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).

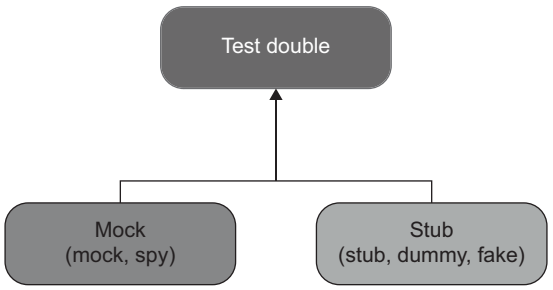

Figure 5.1 All variations of test doubles can be categorized into two types: mocks and stubs.

[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).

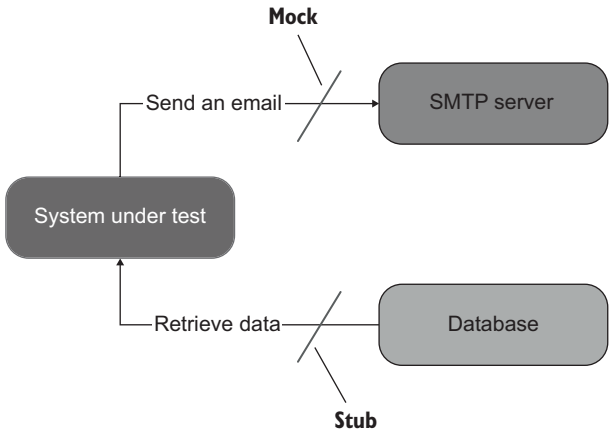

Figure 5.2 Sending an email is an *outcoming* interaction: an interaction that results in a side effect in the SMTP server. A test double emulating such an interaction is a *mock*. Retrieving data from the database is an *incoming* interaction; it doesn't result in a side effect. The corresponding test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

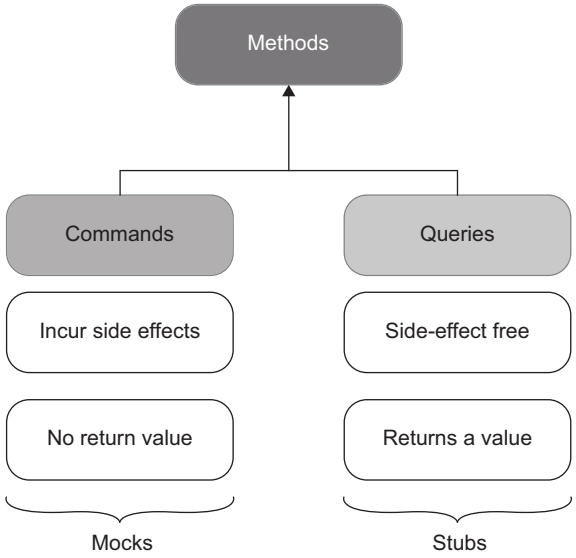

Figure 5.3 In the command query separation (CQS) principle, commands correspond to mocks, while queries are consistent with stubs.

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).

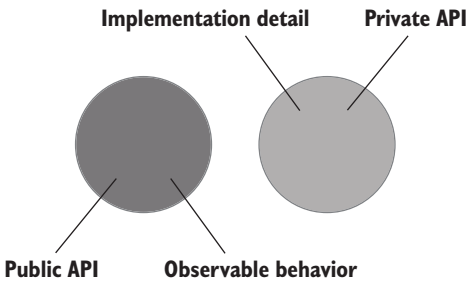

Figure 5.4 In a *well-designed* API, the observable behavior coincides with the public API, while all implementation details are hidden behind the private API.

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).

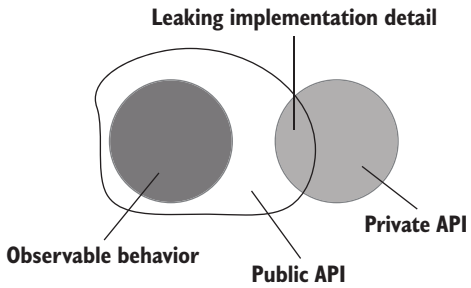

Figure 5.5 A system leaks implementation details when its public API extends beyond the observable behavior.

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.

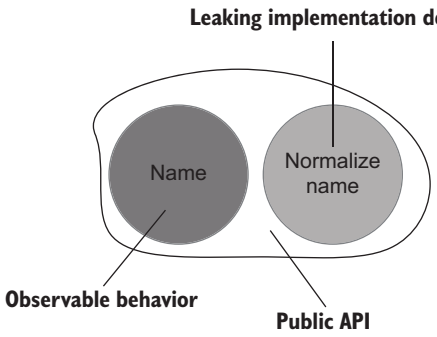

Figure 5.6 The API of `User` is not well-designed: it exposes the `NormalizeName` method, which is not part of the observable behavior.

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).

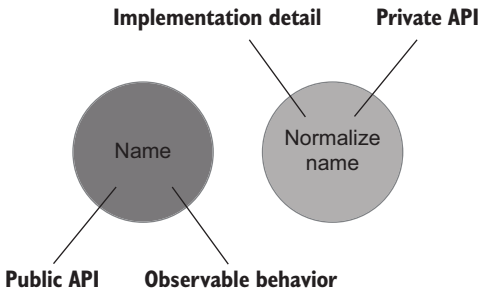

Figure 5.7 `User` **with a well-designed API. Only the observable behavior is public; the implementation details are now private.**

> **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.

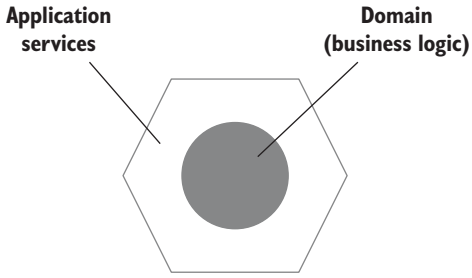

Figure 5.8 A typical application consists of a domain layer and an application services layer. The domain layer contains the application's business logic; application services tie that logic to business use cases.

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*.

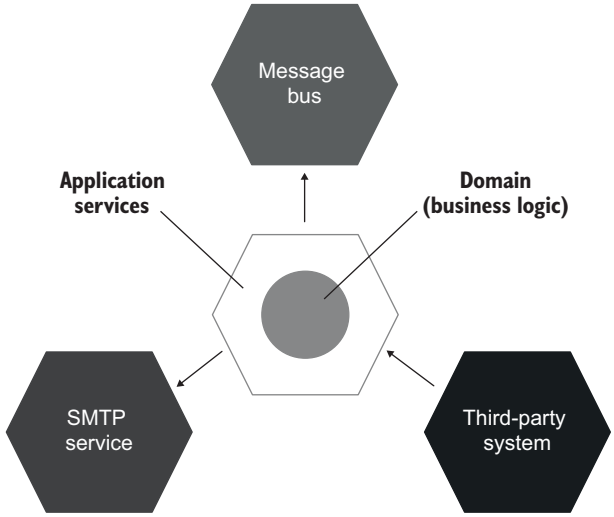

Figure 5.9 A hexagonal architecture is a set of interacting applications—hexagons.

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).

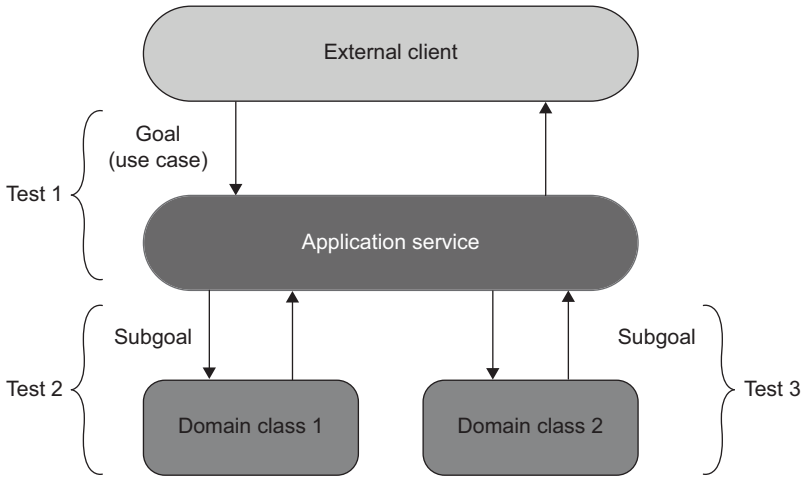

Figure 5.10 Tests working with different layers have a fractal nature: they verify the same behavior at different levels. A test of an application service checks to see how the overall business use case is executed. A test working with a domain class verifies an intermediate subgoal on the way to use-case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

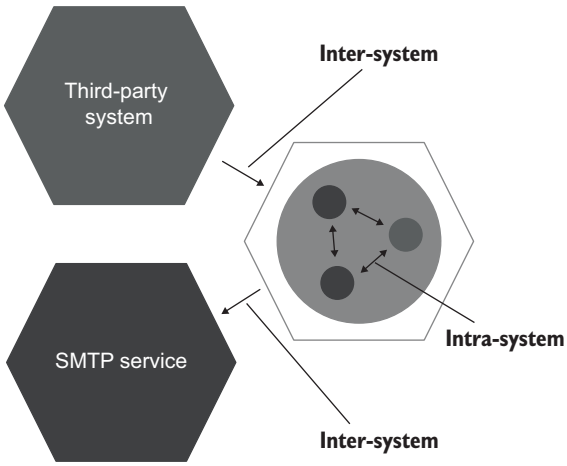

Figure 5.11 There are two types of communications: intra-system (between classes inside the application) and inter-system (between applications).

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.

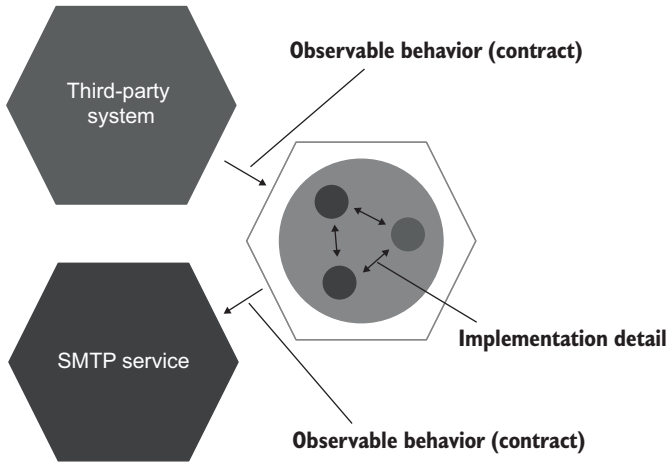

Figure 5.12 Inter-system communications form the observable behavior of your application as a whole. Intra-system communications are implementation details.

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.

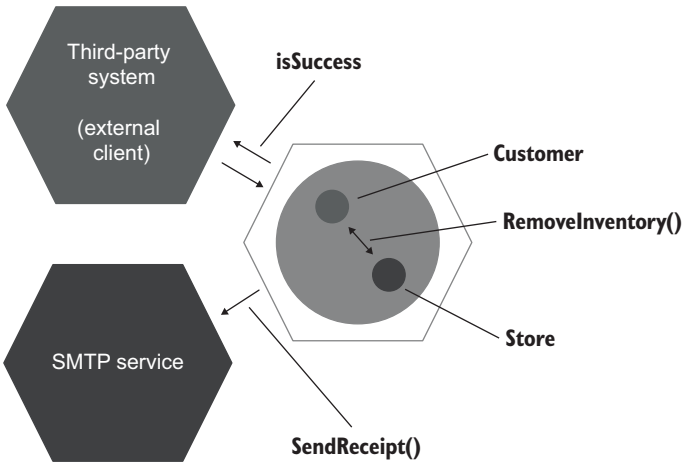

Figure 5.13 The example in listing 5.9 represented using the hexagonal architecture. The communications between the hexagons are inter-system communications. The communication inside the hexagon is intra-system.

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

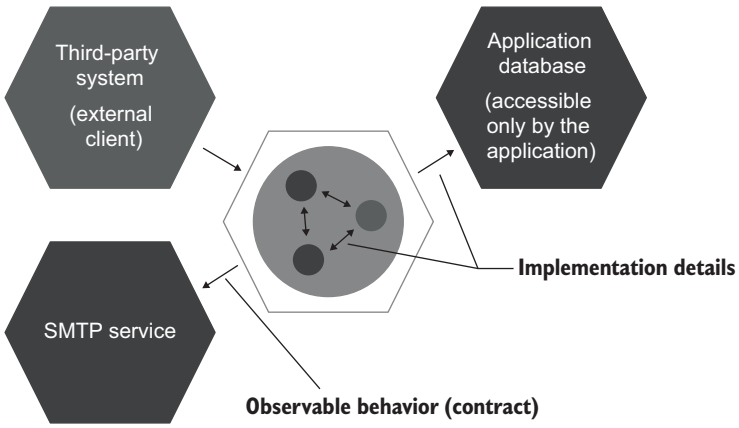

Figure 5.14 Communications with an out-of-process dependency that can't be observed externally are implementation details. They don't have to stay in place after refactoring and therefore shouldn't be verified with mocks.

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.