments by default
- Wide faking
- Nonstrict behavior of fakes
- Nonstrict mocks

6.3 Features supporting future-proofing and usability

A future-proof test will fail only for the right reasons in the face of big changes to the production code in the future. Usability is the quality that allows you to easily understand and use the framework. Isolation frameworks can be very easy to use *badly* and cause very fragile and less-future-proof tests.

These are some features that promote test robustness:

- Recursive fakes
- Defaulting to ignored arguments on behaviors and verifications
- Nonstrict verifications and behavior
- Wide-area faking

6.3.1 Recursive fakes

Recursive faking is a special behavior of fake objects in the case where functions return other objects. Those objects will always be fake, automatically. Any objects returned by functions in those automatically faked objects will be fake as well, recursively.

Here's an example:

```
public interface IPerson
{
    IPerson GetManager();
}

[Test]
public void RecursiveFakes_work()
{
    IPerson p = Substitute.For<IPerson>();

    Assert.IsNotNull(p.GetManager());
    Assert.IsNotNull(p.GetManager().GetManager());
    Assert.IsNotNull(p.GetManager().GetManager().GetManager());
}
```

Notice how you don't need to do anything except write a single line of code to get this working. But why is this ability important? The less you have to tell the test setup about each specific API needing to be fake, the less coupled your test is to the actual implementation of production code, and the less you need to change the test if production code changes in the future.

Not all isolation frameworks allow recursive fakes, so check for this ability on your favorite framework. As far as I know, only .NET frameworks currently even consider this ability. I wish this existed in other languages as well.

Also note that constrained frameworks in .NET can only support recursive fakes on those functions that can be overridden by generated code: public methods that are virtual or part of an interface.

Some people are afraid that such a feature will more easily allow for the breaking of the law of Demeter (http://en.wikipedia.org/wiki/Law_of_Demeter). I disagree, because good design isn't enforced by a tool but is created by people talking to and teaching each other and by doing code reviews as pairs. But you'll see more on the topic of design in chapter 11.

6.3.2 Ignored arguments by default

Currently, in all frameworks except Typemock Isolator, any argument values you send into behavior-changing APIs or verification APIs are used as the default expected values.

Isolator, by default, ignores values you send in, unless you specifically say in the API calls that you care about the argument values. There's no need to always include `Arg.IsAny<Type>` in all methods, which saves typing and avoids generics that hinder readability. With Typemock Isolator (typemock.com), to throw an exception whatever the arguments are, you can just write this:

```
Isolate.WhenCalled(() => stubLogger.Write(""))
                         .WillThrow(new Exception("Fake"));
```

6.3.3 Wide faking

Wide faking is the ability to fake multiple methods at once. In a way, recursive fakes are a subfeature of that idea, but there are also other implementations.

With tools like FakeItEasy, for example, you can signify that all methods of a certain object will return the same value, or just the methods that return a specific type:

```
A.CallTo(foo).Throws(new Exception());
A.CallTo(foo).WithReturnType<string>().Returns("hello world");
```

With Typemock, you can signify that all static methods of a type will return a fake value by default:

```
Isolate.Fake.StaticMethods(typeof(HttpRuntime));
```

From this moment on, each static method on that object returns a fake value based on its type or a recursively fake object if it returns an object.

Again, I think this is great for the future sustainability of the tests as the production code evolves. A method that's added and used by production code six months from now will be automatically faked by all existing tests, so that those tests don't care about the new method.

6.3.4 Nonstrict behavior of fakes

The world of isolation frameworks used to be a very strict one and mostly still is. Many of the frameworks in languages other than .NET (such as Java and Ruby) are by default strict, whereas many of the .NET frameworks had grown out of that stage.

A strict fake's methods can only be invoked successfully if you set them as "expected" by the isolation API. This ability to expect that a method on a fake object will be called doesn't exist in NSubstitute (or FakeItEasy), but it does exist in many of the other frameworks in .NET and other languages (see Moq, Rhino Mocks, and the Typemock Isolator's old API).

If a method were configured to be expected, then any call that differs from the expectation (for example, I expect method `LogError` to be called with a parameter of `a` at the beginning of the test), either by the parameter values defined or by the method name, will usually be handled by throwing an exception.

The test will usually fail on the first unexpected method call to a strict mock object. I say *usually* because whether the mock throws an exception depends on the implementation of the isolation framework. Some frameworks allow you to define whether to delay all exceptions until calling `verify()` at the end of the test.

The main reasons many frameworks were designed this way can be found in the book *Growing Object-Oriented Software, Guided by Tests* by Freeman and Pryce (Addison-Wesley Professional, 2009). In that book, they use the mock assertions to describe the "protocol" of communication between objects. Because a protocol is something that needs to be quite strict, reading the test should help you understand the way an object expects to be interacted with.

So what's problematic about this? The idea itself is not a problem; it's the ease with which one can abuse and overuse this ability that's the problem.

A strict mock can fail in two ways: when an unexpected method is called on it, or when expected methods aren't called on it (which is determined by calling `Received()`).

It's the former that bothers me. Assuming I don't care about internal protocols between objects that are internal to my unit of work, I shouldn't assert on their interactions, or I would be in a world of hurt. A test can fail if I decide to call a method on some internal object in the unit of work that's unrelated to the end result of that unit of work. Nevertheless, my test will fail, whining, "You didn't tell me someone will call *that* method!"

6.3.5 Nonstrict mocks

Most of the time, nonstrict mocks make for less-brittle tests. A nonstrict mock object will allow any call to be made to it, even if it wasn't expected. For methods that return values, it will return the default value if it's a value object or null for an object. In more advanced frameworks, there's also the notion of recursive fakes, in which a fake object that has a method that returns an object will return a fake object by default from that method. And that fake object will also return fake objects from its methods that return objects, recursively. (This exists in Typemock Isolator, as well as NSub, Moq, and partially in Rhino Mocks.)

Listing 5.3 in chapter 5 is a pure example of nonstrict mocks. You don't care what other calls happened. Listing 5.4 and the code block after it show how you can make the test more robust and future-proof by using an argument matcher instead of expecting a full string. Argument matching allows you to create rules on how parameters should be passed for the fake to consider them OK. Notice how it uglies up the test quite easily.

6.4 Isolation framework design antipatterns

Here are some of the antipatterns found in frameworks today that we can easily alleviate:

- Concept confusion
- Record and replay
- Sticky behavior
- Complex syntax

In this section, we'll take a look at each of them.

6.4.1 Concept confusion

Concept confusion is something I like to refer to as *mock overdose*. I'd prefer a framework that doesn't use the word *mock* for everything.

You have to know how many mocks and stubs there are in a test, because more than a single mock in a test is usually a problem. When it doesn't distinguish between the two, the framework could tell you that something is a mock when in fact it's used as a stub. It takes you longer to understand whether this is a real problem or not, so the test readability is hurt.

Here's an example from Moq:

```
[Test]
public void ctor_WhenViewhasError_CallsLogger()
{
    var view = new Mock<IView>();
    var logger = new Mock<ILogger>();

    Presenter p = new Presenter(view.Object, logger.Object);
    view.Raise(v => v.ErrorOccured += null, "fake error");

    logger.Verify(log =>
        log.LogError(It.Is<string>(s=> s.Contains("fake error"))));
}
```

Here's how you can avoid concept confusion:

- Have specific words for *mock* and *stub* in the API. Rhino Mocks does this, for example.
- Don't use the terms *mock* and *stub* at all in the API. Instead, use a generic term for whatever a fake something is. In FakeItEasy, for example, everything is a *Fake*<Something>. There is no mock or stub at all in the API. In NSubstitute, as you might remember, everything is a *Substitute*<Something>. In Typemock Isolator, you'd only call Isolate.Fake.Instance<Something>. There is no mention of mock or stub.
- If you're using an isolation framework that doesn't distinguish mocks and stubs, at the very least name your variables mockXXX and stubXXX to mitigate some of the readability problems.

By removing the overloaded term altogether, or by allowing the user to specify what they're creating, readability of the tests can increase, or at least the terminology will be less confusing.

Here's the previous test with the names of the variables changed to denote how they're used. Does it read better to you?

```
[Test]
public void ctor_WhenViewhasError_CallsLogger()
{
    var stubView = new Mock<IView>();
    var mockLogger = new Mock<ILogger>();

    Presenter p= new Presenter(stubView.Object, mockLogger.Object);
    stubView.Raise(view=> view.ErrorOccured += null, "fake error");
```

```
    mockLogger.Verify(logger =>
      logger.LogError(It.Is<string>(s=>s.Contains("fake error"))));
}
```

6.4.2 Record and replay

Record-and-replay style for isolation frameworks created bad readability. A sure sign of bad readability is when the reader of a test has to look up and down the same test many times in order to understand what's going on. You can usually see this in code written with an isolation framework that supports record-and-replay APIs.

Take a look at this example of using Rhino Mocks (which supports record and replay) from Rasmus Kromann-Larsen's blog, http://rasmuskl.dk/post/Why-AAA-style-mocking-is-better-than-Record-Playback.aspx. (Don't try to compile it. It's just an example.)

```
[Test]
public void ShouldIgnoreRespondentsThatDoesNotExistRecordPlayback()
{
    // Arrange
    var guid = Guid.NewGuid();
    // Part of Act
    IEventRaiser executeRaiser;

    using(_mocks.Record())
    {
        // Arrange (or Assert?)
        Expect.Call(_view.Respondents).Return(new[] {guid.ToString()});
        Expect.Call(_repository.GetById(guid)).Return(null);

        // Part of Act
        _view.ExecuteOperation += null;
        executeRaiser = LastCall.IgnoreArguments()
            .Repeat.Any()
            .GetEventRaiser();

        // Assert
        Expect.Call(_view.OperationErrors = null)
            .IgnoreArguments()
            .Constraints(List.IsIn("Non-existant respondent: " + guid));
    }

    using(_mocks.Playback())
    {
        // Arrange
        new BulkRespondentPresenter(_view, _repository);
        // Act
        executeRaiser.Raise(null, EventArgs.Empty);
    }
}
```

And here's the same code with Moq (which supports arrange-act-assert (AAA)–style testing:

```
[Test]
public void ShouldIgnoreRespondentsThatDoesNotExist()
{
    // Arrange
```

```
    var guid = Guid.NewGuid();
    _viewMock.Setup(x => x.Respondents).Returns(new[] { guid.ToString() });
    _repositoryMock.Setup(x => x.GetById(guid)).Returns(() => null);

    // Act
    _viewMock.Raise(x => x.ExecuteOperation += null, EventArgs.Empty);

    // Assert
    _viewMock.VerifySet(x => x.OperationErrors =
    It.Is<IList<string>>(l=>l.Contains("Non-existant respondent: "+guid)));
}
```

See what a huge difference using AAA style makes over record and replay?

6.4.3 Sticky behavior

Once you tell a fake method to behave in a certain way when called, what happens the next time it gets called in production? Or the next 100 times? Should your test care? If the fake behavior of methods is designed to happen only once, your test will have to provide a "what do I do now" answer every time the production code changes to call the fake method, even if your test doesn't care about those extra calls. It's now coupled more into internal implementation calls.

To solve this, the isolation framework can add default "stickiness" to behaviors. Once you tell a method to behave in a certain way (say, return `false`), it will behave that way *always* until told to behave differently (all future calls will return `false`, even if you call it 100 times). This absolves the test from knowing how the method should behave later on, when it's no longer important for the purpose of the current test.

6.4.4 Complex syntax

With some frameworks, it's hard to remember how to do standard operations, even after you've used them for a while. This adds friction to the coding experience. You can design the API in a way that makes this easier. For example, in FakeItEasy, all possible operations *always* start with a capital `A`. Here's an example from FakeItEasy's wiki, https://github.com/FakeItEasy/FakeItEasy/wiki:

```
var lollipop = A.Fake<ICandy>();
var shop = A.Fake<ICandyShop>();

// To set up a call to return a value is also simple:
A.CallTo(() => shop.GetTopSellingCandy()).Returns(lollipop);

     A.CallTo(() => foo.Bar(A<string>.Ignored,
         "second argument")).Throws(new Exception());

// Use your fake as you would an actual instance of the faked type.
var developer = new SweetTooth();
developer.BuyTastiestCandy(shop);

// Asserting uses the exact same syntax as when configuring calls,
// no need to teach yourself another syntax.
A.CallTo(() => shop.BuyCandy(lollipop)).MustHaveHappened();
```

Creating a fake starts with A

Setting a method's behavior starts with A

Using an argument matcher starts with A

Verifying a method was called starts with A

The same concept exists in Typemock Isolator, where all API calls start with the word `Isolate`.

This single point of entry makes it easier to start with the right word and then use the built-in IDE features of Intellisense to figure out the next move.

With NSubstitute, you need to remember to use `Substitute` to create fakes, to use extension methods of real objects to verify or change behavior, and to use `Arg<T>` to use argument matchers.

6.5 Summary

Isolation frameworks are divided into two categories: constrained and unconstrained frameworks. Depending on the platform they run on, a framework can have more or fewer abilities, and it's important to understand what a framework can or can't do when you choose it.

In .NET, unconstrained frameworks use the profiling APIs, whereas most constrained frameworks generate and compile code at runtime, just as you do manually with handwritten mocks and stubs.

Isolation frameworks that support the values of future-proofing and usability can make your life in unit-test-land easier, whereas those that don't can make your life harder.

That's it! We've covered the core techniques for writing unit tests. The next part of the book deals with managing test code, arranging tests, and creating patterns for tests that you can rely on, maintain easily, and understand clearly.

Part 3

The test code

This part covers techniques for managing and organizing unit tests and for ensuring that the quality of unit tests in real-world projects is high.

Chapter 7 first covers the role of unit testing as part of an automated build process and follows up with several techniques for organizing different kinds of tests according to categories (speed, type) with a goal of reaching what I call the safe green zone. It also explains how to "grow" a test API or test infrastructure for your application.

In chapter 8, we'll look at the three basic pillars of good unit tests—readability, maintainability, and trustworthiness—and explore techniques to support them. If you read only one chapter in this book, chapter 8 should be it.