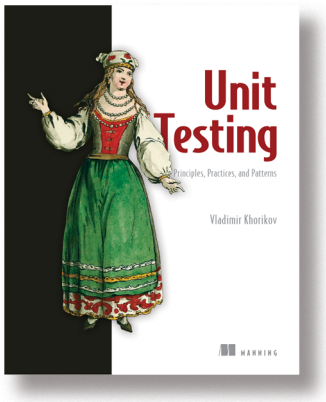
FREE

Exploring Mocks in Unit Testing

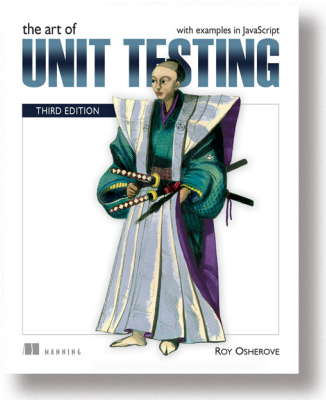
Chapters selected by Vladimir Khorikov



Unit Testing Principles, Practices, and Patterns

by Vladimir Khorikov

ISBN 9781617296277
304 pages
$39.99



The Art of Unit Testing, Third Edition

by Roy Osherove

ISBN 9781617297489
325 pages
Spring 2021
$39.99

Exploring Mocks in Unit Testing

Chapters chosen by Vladimir Khorikov

Manning Author Picks

For online information and ordering of these and other Manning books, please visit www.manning.com. The publisher offers discounts on these books when ordered in quantity.

For more information, please contact

Special Sales Department
Manning Publications Co.
20 Baldwin Road
PO Box 761
Shelter Island, NY 11964
Email: Candace Gillhoolley, corp-sales@manning.com

Manning Publications Co.
20 Baldwin Road Technical
PO Box 761
Shelter Island, NY 11964

Cover designer: Leslie Haimes

ISBN: 9781617299377

contents

introduction iv

Mocks and test fragility 2
Chapter 5 from *Unit Testing Principles, Practices, and Patterns*

Mocking best practices 30
Chapter 9 from *Unit Testing Principles, Practices, and Patterns*

Isolation (mocking) frameworks 44
Chapter 5 from *The Art of Unit Testing*

index 63

introduction

The use of mocks in unit testing has always been a somewhat controversial topic. A lot of people go through a steep learning curve before they master how and, more importantly, *when* to use mocks. A typical progression I see among programmers is to first mock almost every dependency, then transition to the "no-mocks" policy, and then settle on "only mock external dependencies". While mocking only external dependencies is generally good advice, it is incomplete and lacks nuance.

This sampler brings together chapters from two Manning books to enhance clarity of the practice of using mocks. I've selected two chapters from my book *Unit Testing Principles, Practices, and Patterns*. The first one, "Mocks and test fragility", has a thorough discussion of what a mock is and how it is different from stubs and other types of test doubles, and answers the question of what types of dependencies you should use them for. The second chapter, "Mocking best practices", expands on the remaining guidelines related to mocking that will help you maximize their value and minimize maintenance costs. Finally, I've chosen a chapter from *The Art of Unit Testing* by Roy Osherove for some practical demonstrations of mocking frameworks.

The proper use of mocks is one of the most important topics in unit testing. This collection of chapters will help you to understand mocking best practices with real-world illustrations and practical tips that will have you well on your way to writing better code, increasing your productivity, and improving the quality of your software. If you're interested in a deeper exploration of this important topic, the complete versions of the books featured here are great places to start!

Chapter 5 from *Unit Testing Principles, Practices, and Patterns* by Vladimir Khorikov

The goal of this chapter is to give an overview of what a mock is and how it is different from stubs and other test doubles. It also covers when you should use mocks and when to choose the real dependencies in tests.

5 Mocks and test fragility

This chapter covers

- Differentiating mocks from stubs
- Defining observable behavior and implementation details
- Understanding the relationship between mocks and test fragility
- Using mocks without compromising resistance to refactoring

Chapter 4 introduced a frame of reference that you can use to analyze specific tests and unit testing approaches. In this chapter, you'll see that frame of reference in action; we'll use it to dissect the topic of mocks.

The use of mocks in tests is a controversial subject. Some people argue that mocks are a great tool and apply them in most of their tests. Others claim that mocks lead to test fragility and try not to use them at all. As the saying goes, the truth lies somewhere in between. In this chapter, I'll show that, indeed, mocks often result in fragile tests—tests that lack the metric of *resistance to refactoring*. But there are still cases where mocking is applicable and even preferable.

This chapter draws heavily on the discussion about the London versus classical schools of unit testing from chapter 2. In short, the disagreement between the schools stems from their views on the test isolation issue. The London school advocates isolating pieces of code under test from each other and using test doubles for all but immutable dependencies to perform such isolation.

The classical school stands for isolating unit tests themselves so that they can be run in parallel. This school uses test doubles only for dependencies that are shared between tests.

There's a deep and almost inevitable connection between mocks and test fragility. In the next several sections, I will gradually lay down the foundation for you to see why that connection exists. You will also learn how to use mocks so that they don't compromise a test's *resistance to refactoring*.

5.1 Differentiating mocks from stubs

In chapter 2, I briefly mentioned that a *mock* is a test double that allows you to examine interactions between the system under test (SUT) and its collaborators. There's another type of test double: a *stub*. Let's take a closer look at what a mock is and how it is different from a stub.

5.1.1 The types of test doubles

A *test double* is an overarching term that describes all kinds of non-production-ready, fake dependencies in tests. The term comes from the notion of a stunt double in a movie. The major use of test doubles is to facilitate testing; they are passed to the system under test instead of real dependencies, which could be hard to set up or maintain.

According to Gerard Meszaros, there are five variations of test doubles: *dummy*, *stub*, *spy*, *mock*, and *fake*.[1] Such a variety can look intimidating, but in reality, they can all be grouped together into just two types: mocks and stubs (figure 5.1).

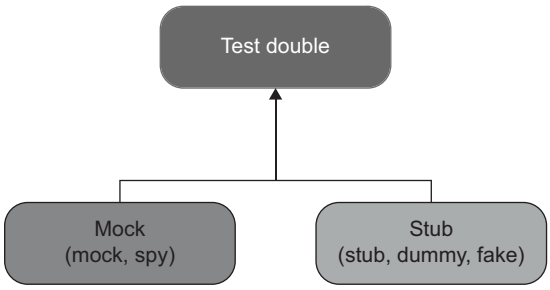


Figure 5.1 All variations of test doubles can be categorized into two types: mocks and stubs.

[1] See *xUnit Test Patterns: Refactoring Test Code* (Addison-Wesley, 2007).

The difference between these two types boils down to the following:

- Mocks help to emulate and examine *outcoming* interactions. These interactions are calls the SUT makes to its dependencies to change their state.
- Stubs help to emulate *incoming* interactions. These interactions are calls the SUT makes to its dependencies to get input data (figure 5.2).

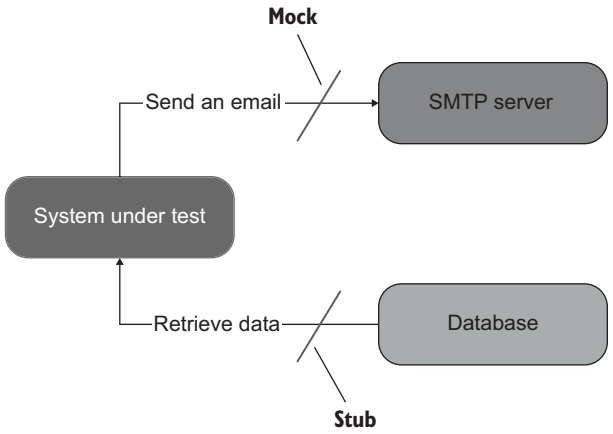


Figure 5.2 Sending an email is an *outcoming* interaction: an interaction that results in a side effect in the SMTP server. A test double emulating such an interaction is a *mock*. Retrieving data from the database is an *incoming* interaction; it doesn't result in a side effect. The corresponding test double is a *stub*.

All other differences between the five variations are insignificant implementation details. For example, *spies* serve the same role as mocks. The distinction is that spies are written manually, whereas mocks are created with the help of a mocking framework. Sometimes people refer to spies as *handwritten mocks*.

On the other hand, the difference between a stub, a dummy, and a fake is in how intelligent they are. A *dummy* is a simple, hardcoded value such as a null value or a made-up string. It's used to satisfy the SUT's method signature and doesn't participate in producing the final outcome. A *stub* is more sophisticated. It's a fully fledged dependency that you configure to return different values for different scenarios. Finally, a *fake* is the same as a stub for most purposes. The difference is in the rationale for its creation: a fake is usually implemented to replace a dependency that doesn't yet exist.

Notice the difference between mocks and stubs (aside from outcoming versus incoming interactions). Mocks help to *emulate and examine* interactions between the SUT and its dependencies, while stubs only help to *emulate* those interactions. This is an important distinction. You will see why shortly.

5.1.2 Mock (the tool) vs. mock (the test double)

The term *mock* is overloaded and can mean different things in different circumstances. I mentioned in chapter 2 that people often use this term to mean any test double, whereas mocks are only a subset of test doubles. But there's another meaning

for the term *mock*. You can refer to the classes from mocking libraries as mocks, too. These classes help you create actual mocks, but they themselves are not mocks per se. The following listing shows an example.

Listing 5.1 Using the `Mock` class from a mocking library to create a mock

```
[Fact]
public void Sending_a_greetings_email()
{
    var mock = new Mock<IEmailGateway>();
    var sut = new Controller(mock.Object);

    sut.GreetUser("user@email.com");

    mock.Verify(
        x => x.SendGreetingsEmail(
            "user@email.com"),
        Times.Once);
}
```

Uses a mock (the tool) to create a mock (the test double)

Examines the call from the SUT to the test double

The test in listing 5.1 uses the `Mock` class from the mocking library of my choice (Moq). This class is a tool that enables you to create a test double—a mock. In other words, the class `Mock` (or `Mock<IEmailGateway>`) is a *mock (the tool)*, while the instance of that class, `mock`, is a *mock (the test double)*. It's important not to conflate a mock (the tool) with a mock (the test double) because you can use a mock (the tool) to create both types of test doubles: mocks and stubs.

The test in the following listing also uses the `Mock` class, but the instance of that class is not a mock, it's a stub.

Listing 5.2 Using the `Mock` class to create a stub

```
[Fact]
public void Creating_a_report()
{
    var stub = new Mock<IDatabase>();
    stub.Setup(x => x.GetNumberOfUsers())
        .Returns(10);
    var sut = new Controller(stub.Object);

    Report report = sut.CreateReport();

    Assert.Equal(10, report.NumberOfUsers);
}
```

Uses a mock (the tool) to create a stub

Sets up a canned answer

This test double emulates an *incoming* interaction—a call that provides the SUT with input data. On the other hand, in the previous example (listing 5.1), the call to `SendGreetingsEmail()` is an *outcoming* interaction. Its sole purpose is to incur a side effect—send an email.

5.1.3 Don't assert interactions with stubs

As I mentioned in section 5.1.1, mocks help to *emulate and examine* outcoming interactions between the SUT and its dependencies, while stubs only help to *emulate* incoming interactions, not *examine* them. The difference between the two stems from the guideline of *never asserting interactions with stubs*. A call from the SUT to a stub is not part of the end result the SUT produces. Such a call is only a means to produce the end result: a stub provides input from which the SUT then generates the output.

> **NOTE** Asserting interactions with stubs is a common anti-pattern that leads to fragile tests.

As you might remember from chapter 4, the only way to avoid false positives and thus improve resistance to refactoring in tests is to make those tests verify the end result (which, ideally, should be meaningful to a non-programmer), not implementation details. In listing 5.1, the check

```
mock.Verify(x => x.SendGreetingsEmail("user@email.com"))
```

corresponds to an actual outcome, and that outcome is meaningful to a domain expert: sending a greetings email is something business people would want the system to do. At the same time, the call to `GetNumberOfUsers()` in listing 5.2 is not an outcome at all. It's an internal implementation detail regarding how the SUT gathers data necessary for the report creation. Therefore, asserting this call would lead to test fragility: it shouldn't matter how the SUT generates the end result, as long as that result is correct. The following listing shows an example of such a brittle test.

Listing 5.3 Asserting an interaction with a stub

```
[Fact]
public void Creating_a_report()
{
    var stub = new Mock<IDatabase>();
    stub.Setup(x => x.GetNumberOfUsers()).Returns(10);
    var sut = new Controller(stub.Object);

    Report report = sut.CreateReport();

    Assert.Equal(10, report.NumberOfUsers);
    stub.Verify(                                  // Asserts the interaction with the stub
        x => x.GetNumberOfUsers(),
        Times.Once);
}
```

This practice of verifying things that aren't part of the end result is also called *overspecification*. Most commonly, overspecification takes place when examining interactions. Checking for interactions with stubs is a flaw that's quite easy to spot because tests shouldn't check for *any* interactions with stubs. Mocks are a more complicated sub-

ject: not all uses of mocks lead to test fragility, but a lot of them do. You'll see why later in this chapter.

5.1.4 Using mocks and stubs together

Sometimes you need to create a test double that exhibits the properties of both a mock and a stub. For example, here's a test from chapter 2 that I used to illustrate the London style of unit testing.

Listing 5.4 `storeMock`: both a mock and a stub

```
[Fact]
public void Purchase_fails_when_not_enough_inventory()
{
    var storeMock = new Mock<IStore>();
    storeMock
        .Setup(x => x.HasEnoughInventory(
            Product.Shampoo, 5))
        .Returns(false);
    var sut = new Customer();

    bool success = sut.Purchase(
        storeMock.Object, Product.Shampoo, 5);

    Assert.False(success);
    storeMock.Verify(
        x => x.RemoveInventory(Product.Shampoo, 5),
        Times.Never);
}
```

Sets up a canned answer

Examines a call from the SUT

This test uses `storeMock` for two purposes: it returns a canned answer and verifies a method call made by the SUT. Notice, though, that these are two different methods: the test sets up the answer from `HasEnoughInventory()` but then verifies the call to `RemoveInventory()`. Thus, the rule of not asserting interactions with stubs is not violated here.

When a test double is both a mock and a stub, it's still called a mock, not a stub. That's mostly the case because we need to pick one name, but also because being a mock is a more important fact than being a stub.

5.1.5 How mocks and stubs relate to commands and queries

The notions of mocks and stubs tie to the command query separation (CQS) principle. The CQS principle states that every method should be either a command or a query, but not both. As shown in figure 5.3, *commands* are methods that produce side effects and don't return any value (return `void`). Examples of side effects include mutating an object's state, changing a file in the file system, and so on. *Queries* are the opposite of that—they are side-effect free and return a value.

To follow this principle, be sure that if a method produces a side effect, that method's return type is `void`. And if the method returns a value, it must stay side-effect

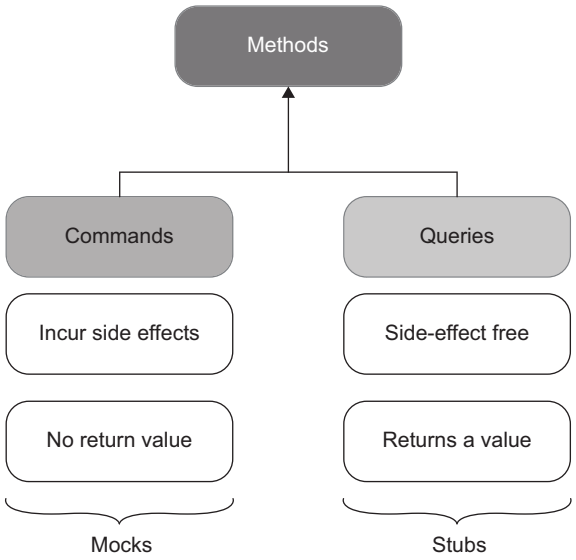


Figure 5.3 In the command query separation (CQS) principle, commands correspond to mocks, while queries are consistent with stubs.

free. In other words, asking a question should not change the answer. Code that maintains such a clear separation becomes easier to read. You can tell what a method does just by looking at its signature, without diving into its implementation details.

Of course, it's not always possible to follow the CQS principle. There are always methods for which it makes sense to both incur a side effect and return a value. A classical example is `stack.Pop()`. This method both removes a top element from the stack and returns it to the caller. Still, it's a good idea to adhere to the CQS principle whenever you can.

Test doubles that substitute commands become mocks. Similarly, test doubles that substitute queries are stubs. Look at the two tests from listings 5.1 and 5.2 again (I'm showing their relevant parts here):

```
var mock = new Mock<IEmailGateway>();
mock.Verify(x => x.SendGreetingsEmail("user@email.com"));

var stub = new Mock<IDatabase>();
stub.Setup(x => x.GetNumberOfUsers()).Returns(10);
```

`SendGreetingsEmail()` is a command whose side effect is sending an email. The test double that substitutes this command is a mock. On the other hand, `GetNumberOfUsers()` is a query that returns a value and doesn't mutate the database state. The corresponding test double is a stub.

5.2 Observable behavior vs. implementation details

Section 5.1 showed what a mock is. The next step on the way to explaining the connection between mocks and test fragility is diving into what causes such fragility.

As you might remember from chapter 4, *test fragility* corresponds to the second attribute of a good unit test: resistance to refactoring. (As a reminder, the four attributes are protection against regressions, resistance to refactoring, fast feedback, and maintainability.) The metric of resistance to refactoring is the most important because whether a unit test possesses this metric is mostly a binary choice. Thus, it's good to max out this metric to the extent that the test still remains in the realm of unit testing and doesn't transition to the category of end-to-end testing. The latter, despite being the best at resistance to refactoring, is generally much harder to maintain.

In chapter 4, you also saw that the main reason tests deliver false positives (and thus fail at resistance to refactoring) is because they couple to the code's implementation details. The only way to avoid such coupling is to verify the end result the code produces (its observable behavior) and distance tests from implementation details as much as possible. In other words, tests must focus on the *whats*, not the *hows*. So, what exactly is an implementation detail, and how is it different from an observable behavior?

5.2.1 Observable behavior is not the same as a public API

All production code can be categorized along two dimensions:

- Public API vs. private API (where API means *application programming interface*)
- Observable behavior vs. implementation details

The categories in these dimensions don't overlap. A method can't belong to both a public and a private API; it's either one or the other. Similarly, the code is either an internal implementation detail or part of the system's observable behavior, but not both.

Most programming languages provide a simple mechanism to differentiate between the code base's public and private APIs. For example, in C#, you can mark any member in a class with the `private` keyword, and that member will be hidden from the client code, becoming part of the class's private API. The same is true for classes: you can easily make them private by using the `private` or `internal` keyword.

The distinction between observable behavior and internal implementation details is more nuanced. For a piece of code to be part of the system's observable behavior, it has to do one of the following things:

- Expose an operation that helps the client achieve one of its goals. An *operation* is a method that performs a calculation or incurs a side effect or both.
- Expose a state that helps the client achieve one of its goals. *State* is the current condition of the system.

Any code that does neither of these two things is an *implementation detail*.

Notice that whether the code is observable behavior depends on who its client is and what the goals of that client are. In order to be a part of observable behavior, the

code needs to have an immediate connection to at least one such goal. The word *client* can refer to different things depending on where the code resides. The common examples are client code from the same code base, an external application, or the user interface.

Ideally, the system's public API surface should coincide with its observable behavior, and all its implementation details should be hidden from the eyes of the clients. Such a system has a *well-designed* API (figure 5.4).

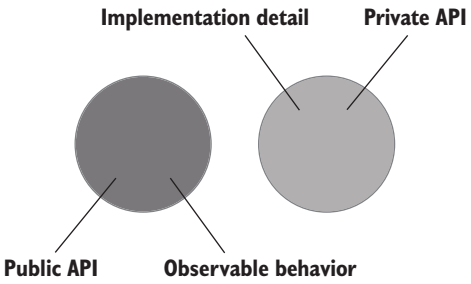


Figure 5.4 In a *well-designed* API, the observable behavior coincides with the public API, while all implementation details are hidden behind the private API.

Often, though, the system's public API extends beyond its observable behavior and starts exposing implementation details. Such a system's implementation details *leak* to its public API surface (figure 5.5).

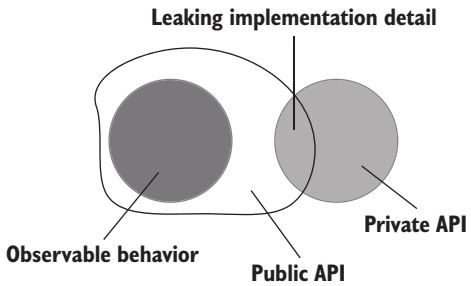


Figure 5.5 A system leaks implementation details when its public API extends beyond the observable behavior.

5.2.2 Leaking implementation details: An example with an operation

Let's take a look at examples of code whose implementation details leak to the public API. Listing 5.5 shows a `User` class with a public API that consists of two members: a `Name` property and a `NormalizeName()` method. The class also has an *invariant*: users' names must not exceed 50 characters and should be truncated otherwise.

Listing 5.5 `User` class with leaking implementation details

```
public class User
{
    public string Name { get; set; }
```

```
    public string NormalizeName(string name)
    {
        string result = (name ?? "").Trim();

        if (result.Length > 50)
            return result.Substring(0, 50);

        return result;
    }
}

public class UserController
{
    public void RenameUser(int userId, string newName)
    {
        User user = GetUserFromDatabase(userId);

        string normalizedName = user.NormalizeName(newName);
        user.Name = normalizedName;

        SaveUserToDatabase(user);
    }
}
```

`UserController` is client code. It uses the `User` class in its `RenameUser` method. The goal of this method, as you have probably guessed, is to change a user's name.

So, why isn't `User`'s API well-designed? Look at its members once again: the `Name` property and the `NormalizeName` method. Both of them are public. Therefore, in order for the class's API to be well-designed, these members should be part of the observable behavior. This, in turn, requires them to do one of the following two things (which I'm repeating here for convenience):

- Expose an *operation* that helps the client achieve one of its goals.
- Expose a *state* that helps the client achieve one of its goals.

Only the `Name` property meets this requirement. It exposes a setter, which is an operation that allows `UserController` to achieve its goal of changing a user's name. The `NormalizeName` method is also an operation, but it doesn't have an immediate connection to the client's goal. The only reason `UserController` calls this method is to satisfy the invariant of `User`. `NormalizeName` is therefore an implementation detail that leaks to the class's public API (figure 5.6).

To fix the situation and make the class's API well-designed, `User` needs to hide `NormalizeName()` and call it internally as part of the property's setter without relying on the client code to do so. Listing 5.6 shows this approach.

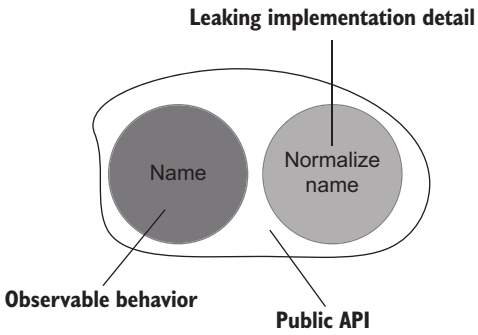


Figure 5.6 The API of `User` is not well-designed: it exposes the `NormalizeName` method, which is not part of the observable behavior.

Listing 5.6 A version of `User` with a well-designed API

```
public class User
{
    private string _name;
    public string Name
    {
        get => _name;
        set => _name = NormalizeName(value);
    }

    private string NormalizeName(string name)
    {
        string result = (name ?? "").Trim();

        if (result.Length > 50)
            return result.Substring(0, 50);

        return result;
    }
}

public class UserController
{
    public void RenameUser(int userId, string newName)
    {
        User user = GetUserFromDatabase(userId);
        user.Name = newName;
        SaveUserToDatabase(user);
    }
}
```

`User`'s API in listing 5.6 is well-designed: only the observable behavior (the `Name` property) is made public, while the implementation details (the `NormalizeName` method) are hidden behind the private API (figure 5.7).

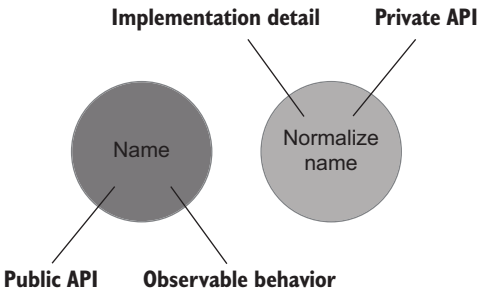


Figure 5.7 `User` **with a well-designed API. Only the observable behavior is public; the implementation details are now private.**

NOTE Strictly speaking, `Name`'s getter should also be made private, because it's not used by `UserController`. In reality, though, you almost always want to read back changes you make. Therefore, in a real project, there will certainly be another use case that requires seeing users' current names via `Name`'s getter.

There's a good rule of thumb that can help you determine whether a class leaks its implementation details. If the number of operations the client has to invoke on the class to achieve a single goal is greater than one, then that class is likely leaking implementation details. *Ideally, any individual goal should be achieved with a single operation.* In listing 5.5, for example, `UserController` has to use two operations from `User`:

```
string normalizedName = user.NormalizeName(newName);
user.Name = normalizedName;
```

After the refactoring, the number of operations has been reduced to one:

```
user.Name = newName;
```

In my experience, this rule of thumb holds true for the vast majority of cases where business logic is involved. There could very well be exceptions, though. Still, be sure to examine each situation where your code violates this rule for a potential leak of implementation details.

5.2.3 Well-designed API and encapsulation

Maintaining a well-designed API relates to the notion of encapsulation. As you might recall from chapter 3, *encapsulation* is the act of protecting your code against inconsistencies, also known as *invariant violations*. An *invariant* is a condition that should be held true at all times. The `User` class from the previous example had one such invariant: no user could have a name that exceeded 50 characters.

Exposing implementation details goes hand in hand with invariant violations—the former often leads to the latter. Not only did the original version of `User` leak its implementation details, but it also didn't maintain proper encapsulation. It allowed the client to bypass the invariant and assign a new name to a user without normalizing that name first.

Encapsulation is crucial for code base maintainability in the long run. The reason why is *complexity*. Code complexity is one of the biggest challenges you'll face in software development. The more complex the code base becomes, the harder it is to work with, which, in turn, results in slowing down development speed and increasing the number of bugs.

Without encapsulation, you have no practical way to cope with ever-increasing code complexity. When the code's API doesn't guide you through what is and what isn't allowed to be done with that code, you have to keep a lot of information in mind to make sure you don't introduce inconsistencies with new code changes. This brings an additional mental burden to the process of programming. Remove as much of that burden from yourself as possible. *You cannot trust yourself to do the right thing all the time—so, eliminate the very possibility of doing the wrong thing*. The best way to do so is to maintain proper encapsulation so that your code base doesn't even provide an option for you to do anything incorrectly. Encapsulation ultimately serves the same goal as unit testing: it enables sustainable growth of your software project.

There's a similar principle: *tell-don't-ask*. It was coined by Martin Fowler (https://martinfowler.com/bliki/TellDontAsk.html) and stands for bundling data with the functions that operate on that data. You can view this principle as a corollary to the practice of encapsulation. Code encapsulation is a goal, whereas bundling data and functions together, as well as hiding implementation details, are the means to achieve that goal:

- *Hiding implementation details* helps you remove the class's internals from the eyes of its clients, so there's less risk of corrupting those internals.
- *Bundling data and operations* helps to make sure these operations don't violate the class's invariants.

5.2.4 Leaking implementation details: An example with state

The example shown in listing 5.5 demonstrated an operation (the `NormalizeName` method) that was an implementation detail leaking to the public API. Let's also look at an example with state. The following listing contains the `MessageRenderer` class you saw in chapter 4. It uses a collection of sub-renderers to generate an HTML representation of a message containing a header, a body, and a footer.

Listing 5.7 State as an implementation detail

```
public class MessageRenderer : IRenderer
{
    public IReadOnlyList<IRenderer> SubRenderers { get; }

    public MessageRenderer()
    {
        SubRenderers = new List<IRenderer>
        {
            new HeaderRenderer(),
            new BodyRenderer(),
```

```
            new FooterRenderer()
        };
    }

    public string Render(Message message)
    {
        return SubRenderers
            .Select(x => x.Render(message))
            .Aggregate("", (str1, str2) => str1 + str2);
    }
}
```

The sub-renderers collection is public. But is it part of observable behavior? Assuming that the client's goal is to render an HTML message, the answer is no. The only class member such a client would need is the `Render` method itself. Thus `SubRenderers` is also a leaking implementation detail.

I bring up this example again for a reason. As you may remember, I used it to illustrate a brittle test. That test was brittle precisely because it was tied to this implementation detail—it checked to see the collection's composition. The brittleness was fixed by re-targeting the test at the `Render` method. The new version of the test verified the resulting message—the only output the client code cared about, the observable behavior.

As you can see, there's an intrinsic connection between good unit tests and a well-designed API. By making all implementation details private, you leave your tests no choice other than to verify the code's observable behavior, which automatically improves their resistance to refactoring.

TIP Making the API well-designed automatically improves unit tests.

Another guideline flows from the definition of a well-designed API: you should expose the absolute minimum number of operations and state. Only code that directly helps clients achieve their goals should be made public. Everything else is implementation details and thus must be hidden behind the private API.

Note that there's no such problem as leaking observable behavior, which would be symmetric to the problem of leaking implementation details. While you can expose an implementation detail (a method or a class that is not supposed to be used by the client), you can't hide an observable behavior. Such a method or class would no longer have an immediate connection to the client goals, because the client wouldn't be able to directly use it anymore. Thus, by definition, this code would cease to be part of observable behavior. Table 5.1 sums it all up.

Table 5.1 The relationship between the code's publicity and purpose. Avoid making implementation details public.

	Observable behavior	Implementation detail
Public	Good	Bad
Private	N/A	Good

5.3 The relationship between mocks and test fragility

The previous sections defined a mock and showed the difference between observable behavior and an implementation detail. In this section, you will learn about hexagonal architecture, the difference between internal and external communications, and (finally!) the relationship between mocks and test fragility.

5.3.1 Defining hexagonal architecture

A typical application consists of two layers, domain and application services, as shown in figure 5.8. The *domain layer* resides in the middle of the diagram because it's the central part of your application. It contains the *business logic*: the essential functionality your application is built for. The domain layer and its business logic differentiate this application from others and provide a competitive advantage for the organization.

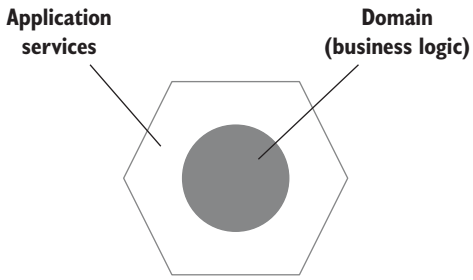


Figure 5.8 A typical application consists of a domain layer and an application services layer. The domain layer contains the application's business logic; application services tie that logic to business use cases.

The application services layer sits on top of the domain layer and orchestrates communication between that layer and the external world. For example, if your application is a RESTful API, all requests to this API hit the application services layer first. This layer then coordinates the work between domain classes and out-of-process dependencies. Here's an example of such coordination for the application service. It does the following:

- Queries the database and uses the data to materialize a domain class instance
- Invokes an operation on that instance
- Saves the results back to the database

The combination of the application services layer and the domain layer forms a *hexagon*, which itself represents your application. It can interact with other applications, which are represented with their own hexagons (see figure 5.9). These other applications could be an SMTP service, a third-party system, a message bus, and so on. A set of interacting hexagons makes up a *hexagonal architecture*.

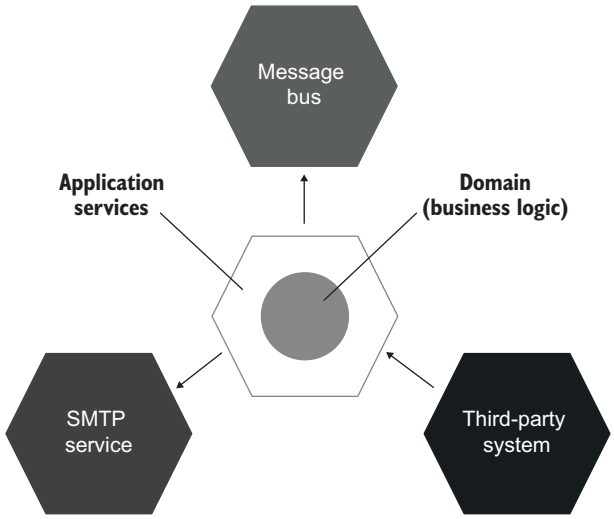


Figure 5.9 A hexagonal architecture is a set of interacting applications—hexagons.

The term *hexagonal architecture* was introduced by Alistair Cockburn. Its purpose is to emphasize three important guidelines:

- *The separation of concerns between the domain and application services layers*—Business logic is the most important part of the application. Therefore, the domain layer should be accountable only for that business logic and exempted from all other responsibilities. Those responsibilities, such as communicating with external applications and retrieving data from the database, must be attributed to application services. Conversely, the application services shouldn't contain any business logic. Their responsibility is to adapt the domain layer by translating the incoming requests into operations on domain classes and then persisting the results or returning them back to the caller. You can view the domain layer as a collection of the application's domain knowledge (*how-to's*) and the application services layer as a set of business use cases (*what-to's*).
- *Communications inside your application*—Hexagonal architecture prescribes a one-way flow of dependencies: from the application services layer to the domain layer. Classes inside the domain layer should only depend on each other; they should not depend on classes from the application services layer. This guideline flows from the previous one. The separation of concerns between the application services layer and the domain layer means that the former knows about the latter, but the opposite is not true. The domain layer should be fully isolated from the external world.
- *Communications between applications*—External applications connect to your application through a common interface maintained by the application services layer. No one has a direct access to the domain layer. Each side in a hexagon represents a connection into or out of the application. Note that although a

hexagon has six sides, it doesn't mean your application can only connect to six other applications. The number of connections is arbitrary. The point is that there can be many such connections.

Each layer of your application exhibits observable behavior and contains its own set of implementation details. For example, observable behavior of the domain layer is the sum of this layer's operations and state that helps the application service layer achieve at least one of its goals. The principles of a well-designed API have a fractal nature: they apply equally to as much as a whole layer or as little as a single class.

When you make each layer's API well-designed (that is, hide its implementation details), your tests also start to have a fractal structure; they verify behavior that helps achieve the same goals but at different levels. A test covering an application service checks to see how this service attains an overarching, coarse-grained goal posed by the external client. At the same time, a test working with a domain class verifies a subgoal that is part of that greater goal (figure 5.10).

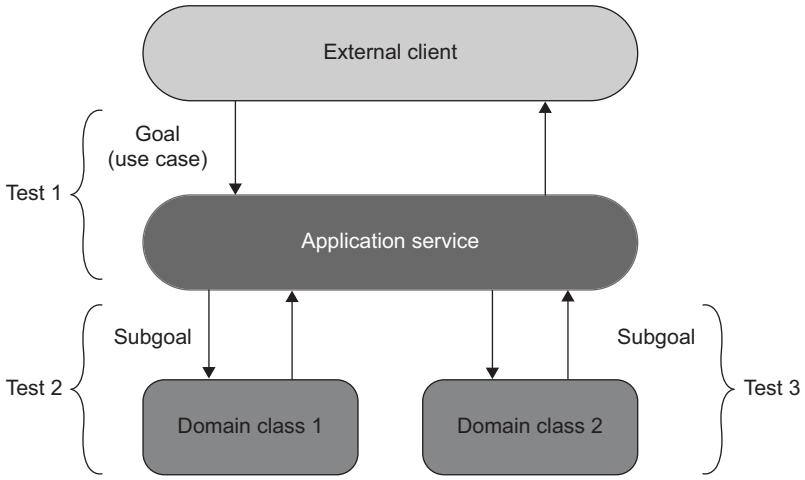


Figure 5.10 Tests working with different layers have a fractal nature: they verify the same behavior at different levels. A test of an application service checks to see how the overall business use case is executed. A test working with a domain class verifies an intermediate subgoal on the way to use-case completion.

You might remember from previous chapters how I mentioned that you should be able to trace any test back to a particular business requirement. Each test should tell a story that is meaningful to a domain expert, and if it doesn't, that's a strong indication that the test couples to implementation details and therefore is brittle. I hope now you can see why.

Observable behavior flows inward from outer layers to the center. The overarching goal posed by the external client gets translated into subgoals achieved by individual

domain classes. Each piece of observable behavior in the domain layer therefore preserves the connection to a particular business use case. You can trace this connection recursively from the innermost (domain) layer outward to the application services layer and then to the needs of the external client. This traceability follows from the definition of observable behavior. For a piece of code to be part of observable behavior, it needs to help the client achieve one of its goals. For a domain class, the client is an application service; for the application service, it's the external client itself.

Tests that verify a code base with a well-designed API also have a connection to business requirements because those tests tie to the observable behavior only. A good example is the `User` and `UserController` classes from listing 5.6 (I'm repeating the code here for convenience).

Listing 5.8 A domain class with an application service

```
public class User
{
    private string _name;
    public string Name
    {
        get => _name;
        set => _name = NormalizeName(value);
    }

    private string NormalizeName(string name)
    {
        /* Trim name down to 50 characters */
    }
}

public class UserController
{
    public void RenameUser(int userId, string newName)
    {
        User user = GetUserFromDatabase(userId);
        user.Name = newName;
        SaveUserToDatabase(user);
    }
}
```

`UserController` in this example is an application service. Assuming that the external client doesn't have a specific goal of normalizing user names, and all names are normalized solely due to restrictions from the application itself, the `NormalizeName` method in the `User` class can't be traced to the client's needs. Therefore, it's an implementation detail and should be made private (we already did that earlier in this chapter). Moreover, tests shouldn't check this method directly. They should verify it only as part of the class's observable behavior—the `Name` property's setter in this example.

This guideline of always tracing the code base's public API to business requirements applies to the vast majority of domain classes and application services but less

so to utility and infrastructure code. The individual problems such code solves are often too low-level and fine-grained and can't be traced to a specific business use case.

5.3.2 Intra-system vs. inter-system communications

There are two types of communications in a typical application: intra-system and inter-system. *Intra-system* communications are communications between classes inside your application. *Inter-system* communications are when your application talks to other applications (figure 5.11).

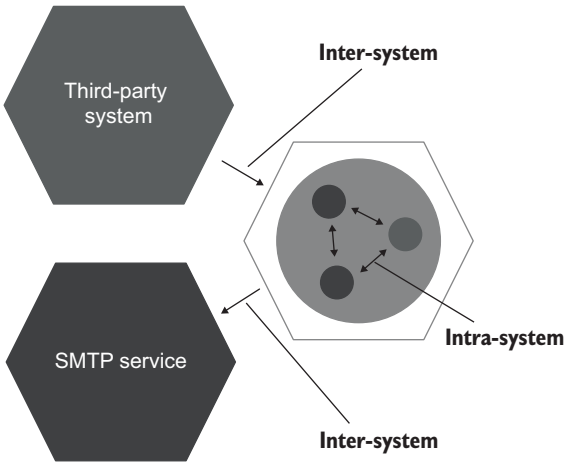


Figure 5.11 There are two types of communications: intra-system (between classes inside the application) and inter-system (between applications).

NOTE Intra-system communications are implementation details; inter-system communications are not.

Intra-system communications are implementation details because the collaborations your domain classes go through in order to perform an operation are not part of their observable behavior. These collaborations don't have an immediate connection to the client's goal. Thus, coupling to such collaborations leads to fragile tests.

Inter-system communications are a different matter. Unlike collaborations between classes inside your application, the way your system talks to the external world forms the observable behavior of that system as a whole. It's part of the contract your application must hold at all times (figure 5.12).

This attribute of inter-system communications stems from the way separate applications evolve together. One of the main principles of such an evolution is maintaining backward compatibility. Regardless of the refactorings you perform inside your system, the communication pattern it uses to talk to external applications should always stay in place, so that external applications can understand it. For example, messages your application emits on a bus should preserve their structure, the calls issued to an SMTP service should have the same number and type of parameters, and so on.

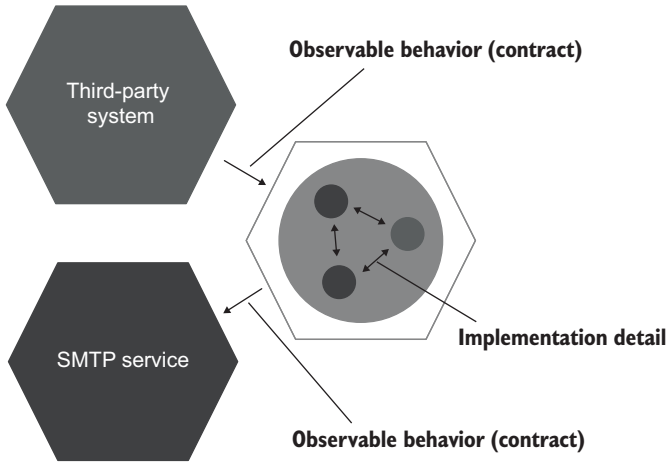


Figure 5.12 **Inter-system communications form the observable behavior of your application as a whole. Intra-system communications are implementation details.**

The use of mocks is beneficial when verifying the communication pattern between your system and external applications. Conversely, using mocks to verify communications between classes inside your system results in tests that couple to implementation details and therefore fall short of the resistance-to-refactoring metric.

5.3.3 Intra-system vs. inter-system communications: An example

To illustrate the difference between intra-system and inter-system communications, I'll expand on the example with the `Customer` and `Store` classes that I used in chapter 2 and earlier in this chapter. Imagine the following business use case:

- A customer tries to purchase a product from a store.
- If the amount of the product in the store is sufficient, then
  - The inventory is removed from the store.
  - An email receipt is sent to the customer.
  - A confirmation is returned.

Let's also assume that the application is an API with no user interface.

In the following listing, the `CustomerController` class is an application service that orchestrates the work between domain classes (`Customer`, `Product`, `Store`) and the external application (`EmailGateway`, which is a proxy to an SMTP service).

Listing 5.9 Connecting the domain model with external applications

```
public class CustomerController
{
    public bool Purchase(int customerId, int productId, int quantity)
```

```
    {
        Customer customer = _customerRepository.GetById(customerId);
        Product product = _productRepository.GetById(productId);

        bool isSuccess = customer.Purchase(
            _mainStore, product, quantity);

        if (isSuccess)
        {
            _emailGateway.SendReceipt(
                customer.Email, product.Name, quantity);
        }

        return isSuccess;
    }
}
```

Validation of input parameters is omitted for brevity. In the `Purchase` method, the customer checks to see if there's enough inventory in the store and, if so, decreases the product amount.

The act of making a purchase is a business use case with both intra-system and inter-system communications. The inter-system communications are those between the `CustomerController` application service and the two external systems: the third-party application (which is also the client initiating the use case) and the email gateway. The intra-system communication is between the `Customer` and the `Store` domain classes (figure 5.13).

In this example, the call to the SMTP service is a side effect that is visible to the external world and thus forms the observable behavior of the application as a whole.

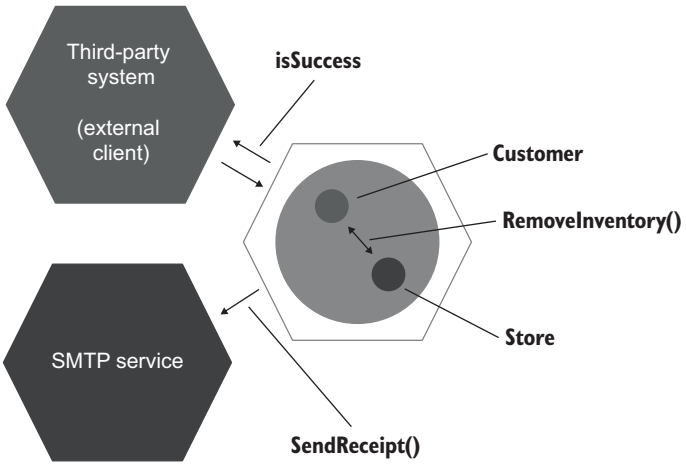


Figure 5.13 The example in listing 5.9 represented using the hexagonal architecture. The communications between the hexagons are inter-system communications. The communication inside the hexagon is intra-system.

It also has a direct connection to the client's goals. The client of the application is the third-party system. This system's goal is to make a purchase, and it expects the customer to receive a confirmation email as part of the successful outcome.

The call to the SMTP service is a legitimate reason to do mocking. It doesn't lead to test fragility because you want to make sure this type of communication stays in place even after refactoring. The use of mocks helps you do exactly that.

The next listing shows an example of a legitimate use of mocks.

Listing 5.10 Mocking that doesn't lead to fragile tests

```
[Fact]
public void Successful_purchase()
{
    var mock = new Mock<IEmailGateway>();
    var sut = new CustomerController(mock.Object);

    bool isSuccess = sut.Purchase(
        customerId: 1, productId: 2, quantity: 5);

    Assert.True(isSuccess);
    mock.Verify(
        x => x.SendReceipt(
            "customer@email.com", "Shampoo", 5),
        Times.Once);
}
```

Verifies that the system sent a receipt about the purchase

Note that the `isSuccess` flag is also observable by the external client and also needs verification. This flag doesn't need mocking, though; a simple value comparison is enough.

Let's now look at a test that mocks the communication between `Customer` and `Store`.

Listing 5.11 Mocking that leads to fragile tests

```
[Fact]
public void Purchase_succeeds_when_enough_inventory()
{
    var storeMock = new Mock<IStore>();
    storeMock
        .Setup(x => x.HasEnoughInventory(Product.Shampoo, 5))
        .Returns(true);
    var customer = new Customer();

    bool success = customer.Purchase(
        storeMock.Object, Product.Shampoo, 5);

    Assert.True(success);
    storeMock.Verify(
        x => x.RemoveInventory(Product.Shampoo, 5),
        Times.Once);
}
```

Unlike the communication between `CustomerController` and the SMTP service, the `RemoveInventory()` method call from `Customer` to `Store` doesn't cross the application boundary: both the caller and the recipient reside inside the application. Also, this method is neither an operation nor a state that helps the client achieve its goals. The client of these two domain classes is `CustomerController` with the goal of making a purchase. The only two members that have an immediate connection to this goal are `customer.Purchase()` and `store.GetInventory()`. The `Purchase()` method initiates the purchase, and `GetInventory()` shows the state of the system after the purchase is completed. The `RemoveInventory()` method call is an intermediate step on the way to the client's goal—an implementation detail.

5.4 The classical vs. London schools of unit testing, revisited

As a reminder from chapter 2 (table 2.1), table 5.2 sums up the differences between the classical and London schools of unit testing.

Table 5.2 The differences between the London and classical schools of unit testing

	Isolation of	A unit is	Uses test doubles for
London school	Units	A class	All but immutable dependencies
Classical school	Unit tests	A class or a set of classes	Shared dependencies

In chapter 2, I mentioned that I prefer the classical school of unit testing over the London school. I hope now you can see why. The London school encourages the use of mocks for all but immutable dependencies and doesn't differentiate between intra-system and inter-system communications. As a result, tests check communications between classes just as much as they check communications between your application and external systems.

This indiscriminate use of mocks is why following the London school often results in tests that couple to implementation details and thus lack resistance to refactoring. As you may remember from chapter 4, the metric of resistance to refactoring (unlike the other three) is mostly a binary choice: a test either has resistance to refactoring or it doesn't. Compromising on this metric renders the test nearly worthless.

The classical school is much better at this issue because it advocates for substituting only dependencies that are shared between tests, which almost always translates into out-of-process dependencies such as an SMTP service, a message bus, and so on. But the classical school is not ideal in its treatment of inter-system communications, either. This school also encourages excessive use of mocks, albeit not as much as the London school.

5.4.1 Not all out-of-process dependencies should be mocked out

Before we discuss out-of-process dependencies and mocking, let me give you a quick refresher on types of dependencies (refer to chapter 2 for more details):

- *Shared dependency*—A dependency shared by tests (not production code)
- *Out-of-process dependency*—A dependency hosted by a process other than the program's execution process (for example, a database, a message bus, or an SMTP service)
- *Private dependency*—Any dependency that is not shared

The classical school recommends avoiding shared dependencies because they provide the means for tests to interfere with each other's execution context and thus prevent those tests from running in parallel. The ability for tests to run in parallel, sequentially, and in any order is called *test isolation*.

If a shared dependency is not out-of-process, then it's easy to avoid reusing it in tests by providing a new instance of it on each test run. In cases where the shared dependency is out-of-process, testing becomes more complicated. You can't instantiate a new database or provision a new message bus before each test execution; that would drastically slow down the test suite. The usual approach is to replace such dependencies with test doubles—mocks and stubs.

Not all out-of-process dependencies should be mocked out, though. *If an out-of-process dependency is only accessible through your application, then communications with such a dependency are not part of your system's observable behavior.* An out-of-process dependency that can't be observed externally, in effect, acts as part of your application (figure 5.14).

Remember, the requirement to always preserve the communication pattern between your application and external systems stems from the necessity to maintain backward compatibility. You have to maintain the way your application talks to external

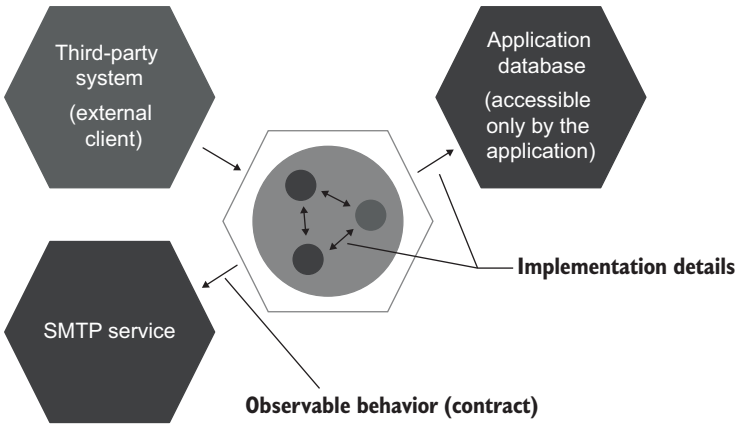


Figure 5.14 Communications with an out-of-process dependency that can't be observed externally are implementation details. They don't have to stay in place after refactoring and therefore shouldn't be verified with mocks.

systems. That's because you can't change those external systems simultaneously with your application; they may follow a different deployment cycle, or you might simply not have control over them.

But when your application acts as a proxy to an external system, and no client can access it directly, the backward-compatibility requirement vanishes. Now you can deploy your application together with this external system, and it won't affect the clients. The communication pattern with such a system becomes an implementation detail.

A good example here is an application database: a database that is used only by your application. No external system has access to this database. Therefore, you can modify the communication pattern between your system and the application database in any way you like, as long as it doesn't break existing functionality. Because that database is completely hidden from the eyes of the clients, you can even replace it with an entirely different storage mechanism, and no one will notice.

The use of mocks for out-of-process dependencies that you have a full control over also leads to brittle tests. You don't want your tests to turn red every time you split a table in the database or modify the type of one of the parameters in a stored procedure. The database and your application must be treated as one system.

This obviously poses an issue. How would you test the work with such a dependency without compromising the feedback speed, the third attribute of a good unit test? You'll see this subject covered in depth in the following two chapters.

5.4.2 Using mocks to verify behavior

Mocks are often said to verify behavior. In the vast majority of cases, they don't. The way each individual class interacts with neighboring classes in order to achieve some goal has nothing to do with observable behavior; it's an implementation detail.

Verifying communications between classes is akin to trying to derive a person's behavior by measuring the signals that neurons in the brain pass among each other. Such a level of detail is too granular. What matters is the behavior that can be traced back to the client goals. The client doesn't care what neurons in your brain light up when they ask you to help. The only thing that matters is the help itself—provided by you in a reliable and professional fashion, of course. Mocks have something to do with behavior only when they verify interactions that cross the application boundary and only when the side effects of those interactions are visible to the external world.

Summary

- *Test double* is an overarching term that describes all kinds of non-production-ready, fake dependencies in tests. There are five variations of test doubles—dummy, stub, spy, mock, and fake—that can be grouped in just two types: mocks and stubs. Spies are functionally the same as mocks; dummies and fakes serve the same role as stubs.
- Mocks help emulate and examine *outcoming interactions*: calls from the SUT to its dependencies that change the state of those dependencies. Stubs help

emulate *incoming interactions*: calls the SUT makes to its dependencies to get input data.

- A mock (the tool) is a class from a mocking library that you can use to create a mock (the test double) or a stub.
- Asserting interactions with stubs leads to *fragile tests*. Such an interaction doesn't correspond to the end result; it's an intermediate step on the way to that result, an implementation detail.
- The command query separation (CQS) principle states that every method should be either a command or a query but not both. Test doubles that substitute commands are mocks. Test doubles that substitute queries are stubs.
- All production code can be categorized along two dimensions: public API versus private API, and observable behavior versus implementation details. Code publicity is controlled by access modifiers, such as `private`, `public`, and `internal` keywords. Code is part of observable behavior when it meets one of the following requirements (any other code is an implementation detail):
  - It exposes an operation that helps the client achieve one of its goals. An *operation* is a method that performs a calculation or incurs a side effect.
  - It exposes a state that helps the client achieve one of its goals. *State* is the current condition of the system.
- *Well-designed code* is code whose observable behavior coincides with the public API and whose implementation details are hidden behind the private API. A code *leaks* implementation details when its public API extends beyond the observable behavior.
- *Encapsulation* is the act of protecting your code against invariant violations. Exposing implementation details often entails a breach in encapsulation because clients can use implementation details to bypass the code's invariants.
- *Hexagonal architecture* is a set of interacting applications represented as hexagons. Each hexagon consists of two layers: domain and application services.
- Hexagonal architecture emphasizes three important aspects:
  - Separation of concerns between the domain and application services layers. The domain layer should be responsible for the business logic, while the application services should orchestrate the work between the domain layer and external applications.
  - A one-way flow of dependencies from the application services layer to the domain layer. Classes inside the domain layer should only depend on each other; they should not depend on classes from the application services layer.
  - External applications connect to your application through a common interface maintained by the application services layer. No one has a direct access to the domain layer.
- Each layer in a hexagon exhibits observable behavior and contains its own set of implementation details.

- There are two types of communications in an application: intra-system and inter-system. *Intra-system* communications are communications between classes inside the application. *Inter-system* communication is when the application talks to external applications.
- Intra-system communications are implementation details. Inter-system communications are part of observable behavior, with the exception of external systems that are accessible only through your application. Interactions with such systems are implementation details too, because the resulting side effects are not observed externally.
- Using mocks to assert intra-system communications leads to *fragile* tests. Mocking is legitimate only when it's used for inter-system communications—communications that cross the application boundary—and only when the side effects of those communications are visible to the external world.

Chapter 9 from *Unit Testing Principles, Practices, and Patterns* by Vladimir Khorikov

This chapter introduces mocking best practices, such as mocking the types at the edges of your system and only mocking the types that you own. You will also learn when and how to write handwritten mocks (spies) instead of regular mocks.

9 Mocking best practices

This chapter covers

- Maximizing the value of mocks
- Replacing mocks with spies
- Mocking best practices

As you might remember from chapter 5, a mock is a test double that helps to emulate and examine interactions between the system under test and its dependencies. As you might also remember from chapter 8, mocks should only be applied to *unmanaged dependencies* (interactions with such dependencies are observable by external applications). Using mocks for anything else results in *brittle tests* (tests that lack the metric of resistance to refactoring). When it comes to mocks, adhering to this one guideline will get you about two-thirds of the way to success.

This chapter shows the remaining guidelines that will help you develop integration tests that have the greatest possible value by maxing out mocks' resistance to refactoring and protection against regressions. I'll first show a typical use of mocks, describe its drawbacks, and then demonstrate how you can overcome those drawbacks.

9.1 Maximizing mocks' value

It's important to limit the use of mocks to unmanaged dependencies, but that's only the first step on the way to maximizing the value of mocks. This topic is best explained with an example, so I'll continue using the CRM system from earlier chapters as a sample project. I'll remind you of its functionality and show the integration test we ended up with. After that, you'll see how that test can be improved with regard to mocking.

As you might recall, the CRM system currently supports only one use case: changing a user's email. The following listing shows where we left off with the controller.

Listing 9.1 User controller

```
public class UserController
{
    private readonly Database _database;
    private readonly EventDispatcher _eventDispatcher;

    public UserController(
        Database database,
        IMessageBus messageBus,
        IDomainLogger domainLogger)
    {
        _database = database;
        _eventDispatcher = new EventDispatcher(
            messageBus, domainLogger);
    }

    public string ChangeEmail(int userId, string newEmail)
    {
        object[] userData = _database.GetUserById(userId);
        User user = UserFactory.Create(userData);

        string error = user.CanChangeEmail();
        if (error != null)
            return error;

        object[] companyData = _database.GetCompany();
        Company company = CompanyFactory.Create(companyData);

        user.ChangeEmail(newEmail, company);

        _database.SaveCompany(company);
        _database.SaveUser(user);
        _eventDispatcher.Dispatch(user.DomainEvents);

        return "OK";
    }
}
```

Note that there's no longer any diagnostic logging, but support logging (the `IDomainLogger` interface) is still in place (see chapter 8 for more details). Also, listing 9.1 introduces a new class: the `EventDispatcher`. It converts domain events generated by

the domain model into calls to *unmanaged* dependencies (something that the controller previously did by itself), as shown next.

Listing 9.2 Event dispatcher

```
public class EventDispatcher
{
    private readonly IMessageBus _messageBus;
    private readonly IDomainLogger _domainLogger;

    public EventDispatcher(
        IMessageBus messageBus,
        IDomainLogger domainLogger)
    {
        _domainLogger = domainLogger;
        _messageBus = messageBus;
    }

    public void Dispatch(List<IDomainEvent> events)
    {
        foreach (IDomainEvent ev in events)
        {
            Dispatch(ev);
        }
    }

    private void Dispatch(IDomainEvent ev)
    {
        switch (ev)
        {
            case EmailChangedEvent emailChangedEvent:
                _messageBus.SendEmailChangedMessage(
                    emailChangedEvent.UserId,
                    emailChangedEvent.NewEmail);
                break;

            case UserTypeChangedEvent userTypeChangedEvent:
                _domainLogger.UserTypeHasChanged(
                    userTypeChangedEvent.UserId,
                    userTypeChangedEvent.OldType,
                    userTypeChangedEvent.NewType);
                break;
        }
    }
}
```

Finally, the following listing shows the integration test. This test goes through all out-of-process dependencies (both managed and unmanaged).

Listing 9.3 Integration test

```
[Fact]
public void Changing_email_from_corporate_to_non_corporate()
{
```

```
    // Arrange
    var db = new Database(ConnectionString);
    User user = CreateUser("user@mycorp.com", UserType.Employee, db);
    CreateCompany("mycorp.com", 1, db);

    var messageBusMock = new Mock<IMessageBus>();
    var loggerMock = new Mock<IDomainLogger>();
    var sut = new UserController(
        db, messageBusMock.Object, loggerMock.Object);

    // Act
    string result = sut.ChangeEmail(user.UserId, "new@gmail.com");

    // Assert
    Assert.Equal("OK", result);

    object[] userData = db.GetUserById(user.UserId);
    User userFromDb = UserFactory.Create(userData);
    Assert.Equal("new@gmail.com", userFromDb.Email);
    Assert.Equal(UserType.Customer, userFromDb.Type);

    object[] companyData = db.GetCompany();
    Company companyFromDb = CompanyFactory.Create(companyData);
    Assert.Equal(0, companyFromDb.NumberOfEmployees);

    messageBusMock.Verify(
        x => x.SendEmailChangedMessage(
             user.UserId, "new@gmail.com"),
        Times.Once);
    loggerMock.Verify(
        x => x.UserTypeHasChanged(
            user.UserId,
            UserType.Employee,
            UserType.Customer),
        Times.Once);
}
```

Sets up the mocks

Verifies the interactions with the mocks

This test mocks out two unmanaged dependencies: `IMessageBus` and `IDomainLogger`. I'll focus on `IMessageBus` first. We'll discuss `IDomainLogger` later in this chapter.

9.1.1 Verifying interactions at the system edges

Let's discuss why the mocks used by the integration test in listing 9.3 aren't ideal in terms of their protection against regressions and resistance to refactoring and how we can fix that.

> **TIP** When mocking, always adhere to the following guideline: verify interactions with unmanaged dependencies at the very edges of your system.

The problem with `messageBusMock` in listing 9.3 is that the `IMessageBus` interface doesn't reside at the system's edge. Look at that interface's implementation.

Listing 9.4 Message bus

```
public interface IMessageBus
{
    void SendEmailChangedMessage(int userId, string newEmail);
}

public class MessageBus : IMessageBus
{
    private readonly IBus _bus;

    public void SendEmailChangedMessage(
        int userId, string newEmail)
    {
        _bus.Send("Type: USER EMAIL CHANGED; " +
            $"Id: {userId}; " +
            $"NewEmail: {newEmail}");
    }
}

public interface IBus
{
    void Send(string message);
}
```

Both the `IMessageBus` and `IBus` interfaces (and the classes implementing them) belong to our project's code base. `IBus` is a wrapper on top of the message bus SDK library (provided by the company that develops that message bus). This wrapper encapsulates nonessential technical details, such as connection credentials, and exposes a nice, clean interface for sending arbitrary text messages to the bus. `IMessageBus` is a wrapper on top of `IBus`; it defines messages specific to your domain. `IMessageBus` helps you keep all such messages in one place and reuse them across the application.

It's possible to merge the `IBus` and `IMessageBus` interfaces together, but that would be a suboptimal solution. These two responsibilities—hiding the external library's complexity and holding all application messages in one place—are best kept separated. This is the same situation as with `ILogger` and `IDomainLogger`, which you saw in chapter 8. `IDomainLogger` implements specific logging functionality required by the business, and it does that by using the generic `ILogger` behind the scenes.

Figure 9.1 shows where `IBus` and `IMessageBus` stand from a hexagonal architecture perspective: `IBus` is the last link in the chain of types between the controller and the message bus, while `IMessageBus` is only an intermediate step on the way.

Mocking `IBus` instead of `IMessageBus` maximizes the mock's protection against regressions. As you might remember from chapter 4, protection against regressions is a function of the amount of code that is executed during the test. Mocking the very last type that communicates with the unmanaged dependency increases the number of classes the integration test goes through and thus improves the protection. This guideline is also the reason you don't want to mock `EventDispatcher`. It resides even further away from the edge of the system, compared to `IMessageBus`.

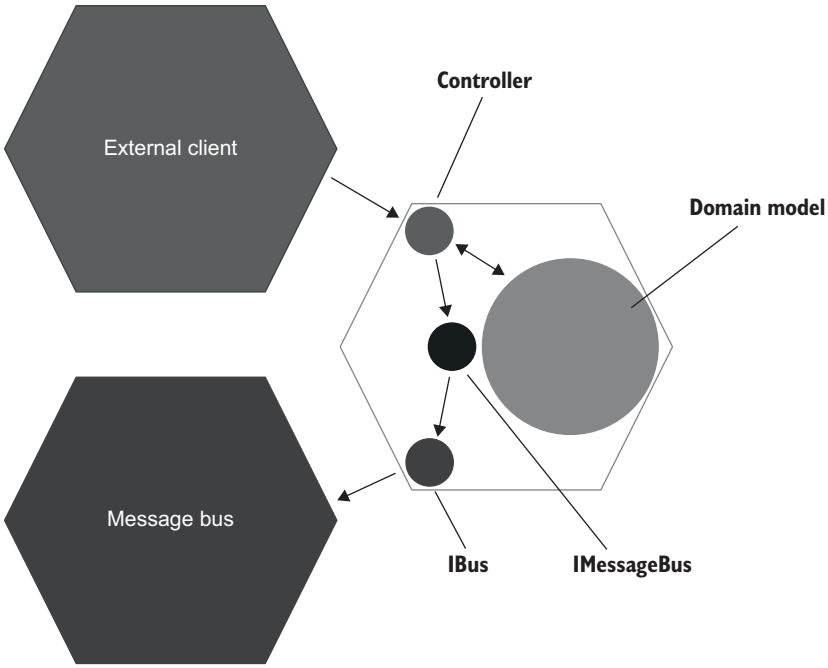


Figure 9.1 **`IBus` resides at the system's edge; `IMessageBus` is only an intermediate link in the chain of types between the controller and the message bus. Mocking `IBus` instead of `IMessageBus` achieves the best protection against regressions.**

Here's the integration test after retargeting it from `IMessageBus` to `IBus`. I'm omitting the parts that didn't change from listing 9.3.

Listing 9.5 Integration test targeting `IBus`

```
[Fact]
public void Changing_email_from_corporate_to_non_corporate()
{
    var busMock = new Mock<IBus>();
    var messageBus = new MessageBus(busMock.Object);
    var loggerMock = new Mock<IDomainLogger>();
    var sut = new UserController(db, messageBus, loggerMock.Object);

    /* ... */

    busMock.Verify(
        x => x.Send(
            "Type: USER EMAIL CHANGED; " +
            $"Id: {user.UserId}; " +
            "NewEmail: new@gmail.com"),
        Times.Once);
}
```

Uses a concrete class instead of the interface

Verifies the actual message sent to the bus

Notice how the test now uses the concrete `MessageBus` class and not the corresponding `IMessageBus` interface. `IMessageBus` is an interface with a single implementation, and, as you'll remember from chapter 8, mocking is the only legitimate reason to have such interfaces. Because we no longer mock `IMessageBus`, this interface can be deleted and its usages replaced with `MessageBus`.

Also notice how the test in listing 9.5 checks the text message sent to the bus. Compare it to the previous version:

```
messageBusMock.Verify(
    x => x.SendEmailChangedMessage(user.UserId, "new@gmail.com"),
    Times.Once);
```

There's a huge difference between verifying a call to a custom class that you wrote and the actual text sent to external systems. External systems expect text messages from your application, not calls to classes like `MessageBus`. In fact, text messages are the only side effect observable externally; classes that participate in producing those messages are mere implementation details. Thus, in addition to the increased protection against regressions, verifying interactions at the very edges of your system also improves resistance to refactoring. The resulting tests are less exposed to potential false positives; no matter what refactorings take place, such tests won't turn red as long as the message's structure is preserved.

The same mechanism is at play here as the one that gives integration and end-to-end tests additional resistance to refactoring compared to unit tests. They are more detached from the code base and, therefore, aren't affected as much during low-level refactorings.

> **TIP** A call to an unmanaged dependency goes through several stages before it leaves your application. Pick the last such stage. It is the best way to ensure backward compatibility with external systems, which is the goal that mocks help you achieve.

9.1.2 Replacing mocks with spies

As you may remember from chapter 5, a *spy* is a variation of a test double that serves the same purpose as a mock. The only difference is that spies are written manually, whereas mocks are created with the help of a mocking framework. Indeed, spies are often called *handwritten mocks*.

It turns out that, when it comes to classes residing at the system edges, *spies are superior to mocks*. Spies help you reuse code in the assertion phase, thereby reducing the test's size and improving readability. The next listing shows an example of a spy that works on top of `IBus`.

Listing 9.6 A spy (also known as a handwritten mock)

```
public interface IBus
{
    void Send(string message);
}
```

```
public class BusSpy : IBus
{
    private List<string> _sentMessages =
        new List<string>();

    public void Send(string message)
    {
        _sentMessages.Add(message);
    }

    public BusSpy ShouldSendNumberOfMessages(int number)
    {
        Assert.Equal(number, _sentMessages.Count);
        return this;
    }

    public BusSpy WithEmailChangedMessage(int userId, string newEmail)
    {
        string message = "Type: USER EMAIL CHANGED; " +
            $"Id: {userId}; " +
            $"NewEmail: {newEmail}";
        Assert.Contains(
            _sentMessages, x => x == message);

        return this;
    }
}
```

Stores all sent messages locally

Asserts that the message has been sent

The following listing is a new version of the integration test. Again, I'm showing only the relevant parts.

Listing 9.7 Using the spy from listing 6.43

```
[Fact]
public void Changing_email_from_corporate_to_non_corporate()
{
    var busSpy = new BusSpy();
    var messageBus = new MessageBus(busSpy);
    var loggerMock = new Mock<IDomainLogger>();
    var sut = new UserController(db, messageBus, loggerMock.Object);

    /* ... */

    busSpy.ShouldSendNumberOfMessages(1)
        .WithEmailChangedMessage(user.UserId, "new@gmail.com");
}
```

Verifying the interactions with the message bus is now succinct and expressive, thanks to the fluent interface that `BusSpy` provides. With that fluent interface, you can chain together several assertions, thus forming cohesive, almost plain-English sentences.

TIP You can rename `BusSpy` into `BusMock`. As I mentioned earlier, the difference between a mock and a spy is an implementation detail. Most programmers

aren't familiar with the term *spy*, though, so renaming the spy as `BusMock` can save your colleagues unnecessary confusion.

There's a reasonable question to be asked here: didn't we just make a full circle and come back to where we started? The version of the test in listing 9.7 looks a lot like the earlier version that mocked `IMessageBus`:

```
messageBusMock.Verify(
    x => x.SendEmailChangedMessage(
        user.UserId, "new@gmail.com"),
    Times.Once);
```

Same as WithEmailChanged-Message(user.UserId, "new@gmail.com")

Same as ShouldSendNumberOfMessages(1)

These assertions are similar because both `BusSpy` and `MessageBus` are wrappers on top of `IBus`. But there's a crucial difference between the two: `BusSpy` is part of the test code, whereas `MessageBus` belongs to the production code. This difference is important because *you shouldn't rely on the production code when making assertions in tests.*

Think of your tests as auditors. A good auditor wouldn't just take the auditee's words at face value; they would double-check everything. The same is true with the spy: it provides an independent checkpoint that raises an alarm when the message structure is changed. On the other hand, a mock on `IMessageBus` puts too much trust in the production code.

9.1.3 What about IDomainLogger?

The mock that previously verified interactions with `IMessageBus` is now targeted at `IBus`, which resides at the system's edge. Here are the current mock assertions in the integration test.

Listing 9.8 Mock assertions

```
busSpy.ShouldSendNumberOfMessages(1)
    .WithEmailChangedMessage(
        user.UserId, "new@gmail.com");
```

Checks interactions with IBus

```
loggerMock.Verify(
    x => x.UserTypeHasChanged(
        user.UserId,
        UserType.Employee,
        UserType.Customer),
    Times.Once);
```

Checks interactions with IDomainLogger

Note that just as `MessageBus` is a wrapper on top of `IBus`, `DomainLogger` is a wrapper on top of `ILogger` (see chapter 8 for more details). Shouldn't the test be retargeted at `ILogger`, too, because this interface also resides at the application boundary?

In most projects, such retargeting isn't necessary. While the logger and the message bus are unmanaged dependencies and, therefore, both require maintaining backward compatibility, the accuracy of that compatibility doesn't have to be the same. With the message bus, it's important not to allow *any* changes to the structure of

the messages, because you never know how external systems will react to such changes. But the exact structure of text logs is not that important for the intended audience (support staff and system administrators). What's important is the existence of those logs and the information they carry. Thus, mocking `IDomainLogger` alone provides the necessary level of protection.

9.2 Mocking best practices

You've learned two major mocking best practices so far:

- Applying mocks to unmanaged dependencies only
- Verifying the interactions with those dependencies at the very edges of your system

In this section, I explain the remaining best practices:

- Using mocks in integration tests only, not in unit tests
- Always verifying the number of calls made to the mock
- Mocking only types that you own

9.2.1 Mocks are for integration tests only

The guideline saying that mocks are for integration tests only, and that you shouldn't use mocks in unit tests, stems from the foundational principle described in chapter 7: the separation of business logic and orchestration. Your code should either communicate with out-of-process dependencies or be complex, but never both. This principle naturally leads to the formation of two distinct layers: the domain model (that handles complexity) and controllers (that handle the communication).

Tests on the domain model fall into the category of unit tests; tests covering controllers are integration tests. Because mocks are for unmanaged dependencies only, and because controllers are the only code working with such dependencies, you should only apply mocking when testing controllers—in integration tests.

9.2.2 Not just one mock per test

You might sometimes hear the guideline of having only one mock per test. According to this guideline, if you have more than one mock, you are likely testing several things at a time.

This is a misconception that follows from a more foundational misunderstanding covered in chapter 2: that a *unit* in a unit test refers to a *unit of code*, and all such units must be tested in isolation from each other. On the contrary: the term *unit* means a *unit of behavior*, not a unit of code. The amount of code it takes to implement such a unit of behavior is irrelevant. It could span across multiple classes, a single class, or take up just a tiny method.

With mocks, the same principle is at play: *it's irrelevant how many mocks it takes to verify a unit of behavior*. Earlier in this chapter, it took us two mocks to check the scenario of changing the user email from corporate to non-corporate: one for the logger and

the other for the message bus. That number could have been larger. In fact, you don't have control over how many mocks to use in an integration test. The number of mocks depends solely on the number of unmanaged dependencies participating in the operation.

9.2.3 Verifying the number of calls

When it comes to communications with unmanaged dependencies, it's important to ensure both of the following:

- The existence of expected calls
- The absence of unexpected calls

This requirement, once again, stems from the need to maintain backward compatibility with unmanaged dependencies. The compatibility must go both ways: your application shouldn't omit messages that external systems expect, and it also shouldn't produce unexpected messages. It's not enough to check that the system under test sends a message like this:

```
messageBusMock.Verify(
    x => x.SendEmailChangedMessage(user.UserId, "new@gmail.com"));
```

You also need to ensure that this message is sent exactly once:

```
messageBusMock.Verify(
    x => x.SendEmailChangedMessage(user.UserId, "new@gmail.com"),
    Times.Once);
```

Ensures that the method is called only once

With most mocking libraries, you can also explicitly verify that no other calls are made on the mock. In Moq (the mocking library of my choice), this verification looks as follows:

```
messageBusMock.Verify(
    x => x.SendEmailChangedMessage(user.UserId, "new@gmail.com"),
    Times.Once);
messageBusMock.VerifyNoOtherCalls();
```

The additional check

`BusSpy` implements this functionality, too:

```
busSpy
    .ShouldSendNumberOfMessages(1)
    .WithEmailChangedMessage(user.UserId, "new@gmail.com");
```

The spy's check `ShouldSendNumberOfMessages(1)` encompasses both `Times.Once` and `VerifyNoOtherCalls()` verifications from the mock.

9.2.4 Only mock types that you own

The last guideline I'd like to talk about is mocking only types that you own. It was first introduced by Steve Freeman and Nat Pryce.[2] The guideline states that you should always write your own adapters on top of third-party libraries and mock those adapters instead of the underlying types. A few of their arguments are as follows:

- You often don't have a deep understanding of how the third-party code works.
- Even if that code already provides built-in interfaces, it's risky to mock those interfaces, because you have to be sure the behavior you mock matches what the external library actually does.
- Adapters abstract non-essential technical details of the third-party code and define the relationship with the library in your application's terms.

I fully agree with this analysis. Adapters, in effect, act as an anti-corruption layer between your code and the external world.[3] These help you to

- Abstract the underlying library's complexity
- Only expose features you need from the library
- Do that using your project's domain language

The `IBus` interface in our sample CRM project serves exactly that purpose. Even if the underlying message bus's library provides as nice and clean an interface as `IBus`, you are still better off introducing your own wrapper on top of it. You never know how the third-party code will change when you upgrade the library. Such an upgrade could cause a ripple effect across the whole code base! The additional abstraction layer restricts that ripple effect to just one class: the adapter itself.

Note that the "mock your own types" guideline *doesn't* apply to in-process dependencies. As I explained previously, mocks are for unmanaged dependencies only. Thus, there's no need to abstract in-memory or managed dependencies. For instance, if a library provides a date and time API, you can use that API as-is, because it doesn't reach out to unmanaged dependencies. Similarly, there's no need to abstract an ORM as long as it's used for accessing a database that isn't visible to external applications. Of course, you can introduce your own wrapper on top of any library, but it's rarely worth the effort for anything other than unmanaged dependencies.

Summary

- Verify interactions with an unmanaged dependency at the very edges of your system. Mock the last type in the chain of types between the controller and the unmanaged dependency. This helps you increase both protection against regressions (due to more code being validated by the integration test) and

[2] See page 69 in *Growing Object-Oriented Software, Guided by Tests* by Steve Freeman and Nat Pryce (Addison-Wesley Professional, 2009).

[3] See *Domain-Driven Design: Tackling Complexity in the Heart of Software* by Eric Evans (Addison-Wesley, 2003).

resistance to refactoring (due to detaching the mock from the code's implementation details).

- *Spies* are handwritten mocks. When it comes to classes residing at the system's edges, spies are superior to mocks. They help you reuse code in the assertion phase, thereby reducing the test's size and improving readability.
- Don't rely on production code when making assertions. Use a separate set of literals and constants in tests. Duplicate those literals and constants from the production code if necessary. Tests should provide a checkpoint independent of the production code. Otherwise, you risk producing *tautology tests* (tests that don't verify anything and contain semantically meaningless assertions).
- Not all unmanaged dependencies require the same level of backward compatibility. If the exact structure of the message isn't important, and you only want to verify the existence of that message and the information it carries, you can ignore the guideline of verifying interactions with unmanaged dependencies at the very edges of your system. The typical example is logging.
- Because mocks are for unmanaged dependencies only, and because controllers are the only code working with such dependencies, you should only apply mocking when testing controllers—in integration tests. Don't use mocks in unit tests.
- The number of mocks used in a test is irrelevant. That number depends solely on the number of unmanaged dependencies participating in the operation.
- Ensure both the existence of *expected* calls and the absence of *unexpected* calls to mocks.
- Only mock types that you own. Write your own adapters on top of third-party libraries that provide access to unmanaged dependencies. Mock those adapters instead of the underlying types.

Chapter 5 from *The Art of Unit Testing*
by Roy Osherove

This chapter contains some practical tips for using mocking frameworks, with examples in JavaScript. It provides some practical tips for picking the right framework based on its abilities.

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`

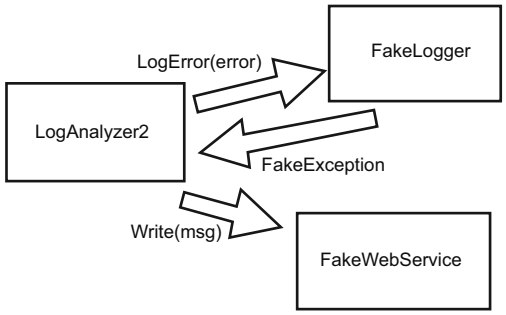


Figure 5.1 The logger will be stubbed out to simulate an exception, and a fake web service will be used as a mock to see if it was called correctly. The whole test will be about how `LogAnalyzer2` interacts with other objects.

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")));
}
```

1 Simulate the error

2 Uses mock to check log call

Notice that you use a stub 1 to trigger the event and a mock 2 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.

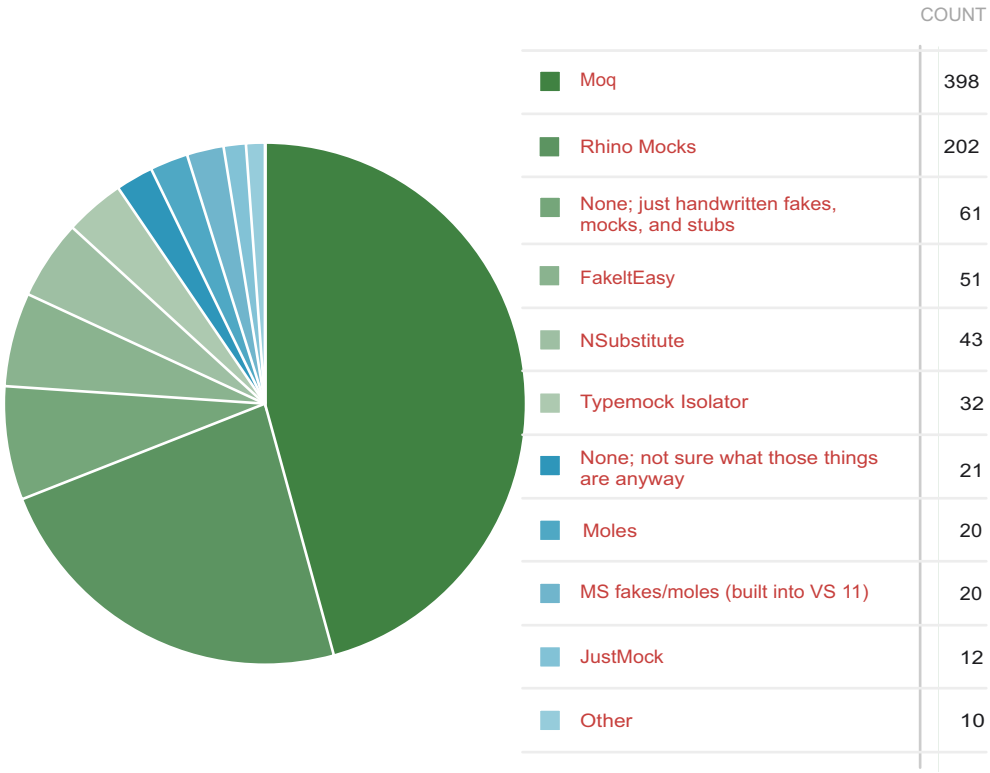


Figure 5.2 **Isolation framework usage among my blog readers**

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.

index

A

adapters 41
API (application programming interface) 14, 21, 41
  public vs. private 9
  well-designed 10–11, 15, 18
Arg class 51
arrange-act-assert 47, 49–50

B

brittle tests 26, 30
bugs 14
business logic 16–17

C

classical school of unit testing
  mocks 24–26
    mocking out out-of-process dependencies 25–26
    using mocks to verify behavior 26
code, avoiding unreadable 60
code complexity 14
command query separation. *See* CQS principle
commands 7
communications
  between applications 17, 20
  between classes in application 20, 26
context argument 51
controllers 39
CQS principle 7–8

D

data, bundling 14
dependencies, types of 25
domain layers 16–17, 19
domain model 39
  connecting with external applications 21
dummy test double 3–4
dynamic fake objects 47
dynamic mock objects
  creating 49–50
  defined 47
  using NSubstitute 47–48
  using stubs with 51–56
dynamic stubs 44

E

EasyMock 45
end-to-end tests 36
Equals() method 55
ErrorInfo object 54
events
  testing if triggered 57–58
  testing listener 56–57

F

fake dependencies 3
fake test double 3–4
fakes
  creating 60
  overview 50–51
  using mock and stub 51–56
false positives 6, 9

fast feedback 9
Forces method 50
fragile tests 6, 23
frameworks
  .NET 58
  advantages of 60
  avoiding misuse of
    more than one mock per test 61
    overspecifying tests 61–62
    unreadable test code 60
    verifying wrong things 60
  dynamic mock objects
    creating 49–50
    defined 47
    using NSubstitute 47–48
  events
    testing if triggered 57–58
    testing listener 56–57
  overview 44–45
  purpose of 45–46
  simulating fake values
    overview 50–51
    using mock and stub 51–56

H

handwritten mocks 4, 36
hexagonal architecture 16–17
  defining 16–20
  purpose of 17
hexagons 16, 18

I

IFileNameRules interface 50
ILogger interface 48–49
implementation details 9–15
incoming interactions 4–5
internal keyword 9
invariant violations 13
invariants 10, 13
isolation frameworks
  advantages of 60
  avoiding misuse of
    more than one mock per test 61
    overspecifying tests 61–62
    unreadable test code 60
    verifying wrong things 60
  dynamic mock objects
    creating 49–50
    defined 47
    using NSubstitute 47–48
  events
    testing if triggered 57–58
    testing listener 56–57
  for .NET 58
  overview 44–45
  purpose of 45–46
  simulating fake values
    overview 50–51
    using mock and stub 51–56
isSuccess flag 23

J

JustMock 45

L

LogError() method 49
London school of unit testing
  mocks 24–26
    mocking out out-of-process
      dependencies 25–26
    using mocks to verify behavior 26

M

MailSender class 52
maintainability 9
message bus 34, 38
methods, verifying 60
mock objects
  creating 49–50
  defined 47
  using NSubstitute 47–48
  using one per test 61
  using stubs with 51–56
mocking frameworks
  .NET 58
  advantages of 60
  avoiding misuse of
    more than one mock per test 61
    overspecifying tests 61–62
    unreadable test code 60
    verifying wrong things 60
  dynamic mock objects
    creating 49–50
    defined 47
    using NSubstitute 47–48
  events
    testing if triggered 57–58
    testing listener 56–57
  overview 44–45

purpose of 45–46
simulating fake values
overview 50–51
using mock and stub 51–56
mocks
best practices 39–41
for integration tests only 39
not just one mock per test 39–40
only mock types that you own 41
verifying number of calls 40
London school vs. classical school 24–26
mocking out out-of-process dependencies 25–26
using mocks to verify behavior 26
maximizing value of 31–39
IDomainLogger 38–39
replacing mocks with spies 36–38
verifying interactions at system edges 33–36
observable behavior vs. implementation details 9–15
leaking implementation details 10–15
observable behavior vs. public API 9–10
well-designed API and encapsulation 13–14
stubs 3–8
asserting interactions with stubs 6–7
commands and queries 7–8
mock (tool) vs. mock (test double) 4–5
types of test doubles 3–4
using mocks and stubs together 7
test fragility 16–24
defining hexagonal architecture 16–20
intra-system vs. inter-system communications 20–24
Moles 45
Moq 5, 40, 45, 58

N

.NET, isolation frameworks for 58
NMock 45
nonstrict mocks 61
NSubstitute, overview 47–48
NuGet 47
NUnit.Mocks 45

O

object-relational mapping (ORM) 41
observable behavior 9, 15, 18, 25
leaking implementation details 10–15
public API 9–10
well-designed API and encapsulation 13–14
operations 9, 14
ORM (object-relational mapping) 41
outcoming interactions 4–5
out-of-process dependencies 25
overspecification 6
overspecification, avoiding, in tests 61–62

P

parameter verification 60
private APIs 9
private dependencies 25
private keyword 9
protection against regressions 9
Public API 9, 19

Q

queries 7

R

Received() method 49
ReSharper 59
resistance to refactoring 2–3, 9
Rhino Mocks 45, 58

S

shared dependencies 25
simulating fake values
overview 50–51
using mock and stub 51–56
SMTP service 20, 22–25
spies 4, 36–38
spy test double 3
state 9, 11
strict mocks 61
stubs, using mock objects with 51–56
stubs, mocks 3–8
asserting interactions with stubs 6–7
commands and queries 7–8
mock (tool) vs. mock (test double) 4–5
types of test doubles 3–4
using mocks and stubs together 7
sub-renderers collection 15
Substitute class 47
SUT (system under test) 3–4, 6–7
system leaks 10

T

tell-don't-ask principle 14
test doubles 3–4, 8
test fragility, mocks and 16–24
  defining hexagonal architecture 16–20
  intra-system vs. inter-system communications 20–24
test isolation 25
third-party applications 22
Typemock Isolator 45

U

unit of behavior 39
units of code 39
unmanaged dependencies 30, 32, 34, 36, 40

V

values, fake
  overview 50–51
  using mock and stub 51–56
verifyAll() method 50
void type 7

W

WebService class 52
Write() method 54