2 What is a unit test?

This chapter covers

- What a unit test is
- The differences between shared, private, and volatile dependencies
- The two schools of unit testing: classical and London
- The differences between unit, integration, and end-to-end tests

As mentioned in chapter 1, there are a surprising number of nuances in the definition of a unit test. Those nuances are more important than you might think—so much so that the differences in interpreting them have led to two distinct views on how to approach unit testing.

These views are known as the *classical* and the *London* schools of unit testing. The classical school is called "classical" because it's how everyone originally approached unit testing and test-driven development. The London school takes root in the programming community in London. The discussion in this chapter about the differences between the classical and London styles lays the foundation for chapter 5, where I cover the topic of mocks and test fragility in detail.

Let's start by defining a unit test, with all due caveats and subtleties. This definition is the key to the difference between the classical and London schools.

2.1 The definition of "unit test"

There are a lot of definitions of a unit test. Stripped of their non-essential bits, the definitions all have the following three most important attributes. A unit test is an automated test that

- Verifies a small piece of code (also known as a *unit*),
- Does it quickly,
- And does it in an isolated manner.

The first two attributes here are pretty non-controversial. There might be some dispute as to what exactly constitutes a fast unit test because it's a highly subjective measure. But overall, it's not that important. If your test suite's execution time is good enough for you, it means your tests are quick enough.

What people have vastly different opinions about is the third attribute. The isolation issue is the root of the differences between the classical and London schools of unit testing. As you will see in the next section, all other differences between the two schools flow naturally from this single disagreement on what exactly *isolation* means. I prefer the classical style for the reasons I describe in section 2.3.

The classical and London schools of unit testing

The classical approach is also referred to as the *Detroit* and, sometimes, the *classicist* approach to unit testing. Probably the most canonical book on the classical school is the one by Kent Beck: *Test-Driven Development: By Example* (Addison-Wesley Professional, 2002).

The London style is sometimes referred to as *mockist*. Although the term *mockist* is widespread, people who adhere to this style of unit testing generally don't like it, so I call it the London style throughout this book. The most prominent proponents of this approach are Steve Freeman and Nat Pryce. I recommend their book, *Growing Object-Oriented Software, Guided by Tests* (Addison-Wesley Professional, 2009), as a good source on this subject.

2.1.1 The isolation issue: The London take

What does it mean to verify a piece of code—a unit—in an isolated manner? The London school describes it as isolating the system under test from its collaborators. It means if a class has a dependency on another class, or several classes, you need to replace all such dependencies with test doubles. This way, you can focus on the class under test exclusively by separating its behavior from any external influence.

DEFINITION A *test double* is an object that looks and behaves like its release-intended counterpart but is actually a simplified version that reduces the complexity and facilitates testing. This term was introduced by Gerard Meszaros in his book, *xUnit Test Patterns: Refactoring Test Code* (Addison-Wesley, 2007). The name itself comes from the notion of a stunt double in movies.

Figure 2.1 shows how the isolation is usually achieved. A unit test that would otherwise verify the system under test along with all its dependencies now can do that separately from those dependencies.

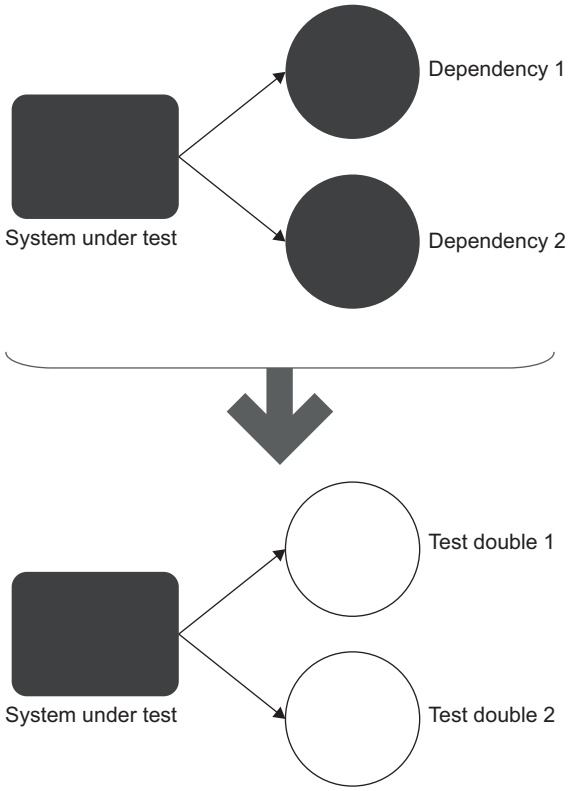

Figure 2.1 Replacing the dependencies of the system under test with test doubles allows you to focus on verifying the system under test exclusively, as well as split the otherwise large interconnected object graph.

One benefit of this approach is that if the test fails, you know for sure which part of the code base is broken: it's the system under test. There could be no other suspects, because all of the class's neighbors are replaced with the test doubles.

Another benefit is the ability to split the *object graph*—the web of communicating classes solving the same problem. This web may become quite complicated: every class in it may have several immediate dependencies, each of which relies on dependencies of their own, and so on. Classes may even introduce circular dependencies, where the chain of dependency eventually comes back to where it started.

Trying to test such an interconnected code base is hard without test doubles. Pretty much the only choice you are left with is re-creating the full object graph in the test, which might not be a feasible task if the number of classes in it is too high.

With test doubles, you can put a stop to this. You can substitute the immediate dependencies of a class; and, by extension, you don't have to deal with the dependencies of those dependencies, and so on down the recursion path. You are effectively breaking up the graph—and that can significantly reduce the amount of preparations you have to do in a unit test.

And let's not forget another small but pleasant side benefit of this approach to unit test isolation: it allows you to introduce a project-wide guideline of testing only one class at a time, which establishes a simple structure in the whole unit test suite. You no longer have to think much about how to cover your code base with tests. Have a class? Create a corresponding class with unit tests! Figure 2.2 shows how it usually looks.

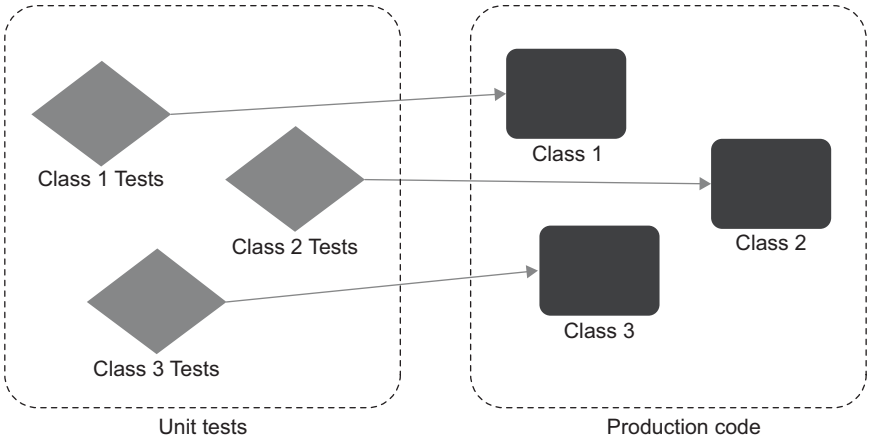

Figure 2.2 Isolating the class under test from its dependencies helps establish a simple test suite structure: one class with tests for each class in the production code.

Let's now look at some examples. Since the classical style probably looks more familiar to most people, I'll show sample tests written in that style first and then rewrite them using the London approach.

Let's say that we operate an online store. There's just one simple use case in our sample application: a customer can purchase a product. When there's enough inventory in the store, the purchase is deemed to be successful, and the amount of the product in the store is reduced by the purchase's amount. If there's not enough product, the purchase is not successful, and nothing happens in the store.

Listing 2.1 shows two tests verifying that a purchase succeeds only when there's enough inventory in the store. The tests are written in the classical style and use the

typical three-phase sequence: arrange, act, and assert (AAA for short—I talk more about this sequence in chapter 3).

Listing 2.1 Tests written using the classical style of unit testing

```
[Fact]
public void Purchase_succeeds_when_enough_inventory()
{
    // Arrange
    var store = new Store();
    store.AddInventory(Product.Shampoo, 10);
    var customer = new Customer();

    // Act
    bool success = customer.Purchase(store, Product.Shampoo, 5);

    // Assert
    Assert.True(success);
    Assert.Equal(5, store.GetInventory(Product.Shampoo));
}

[Fact]
public void Purchase_fails_when_not_enough_inventory()
{
    // Arrange
    var store = new Store();
    store.AddInventory(Product.Shampoo, 10);
    var customer = new Customer();

    // Act
    bool success = customer.Purchase(store, Product.Shampoo, 15);

    // Assert
    Assert.False(success);
    Assert.Equal(10, store.GetInventory(Product.Shampoo));
}

public enum Product
{
    Shampoo,
    Book
}
```

Reduces the product amount in the store by five

The product amount in the store remains unchanged.

As you can see, the arrange part is where the tests make ready all dependencies and the system under test. The call to `customer.Purchase()` is the act phase, where you exercise the behavior you want to verify. The assert statements are the verification stage, where you check to see if the behavior led to the expected results.

During the arrange phase, the tests put together two kinds of objects: the system under test (SUT) and one collaborator. In this case, `Customer` is the SUT and `Store` is the collaborator. We need the collaborator for two reasons:

- To get the method under test to compile, because `customer.Purchase()` requires a `Store` instance as an argument
- For the assertion phase, since one of the results of `customer.Purchase()` is a potential decrease in the product amount in the store

`Product.Shampoo` and the numbers `5` and `15` are constants.

DEFINITION A *method under test* (*MUT*) is a method in the SUT called by the test. The terms *MUT* and *SUT* are often used as synonyms, but normally, *MUT* refers to a method while *SUT* refers to the whole class.

This code is an example of the classical style of unit testing: the test doesn't replace the collaborator (the `Store` class) but rather uses a production-ready instance of it. One of the natural outcomes of this style is that the test now effectively verifies both `Customer` and `Store`, not just `Customer`. Any bug in the inner workings of `Store` that affects `Customer` will lead to failing these unit tests, even if `Customer` still works correctly. The two classes are not isolated from each other in the tests.

Let's now modify the example toward the London style. I'll take the same tests and replace the `Store` instances with test doubles—specifically, mocks.

I use Moq (https://github.com/moq/moq4) as the mocking framework, but you can find several equally good alternatives, such as NSubstitute (https://github.com/nsubstitute/NSubstitute). All object-oriented languages have analogous frameworks. For instance, in the Java world, you can use Mockito, JMock, or EasyMock.

DEFINITION A *mock* is a special kind of test double that allows you to examine interactions between the system under test and its collaborators.

We'll get back to the topic of mocks, stubs, and the differences between them in later chapters. For now, the main thing to remember is that mocks are a subset of test doubles. People often use the terms *test double* and *mock* as synonyms, but technically, they are not (more on this in chapter 5):

- *Test double* is an overarching term that describes all kinds of non-production-ready, fake dependencies in a test.
- *Mock* is just one kind of such dependencies.

The next listing shows how the tests look after isolating `Customer` from its collaborator, `Store`.

Listing 2.2 Tests written using the London style of unit testing

```
[Fact]
public void Purchase_succeeds_when_enough_inventory()
{
    // Arrange
    var storeMock = new Mock<IStore>();
    storeMock
```

```
        .Setup(x => x.HasEnoughInventory(Product.Shampoo, 5))
        .Returns(true);
    var customer = new Customer();

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

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

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

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

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

Note how different these tests are from those written in the classical style. In the arrange phase, the tests no longer instantiate a production-ready instance of `Store` but instead create a substitution for it, using Moq's built-in class `Mock<T>`.

Furthermore, instead of modifying the state of `Store` by adding a shampoo inventory to it, we directly tell the mock how to respond to calls to `HasEnoughInventory()`. The mock reacts to this request the way the tests need, regardless of the actual state of `Store`. In fact, the tests no longer use `Store`—we have introduced an `IStore` interface and are mocking that interface instead of the `Store` class.

In chapter 8, I write in detail about working with interfaces. For now, just make a note that interfaces are required for isolating the system under test from its collaborators. (You can also mock a concrete class, but that's an anti-pattern; I cover this topic in chapter 11.)

The assertion phase has changed too, and that's where the key difference lies. We still check the output from `customer.Purchase` as before, but the way we verify that the customer did the right thing to the store is different. Previously, we did that by asserting against the store's state. Now, we examine the interactions between `Customer` and `Store`: the tests check to see if the customer made the correct call on the store. We do this by passing the method the customer should call on the store (`x.RemoveInventory`) as well as the number of times it should do that. If the purchases succeeds, the customer should call this method once (`Times.Once`). If the purchases fails, the customer shouldn't call it at all (`Times.Never`).

2.1.2 The isolation issue: The classical take

To reiterate, the London style approaches the isolation requirement by segregating the piece of code under test from its collaborators with the help of test doubles: specifically, mocks. Interestingly enough, this point of view also affects your standpoint on what constitutes a small piece of code (a unit). Here are all the attributes of a unit test once again:

- A unit test verifies a small piece of code (a unit),
- Does it quickly,
- And does it in an isolated manner.

In addition to the third attribute leaving room for interpretation, there's some room in the possible interpretations of the first attribute as well. How small should a small piece of code be? As you saw from the previous section, if you adopt the position of isolating every individual class, then it's natural to accept that the piece of code under test should also be a single class, or a method inside that class. It can't be more than that due to the way you approach the isolation issue. In some cases, you might test a couple of classes at once; but in general, you'll always strive to maintain this guideline of unit testing one class at a time.

As I mentioned earlier, there's another way to interpret the isolation attribute—the classical way. In the classical approach, it's not the code that needs to be tested in an isolated manner. Instead, unit tests themselves should be run in isolation from each other. That way, you can run the tests in parallel, sequentially, and in any order, whatever fits you best, and they still won't affect each other's outcome.

Isolating tests from each other means it's fine to exercise several classes at once as long as they all reside in the memory and don't reach out to a shared state, through which the tests can communicate and affect each other's execution context. Typical examples of such a shared state are out-of-process dependencies—the database, the file system, and so on.

For instance, one test could create a customer in the database as part of its arrange phase, and another test would delete it as part of its own arrange phase, before the first test completes executing. If you run these two tests in parallel, the first test will fail, not because the production code is broken, but rather because of the interference from the second test.

Shared, private, and out-of-process dependencies

A *shared dependency* is a dependency that is shared between tests and provides means for those tests to affect each other's outcome. A typical example of *shared dependencies* is a static mutable field. A change to such a field is visible across all unit tests running within the same process. A database is another typical example of a *shared dependency*.

A *private dependency* is a dependency that is not shared.

An *out-of-process dependency* is a dependency that runs outside the application's execution process; it's a proxy to data that is not yet in the memory. An *out-of-process dependency* corresponds to a shared dependency in the vast majority of cases, but not always. For example, a database is both out-of-process and shared. But if you launch that database in a Docker container before each test run, that would make this dependency out-of-process but not shared, since tests no longer work with the same instance of it. Similarly, a read-only database is also out-of-process but not shared, even if it's reused by tests. Tests can't mutate data in such a database and thus can't affect each other's outcome.

This take on the isolation issue entails a much more modest view on the use of mocks and other test doubles. You can still use them, but you normally do that for only those dependencies that introduce a shared state between tests. Figure 2.3 shows how it looks.

Note that shared dependencies are shared *between unit tests*, not between classes under test (units). In that sense, a singleton dependency is not shared as long as you are able to create a new instance of it in each test. While there's only one instance of a

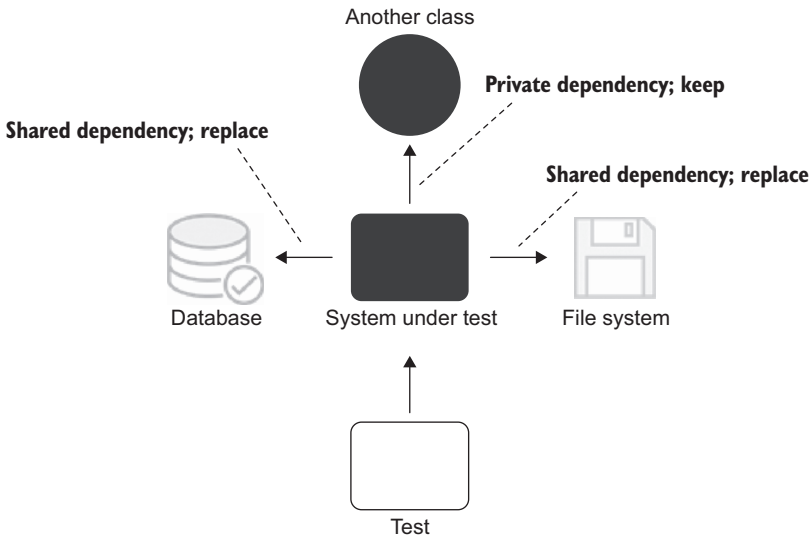

Figure 2.3 Isolating unit tests from each other entails isolating the class under test from shared dependencies only. Private dependencies can be kept intact.

singleton in the production code, tests may very well not follow this pattern and not reuse that singleton. Thus, such a dependency would be private.

For example, there's normally only one instance of a configuration class, which is reused across all production code. But if it's injected into the SUT the way all other dependencies are, say, via a constructor, you can create a new instance of it in each test; you don't have to maintain a single instance throughout the test suite. You can't create a new file system or a database, however; they must be either shared between tests or substituted away with test doubles.

Shared vs. volatile dependencies

Another term has a similar, yet not identical, meaning: *volatile dependency*. I recommend *Dependency Injection: Principles, Practices, Patterns* by Steven van Deursen and Mark Seemann (Manning Publications, 2018) as a go-to book on the topic of dependency management.

A *volatile dependency* is a dependency that exhibits one of the following properties:

- *It introduces a requirement to set up and configure a runtime environment in addition to what is installed on a developer's machine by default*. Databases and API services are good examples here. They require additional setup and are not installed on machines in your organization by default.
- *It contains nondeterministic behavior*. An example would be a random number generator or a class returning the current date and time. These dependencies are non-deterministic because they provide different results on each invocation.

As you can see, there's an overlap between the notions of *shared* and *volatile* dependencies. For example, a dependency on the database is both shared and volatile. But that's not the case for the file system. The file system is not volatile because it is installed on every developer's machine and it behaves deterministically in the vast majority of cases. Still, the file system introduces a means by which the unit tests can interfere with each other's execution context; hence it is shared. Likewise, a random number generator is volatile, but because you can supply a separate instance of it to each test, it isn't shared.

Another reason for substituting shared dependencies is to increase the test execution speed. Shared dependencies almost always reside outside the execution process, while private dependencies usually don't cross that boundary. Because of that, calls to shared dependencies, such as a database or the file system, take more time than calls to private dependencies. And since the necessity to run quickly is the second attribute of the unit test definition, such calls push the tests with shared dependencies out of the realm of unit testing and into the area of integration testing. I talk more about integration testing later in this chapter.

This alternative view of isolation also leads to a different take on what constitutes a unit (a small piece of code). A unit doesn't necessarily have to be limited to a class.

You can just as well unit test a group of classes, as long as none of them is a shared dependency.

2.2 The classical and London schools of unit testing

As you can see, the root of the differences between the London and classical schools is the isolation attribute. The London school views it as isolation of the system under test from its collaborators, whereas the classical school views it as isolation of unit tests themselves from each other.

This seemingly minor difference has led to a vast disagreement about how to approach unit testing, which, as you already know, produced the two schools of thought. Overall, the disagreement between the schools spans three major topics:

- The isolation requirement
- What constitutes a piece of code under test (a unit)
- Handling dependencies

Table 2.1 sums it all up.

Table 2.1 The differences between the London and classical schools of unit testing, summed up by the approach to isolation, the size of a unit, and the use of test doubles

	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

2.2.1 How the classical and London schools handle dependencies

Note that despite the ubiquitous use of test doubles, the London school still allows for using some dependencies in tests as-is. The litmus test here is whether a dependency is mutable. It's fine not to substitute objects that don't ever change—*immutable* objects.

And you saw in the earlier examples that, when I refactored the tests toward the London style, I didn't replace the `Product` instances with mocks but rather used the real objects, as shown in the following code (repeated from listing 2.2 for your convenience):

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

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

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

Of the two dependencies of `Customer`, only `Store` contains an internal state that can change over time. The `Product` instances are immutable (`Product` itself is a C# enum). Hence I substituted the `Store` instance only.

It makes sense, if you think about it. You wouldn't use a test double for the `5` number in the previous test either, would you? That's because it is also immutable—you can't possibly modify this number. Note that I'm not talking about a variable containing the number, but rather the number itself. In the statement `RemoveInventory(Product.Shampoo, 5)`, we don't even use a variable; `5` is declared right away. The same is true for `Product.Shampoo`.

Such immutable objects are called *value objects* or *values*. Their main trait is that they have no individual identity; they are identified solely by their content. As a corollary, if two such objects have the same content, it doesn't matter which of them you're working with: these instances are interchangeable. For example, if you've got two `5` integers, you can use them in place of one another. The same is true for the products in our case: you can reuse a single `Product.Shampoo` instance or declare several of them—it won't make any difference. These instances will have the same content and thus can be used interchangeably.

Note that the concept of a *value object* is language-agnostic and doesn't require a particular programming language or framework. You can read more about value objects in my article "Entity vs. Value Object: The ultimate list of differences" at http://mng.bz/KE9O.

Figure 2.4 shows the categorization of dependencies and how both schools of unit testing treat them. A *dependency* can be either *shared* or *private*. A *private* dependency, in turn, can be either *mutable* or *immutable*. In the latter case, it is called a *value object*. For example, a database is a shared dependency—its internal state is shared across all automated tests (that don't replace it with a test double). A `Store` instance is a private dependency that is mutable. And a `Product` instance (or an instance of a number `5`, for that matter) is an example of a private dependency that is immutable—a value object. All shared dependencies are mutable, but for a mutable dependency to be shared, it has to be reused by tests.

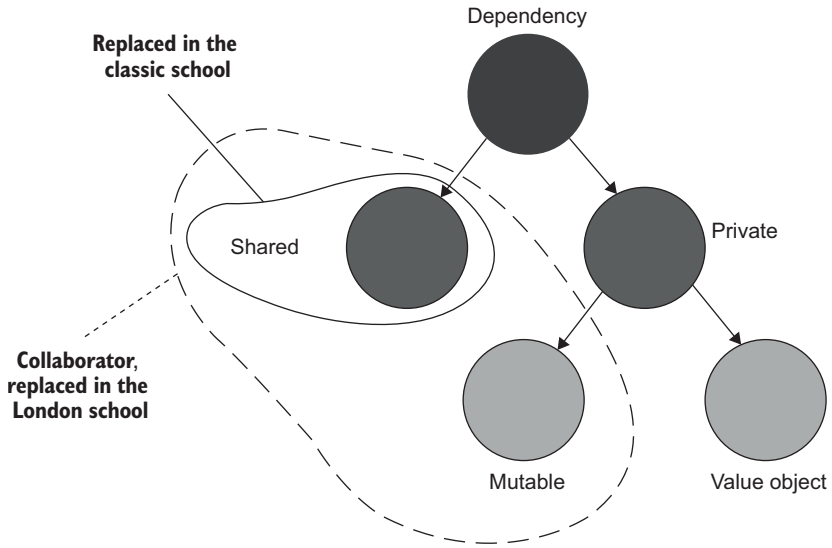

Figure 2.4 The hierarchy of dependencies. The classical school advocates for replacing shared dependencies with test doubles. The London school advocates for the replacement of private dependencies as well, as long as they are mutable.

I'm repeating table 2.1 with the differences between the schools for your convenience.

	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

Collaborator vs. dependency

A *collaborator* is a dependency that is either shared or mutable. For example, a class providing access to the database is a collaborator since the database is a shared dependency. `Store` is a collaborator too, because its state can change over time.

`Product` and number `5` are also dependencies, but they're not collaborators. They're *values* or *value objects*.

A typical class may work with dependencies of both types: *collaborators* and *values*. Look at this method call:

```
customer.Purchase(store, Product.Shampoo, 5)
```

Here we have three dependencies. One of them (`store`) is a collaborator, and the other two (`Product.Shampoo`, `5`) are not.

And let me reiterate one point about the types of dependencies. Not all out-of-process dependencies fall into the category of shared dependencies. A shared dependency almost always resides outside the application's process, but the opposite isn't true (see figure 2.5). In order for an out-of-process dependency to be shared, it has to provide means for unit tests to communicate with each other. The communication is done through modifications of the dependency's internal state. In that sense, an immutable out-of-process dependency doesn't provide such a means. The tests simply can't modify anything in it and thus can't interfere with each other's execution context.

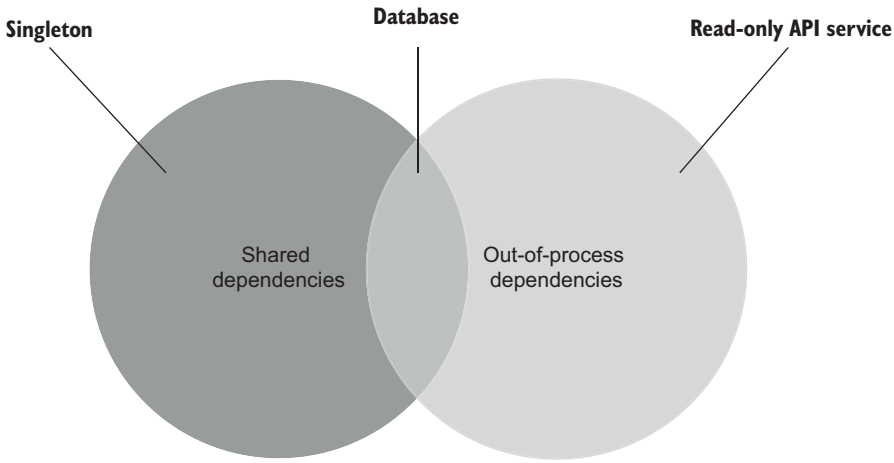

Figure 2.5 The relation between shared and out-of-process dependencies. An example of a dependency that is shared but not out-of-process is a singleton (an instance that is reused by all tests) or a static field in a class. A database is shared and out-of-process—it resides outside the main process and is mutable. A read-only API is out-of-process but not shared, since tests can't modify it and thus can't affect each other's execution flow.

For example, if there's an API somewhere that returns a catalog of all products the organization sells, this isn't a shared dependency as long as the API doesn't expose the functionality to change the catalog. It's true that such a dependency is volatile and sits outside the application's boundary, but since the tests can't affect the data it returns, it isn't shared. This doesn't mean you have to include such a dependency in the testing scope. In most cases, you still need to replace it with a test double to keep the test fast. But if the out-of-process dependency is quick enough and the connection to it is stable, you can make a good case for using it as-is in the tests.

Having that said, in this book, I use the terms *shared dependency* and *out-of-process dependency* interchangeably unless I explicitly state otherwise. In real-world projects, you rarely have a shared dependency that isn't out-of-process. If a dependency is in-process, you can easily supply a separate instance of it to each test; there's no need to share it between tests. Similarly, you normally don't encounter an out-of-process

dependency that's not shared. Most such dependencies are mutable and thus can be modified by tests.

With this foundation of definitions, let's contrast the two schools on their merits.

2.3 Contrasting the classical and London schools of unit testing

To reiterate, the main difference between the classical and London schools is in how they treat the isolation issue in the definition of a unit test. This, in turn, spills over to the treatment of a *unit*—the thing that should be put under test—and the approach to handling dependencies.

As I mentioned previously, I prefer the classical school of unit testing. It tends to produce tests of higher quality and thus is better suited for achieving the ultimate goal of unit testing, which is the sustainable growth of your project. The reason is fragility: tests that use mocks tend to be more brittle than classical tests (more on this in chapter 5). For now, let's take the main selling points of the London school and evaluate them one by one.

The London school's approach provides the following benefits:

- *Better granularity*. The tests are fine-grained and check only one class at a time.
- *It's easier to unit test a larger graph of interconnected classes*. Since all collaborators are replaced by test doubles, you don't need to worry about them at the time of writing the test.
- *If a test fails, you know for sure which functionality has failed*. Without the class's collaborators, there could be no suspects other than the class under test itself. Of course, there may still be situations where the system under test uses a value object and it's the change in this value object that makes the test fail. But these cases aren't that frequent because all other dependencies are eliminated in tests.

2.3.1 Unit testing one class at a time

The point about better granularity relates to the discussion about what constitutes a *unit* in unit testing. The London school considers a class as such a unit. Coming from an object-oriented programming background, developers usually regard classes as the atomic building blocks that lie at the foundation of every code base. This naturally leads to treating classes as the atomic units to be verified in tests, too. This tendency is understandable but misleading.

> **TIP** Tests shouldn't verify *units of code*. Rather, they should verify *units of behavior*: something that is meaningful for the problem domain and, ideally, something that a business person can recognize as useful. The number of classes it takes to implement such a unit of behavior is irrelevant. The unit could span across multiple classes or only one class, or even take up just a tiny method.

And so, aiming at better code granularity isn't helpful. As long as the test checks a single unit of behavior, it's a good test. Targeting something less than that can in fact damage your unit tests, as it becomes harder to understand exactly what these tests verify. *A test should tell a story about the problem your code helps to solve, and this story should be cohesive and meaningful to a non-programmer.*

For instance, this is an example of a cohesive story:

```
When I call my dog, he comes right to me.
```

Now compare it to the following:

```
When I call my dog, he moves his front left leg first, then the front right
leg, his head turns, the tail start wagging...
```

The second story makes much less sense. What's the purpose of all those movements? Is the dog coming to me? Or is he running away? You can't tell. This is what your tests start to look like when you target individual classes (the dog's legs, head, and tail) instead of the actual behavior (the dog coming to his master). I talk more about this topic of observable behavior and how to differentiate it from internal implementation details in chapter 5.

2.3.2 Unit testing a large graph of interconnected classes

The use of mocks in place of real collaborators can make it easier to test a class—especially when there's a complicated dependency graph, where the class under test has dependencies, each of which relies on dependencies of its own, and so on, several layers deep. With test doubles, you can substitute the class's immediate dependencies and thus break up the graph, which can significantly reduce the amount of preparation you have to do in a unit test. If you follow the classical school, you have to re-create the full object graph (with the exception of shared dependencies) just for the sake of setting up the system under test, which can be a lot of work.

Although this is all true, this line of reasoning focuses on the wrong problem. Instead of finding ways to test a large, complicated graph of interconnected classes, you should focus on not having such a graph of classes in the first place. More often than not, a large class graph is a result of a code design problem.

It's actually a good thing that the tests point out this problem. As we discussed in chapter 1, the ability to unit test a piece of code is a good negative indicator—it predicts poor code quality with a relatively high precision. If you see that to unit test a class, you need to extend the test's arrange phase beyond all reasonable limits, it's a certain sign of trouble. The use of mocks only hides this problem; it doesn't tackle the root cause. I talk about how to fix the underlying code design problem in part 2.

2.3.3 Revealing the precise bug location

If you introduce a bug to a system with London-style tests, it normally causes only tests whose SUT contains the bug to fail. However, with the classical approach, tests that target the clients of the malfunctioning class can also fail. This leads to a ripple effect where a single bug can cause test failures across the whole system. As a result, it becomes harder to find the root of the issue. You might need to spend some time debugging the tests to figure it out.

It's a valid concern, but I don't see it as a big problem. If you run your tests regularly (ideally, after each source code change), then you know what caused the bug—it's what you edited last, so it's not that difficult to find the issue. Also, you don't have to look at all the failing tests. Fixing one automatically fixes all the others.

Furthermore, there's some value in failures cascading all over the test suite. If a bug leads to a fault in not only one test but a whole lot of them, it shows that the piece of code you have just broken is of great value—the entire system depends on it. That's useful information to keep in mind when working with the code.

2.3.4 Other differences between the classical and London schools

Two remaining differences between the classical and London schools are

- Their approach to system design with test-driven development (TDD)
- The issue of over-specification

Test-driven development

Test-driven development is a software development process that relies on tests to drive the project development. The process consists of three (some authors specify four) stages, which you repeat for every test case:

1. Write a failing test to indicate which functionality needs to be added and how it should behave.
2. Write just enough code to make the test pass. At this stage, the code doesn't have to be elegant or clean.
3. Refactor the code. Under the protection of the passing test, you can safely clean up the code to make it more readable and maintainable.

Good sources on this topic are the two books I recommended earlier: Kent Beck's *Test-Driven Development: By Example*, and *Growing Object-Oriented Software, Guided by Tests* by Steve Freeman and Nat Pryce.

The London style of unit testing leads to outside-in TDD, where you start from the higher-level tests that set expectations for the whole system. By using mocks, you specify which collaborators the system should communicate with to achieve the expected result. You then work your way through the graph of classes until you implement every one of them. Mocks make this design process possible because you can focus on one

class at a time. You can cut off all of the SUT's collaborators when testing it and thus postpone implementing those collaborators to a later time.

The classical school doesn't provide quite the same guidance since you have to deal with the real objects in tests. Instead, you normally use the inside-out approach. In this style, you start from the domain model and then put additional layers on top of it until the software becomes usable by the end user.

But the most crucial distinction between the schools is the issue of over-specification: that is, coupling the tests to the SUT's implementation details. The London style tends to produce tests that couple to the implementation more often than the classical style. And this is the main objection against the ubiquitous use of mocks and the London style in general.

There's much more to the topic of mocking. Starting with chapter 4, I gradually cover everything related to it.

2.4 Integration tests in the two schools

The London and classical schools also diverge in their definition of an *integration test*. This disagreement flows naturally from the difference in their views on the isolation issue.

The London school considers any test that uses a real collaborator object an integration test. Most of the tests written in the classical style would be deemed integration tests by the London school proponents. For an example, see listing 1.4, in which I first introduced the two tests covering the customer purchase functionality. That code is a typical unit test from the classical perspective, but it's an integration test for a follower of the London school.

In this book, I use the classical definitions of both unit and integration testing. Again, a unit test is an automated test that has the following characteristics:

- It verifies a small piece of code,
- Does it quickly,
- And does it in an isolated manner.

Now that I've clarified what the first and third attributes mean, I'll redefine them from the point of view of the classical school. A unit test is a test that

- Verifies a *single unit of behavior*,
- Does it quickly,
- And does it in isolation *from other tests*.

An integration test, then, is a test that doesn't meet one of these criteria. For example, a test that reaches out to a shared dependency—say, a database—can't run in isolation from other tests. A change in the database's state introduced by one test would alter the outcome of all other tests that rely on the same database if run in parallel. You'd have to take additional steps to avoid this interference. In particular, you would have to run such tests sequentially, so that each test would wait its turn to work with the shared dependency.

Similarly, an outreach to an out-of-process dependency makes the test slow. A call to a database adds hundreds of milliseconds, potentially up to a second, of additional execution time. Milliseconds might not seem like a big deal at first, but when your test suite grows large enough, every second counts.

In theory, you could write a slow test that works with in-memory objects only, but it's not that easy to do. Communication between objects inside the same memory space is much less expensive than between separate processes. Even if the test works with hundreds of in-memory objects, the communication with them will still execute faster than a call to a database.

Finally, a test is an integration test when it verifies two or more units of behavior. This is often a result of trying to optimize the test suite's execution speed. When you have two slow tests that follow similar steps but verify different units of behavior, it might make sense to merge them into one: one test checking two similar things runs faster than two more-granular tests. But then again, the two original tests would have been integration tests already (due to them being slow), so this characteristic usually isn't decisive.

An integration test can also verify how two or more modules developed by separate teams work together. This also falls into the third bucket of tests that verify multiple units of behavior at once. But again, because such an integration normally requires an out-of-process dependency, the test will fail to meet all three criteria, not just one.

Integration testing plays a significant part in contributing to software quality by verifying the system as a whole. I write about integration testing in detail in part 3.

2.4.1 End-to-end tests are a subset of integration tests

In short, an *integration test* is a test that verifies that your code works in integration with shared dependencies, out-of-process dependencies, or code developed by other teams in the organization. There's also a separate notion of an *end-to-end test*. End-to-end tests are a subset of integration tests. They, too, check to see how your code works with out-of-process dependencies. The difference between an end-to-end test and an integration test is that end-to-end tests usually include more of such dependencies.

The line is blurred at times, but in general, an integration test works with only one or two out-of-process dependencies. On the other hand, an end-to-end test works with all out-of-process dependencies, or with the vast majority of them. Hence the name *end-to-end*, which means the test verifies the system from the end user's point of view, including all the external applications this system integrates with (see figure 2.6).

People also use such terms as *UI tests* (UI stands for *user interface*), *GUI tests* (GUI is *graphical user interface*), and *functional tests*. The terminology is ill-defined, but in general, these terms are all synonyms.

Let's say your application works with three out-of-process dependencies: a database, the file system, and a payment gateway. A typical integration test would include only the database and file system in scope and use a test double to replace the payment gateway. That's because you have full control over the database and file system,

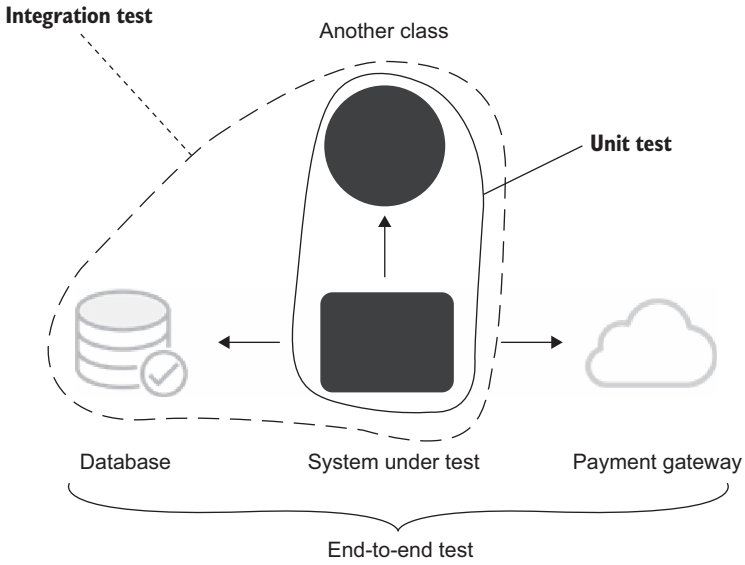

Figure 2.6 End-to-end tests normally include all or almost all out-of-process dependencies in the scope. Integration tests check only one or two such dependencies—those that are easier to set up automatically, such as the database or the file system.

and thus can easily bring them to the required state in tests, whereas you don't have the same degree of control over the payment gateway. With the payment gateway, you may need to contact the payment processor organization to set up a special test account. You might also need to check that account from time to time to manually clean up all the payment charges left over from the past test executions.

Since end-to-end tests are the most expensive in terms of maintenance, it's better to run them late in the build process, after all the unit and integration tests have passed. You may possibly even run them only on the build server, not on individual developers' machines.

Keep in mind that even with end-to-end tests, you might not be able to tackle all of the out-of-process dependencies. There may be no test version of some dependencies, or it may be impossible to bring those dependencies to the required state automatically. So you may still need to use a test double, reinforcing the fact that there isn't a distinct line between integration and end-to-end tests.

Summary

- Throughout this chapter, I've refined the definition of a unit test:
  - A unit test verifies a single unit of behavior,
  - Does it quickly,
  - And does it in isolation from other tests.

- The isolation issue is disputed the most. The dispute led to the formation of two schools of unit testing: the classical (Detroit) school, and the London (mockist) school. This difference of opinion affects the view of what constitutes a unit and the treatment of the system under test's (SUT's) dependencies.
  - The London school states that the units under test should be isolated from each other. A unit under test is a *unit of code*, usually a class. All of its dependencies, except immutable dependencies, should be replaced with test doubles in tests.
  - The classical school states that the *unit tests* need to be isolated from each other, not units. Also, a unit under test is a *unit of behavior*, not a unit of code. Thus, only shared dependencies should be replaced with test doubles. Shared dependencies are dependencies that provide means for tests to affect each other's execution flow.
- The London school provides the benefits of better granularity, the ease of testing large graphs of interconnected classes, and the ease of finding which functionality contains a bug after a test failure.
- The benefits of the London school look appealing at first. However, they introduce several issues. First, the focus on classes under test is misplaced: tests should verify units of behavior, not units of code. Furthermore, the inability to unit test a piece of code is a strong sign of a problem with the code design. The use of test doubles doesn't fix this problem, but rather only hides it. And finally, while the ease of determining which functionality contains a bug after a test failure is helpful, it's not that big a deal because you often know what caused the bug anyway—it's what you edited last.
- The biggest issue with the London school of unit testing is the problem of overspecification—coupling tests to the SUT's implementation details.
- An integration test is a test that doesn't meet at least one of the criteria for a unit test. End-to-end tests are a subset of integration tests; they verify the system from the end user's point of view. End-to-end tests reach out directly to all or almost all out-of-process dependencies your application works with.
- For a canonical book about the classical style, I recommend Kent Beck's *Test-Driven Development: By Example*. For more on the London style, see *Growing Object-Oriented Software, Guided by Tests*, by Steve Freeman and Nat Pryce. For further reading about working with dependencies, I recommend *Dependency Injection: Principles, Practices, Patterns* by Steven van Deursen and Mark Seemann.