6 Styles of unit testing

This chapter covers

- Comparing styles of unit testing
- The relationship between functional and hexagonal architectures
- Transitioning to output-based testing

Chapter 4 introduced the four attributes of a good unit test: protection against regressions, resistance to refactoring, fast feedback, and maintainability. These attributes form a frame of reference that you can use to analyze specific tests and unit testing approaches. We analyzed one such approach in chapter 5: the use of mocks.

In this chapter, I apply the same frame of reference to the topic of unit testing *styles*. There are three such styles: output-based, state-based, and communication-based testing. Among the three, the output-based style produces tests of the highest quality, state-based testing is the second-best choice, and communication-based testing should be used only occasionally.

Unfortunately, you can't use the output-based testing style everywhere. It's only applicable to code written in a purely functional way. But don't worry; there are techniques that can help you transform more of your tests into the output-based style. For that, you'll need to use functional programming principles to restructure the underlying code toward a functional architecture.

Note that this chapter doesn't provide a deep dive into the topic of functional programming. Still, by the end of this chapter, I hope you'll have an intuitive understanding of how functional programming relates to output-based testing. You'll also learn how to write more of your tests using the output-based style, as well as the limitations of functional programming and functional architecture.

6.1 The three styles of unit testing

As I mentioned in the chapter introduction, there are three styles of unit testing:

- Output-based testing
- State-based testing
- Communication-based testing

You can employ one, two, or even all three styles together in a single test. This section lays the foundation for the whole chapter by defining (with examples) those three styles of unit testing. You'll see how they score against each other in the section after that.

6.1.1 Defining the output-based style

The first style of unit testing is the *output-based* style, where you feed an input to the system under test (SUT) and check the output it produces (figure 6.1). This style of unit testing is only applicable to code that doesn't change a global or internal state, so the only component to verify is its return value.

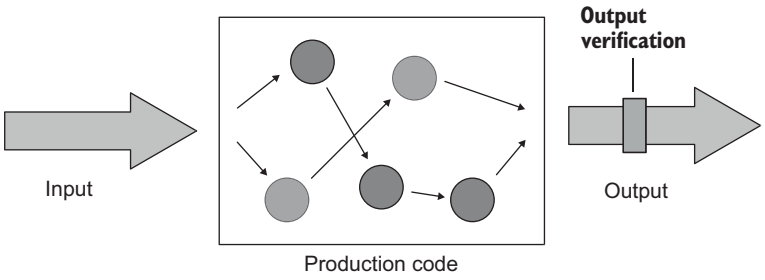

Figure 6.1 In output-based testing, tests verify the output the system generates. This style of testing assumes there are no side effects and the only result of the SUT's work is the value it returns to the caller.

The following listing shows an example of such code and a test covering it. The `PriceEngine` class accepts an array of products and calculates a discount.

Listing 6.1 Output-based testing

```
public class PriceEngine
{
    public decimal CalculateDiscount(params Product[] products)
```

```
    {
        decimal discount = products.Length * 0.01m;
        return Math.Min(discount, 0.2m);
    }
}

[Fact]
public void Discount_of_two_products()
{
    var product1 = new Product("Hand wash");
    var product2 = new Product("Shampoo");
    var sut = new PriceEngine();

    decimal discount = sut.CalculateDiscount(product1, product2);

    Assert.Equal(0.02m, discount);
}
```

`PriceEngine` multiplies the number of products by 1% and caps the result at 20%. There's nothing else to this class. It doesn't add the products to any internal collection, nor does it persist them in a database. The only outcome of the `CalculateDiscount()` method is the discount it returns: the *output* value (figure 6.2).

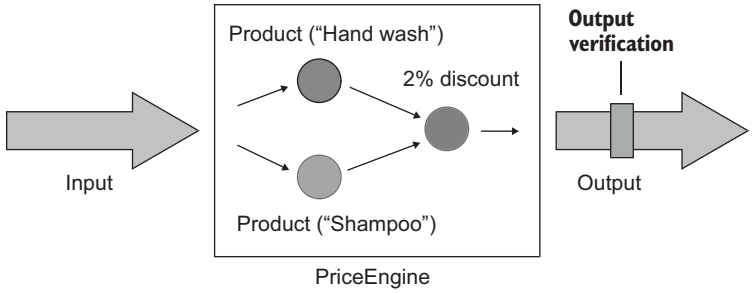

Figure 6.2 `PriceEngine` represented using input-output notation. Its `CalculateDiscount()` method accepts an array of products and calculates a discount.

The output-based style of unit testing is also known as *functional*. This name takes root in *functional programming*, a method of programming that emphasizes a preference for side-effect-free code. We'll talk more about functional programming and functional architecture later in this chapter.

6.1.2 Defining the state-based style

The *state-based* style is about verifying the state of the system after an operation is complete (figure 6.3). The term *state* in this style of testing can refer to the state of the SUT itself, of one of its collaborators, or of an out-of-process dependency, such as the database or the filesystem.

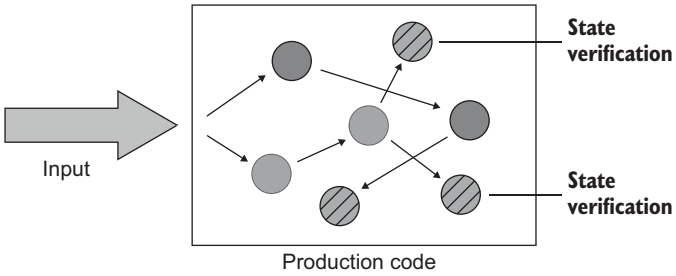

Figure 6.3 In state-based testing, tests verify the final state of the system after an operation is complete. The dashed circles represent that final state.

Here's an example of state-based testing. The `Order` class allows the client to add a new product.

Listing 6.2 State-based testing

```
public class Order
{
    private readonly List<Product> _products = new List<Product>();
    public IReadOnlyList<Product> Products => _products.ToList();

    public void AddProduct(Product product)
    {
        _products.Add(product);
    }
}

[Fact]
public void Adding_a_product_to_an_order()
{
    var product = new Product("Hand wash");
    var sut = new Order();

    sut.AddProduct(product);

    Assert.Equal(1, sut.Products.Count);
    Assert.Equal(product, sut.Products[0]);
}
```

The test verifies the `Products` collection after the addition is completed. Unlike the example of output-based testing in listing 6.1, the outcome of `AddProduct()` is the change made to the order's state.

6.1.3 Defining the communication-based style

Finally, the third style of unit testing is *communication-based* testing. This style uses mocks to verify communications between the system under test and its collaborators (figure 6.4).

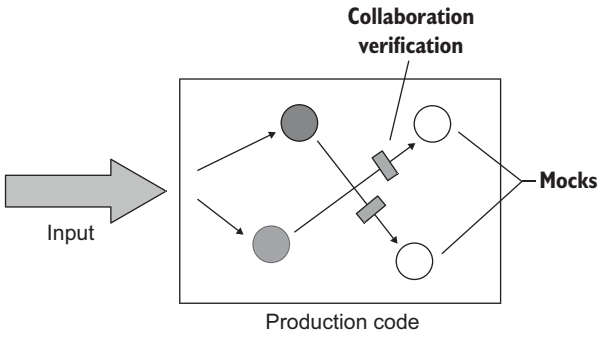

Figure 6.4 In communication-based testing, tests substitute the SUT's collaborators with mocks and verify that the SUT calls those collaborators correctly.

The following listing shows an example of communication-based testing.

Listing 6.3 Communication-based testing

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

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

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

Styles and schools of unit testing

The classical school of unit testing prefers the state-based style over the communication-based one. The London school makes the opposite choice. Both schools use output-based testing.

6.2 Comparing the three styles of unit testing

There's nothing new about output-based, state-based, and communication-based styles of unit testing. In fact, you already saw all of these styles previously in this book. What's interesting is comparing them to each other using the four attributes of a good unit test. Here are those attributes again (refer to chapter 4 for more details):

- Protection against regressions
- Resistance to refactoring
- Fast feedback
- Maintainability

In our comparison, let's look at each of the four separately.

6.2.1 Comparing the styles using the metrics of protection against regressions and feedback speed

Let's first compare the three styles in terms of the protection against regressions and feedback speed attributes, as these attributes are the most straightforward in this particular comparison. The metric of protection against regressions doesn't depend on a particular style of testing. This metric is a product of the following three characteristics:

- The amount of code that is executed during the test
- The complexity of that code
- Its domain significance

Generally, you can write a test that exercises as much or as little code as you like; no particular style provides a benefit in this area. The same is true for the code's complexity and domain significance. The only exception is the communication-based style: overusing it can result in shallow tests that verify only a thin slice of code and mock out everything else. Such shallowness is not a definitive feature of communication-based testing, though, but rather is an extreme case of abusing this technique.

There's little correlation between the styles of testing and the test's feedback speed. As long as your tests don't touch out-of-process dependencies and thus stay in the realm of unit testing, all styles produce tests of roughly equal speed of execution. Communication-based testing can be slightly worse because mocks tend to introduce additional latency at runtime. But the difference is negligible, unless you have tens of thousands of such tests.

6.2.2 Comparing the styles using the metric of resistance to refactoring

When it comes to the metric of resistance to refactoring, the situation is different. *Resistance to refactoring* is the measure of how many *false positives* (false alarms) tests generate during refactorings. False positives, in turn, are a result of tests coupling to code's implementation details as opposed to observable behavior.

Output-based testing provides the best protection against false positives because the resulting tests couple only to the method under test. The only way for such tests to couple to implementation details is when the method under test is itself an implementation detail.

State-based testing is usually more prone to false positives. In addition to the method under test, such tests also work with the class's state. Probabilistically speaking, the greater the coupling between the test and the production code, the greater the chance for this test to tie to a leaking implementation detail. State-based tests tie to a larger API surface, and hence the chances of coupling them to implementation details are also higher.

Communication-based testing is the most vulnerable to false alarms. As you may remember from chapter 5, the vast majority of tests that check interactions with test

doubles end up being brittle. This is always the case for interactions with stubs—you should never check such interactions. Mocks are fine 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. As you can see, using communication-based testing requires extra prudence in order to maintain proper resistance to refactoring.

But just like shallowness, brittleness is not a definitive feature of the communication-based style, either. You can reduce the number of false positives to a minimum by maintaining proper encapsulation and coupling tests to observable behavior only. Admittedly, though, the amount of due diligence varies depending on the style of unit testing.

6.2.3 Comparing the styles using the metric of maintainability

Finally, the maintainability metric is highly correlated with the styles of unit testing; but, unlike with resistance to refactoring, there's not much you can do to mitigate that. *Maintainability* evaluates the unit tests' maintenance costs and is defined by the following two characteristics:

- How hard it is to understand the test, which is a function of the test's size
- How hard it is to run the test, which is a function of how many out-of-process dependencies the test works with directly

Larger tests are less maintainable because they are harder to grasp or change when needed. Similarly, a test that directly works with one or several out-of-process dependencies (such as the database) is less maintainable because you need to spend time keeping those out-of-process dependencies operational: rebooting the database server, resolving network connectivity issues, and so on.

Maintainability of output-based tests

Compared with the other two types of testing, output-based testing is the most maintainable. The resulting tests are almost always short and concise and thus are easier to maintain. This benefit of the output-based style stems from the fact that this style boils down to only two things: supplying an input to a method and verifying its output, which you can often do with just a couple lines of code.

Because the underlying code in output-based testing must not change the global or internal state, these tests don't deal with out-of-process dependencies. Hence, output-based tests are best in terms of both maintainability characteristics.

Maintainability of state-based tests

State-based tests are normally less maintainable than output-based ones. This is because state verification often takes up more space than output verification. Here's another example of state-based testing.

Listing 6.4 State verification that takes up a lot of space

```
[Fact]
public void Adding_a_comment_to_an_article()
{
    var sut = new Article();
    var text = "Comment text";
    var author = "John Doe";
    var now = new DateTime(2019, 4, 1);

    sut.AddComment(text, author, now);

    Assert.Equal(1, sut.Comments.Count);
    Assert.Equal(text, sut.Comments[0].Text);
    Assert.Equal(author, sut.Comments[0].Author);
    Assert.Equal(now, sut.Comments[0].DateCreated);
}
```

Verifies the state of the article

This test adds a comment to an article and then checks to see if the comment appears in the article's list of comments. Although this test is simplified and contains just a single comment, its assertion part already spans four lines. State-based tests often need to verify much more data than that and, therefore, can grow in size significantly.

You can mitigate this issue by introducing helper methods that hide most of the code and thus shorten the test (see listing 6.5), but these methods require significant effort to write and maintain. This effort is justified only when those methods are going to be reused across multiple tests, which is rarely the case. I'll explain more about helper methods in part 3 of this book.

Listing 6.5 Using helper methods in assertions

```
[Fact]
public void Adding_a_comment_to_an_article()
{
    var sut = new Article();
    var text = "Comment text";
    var author = "John Doe";
    var now = new DateTime(2019, 4, 1);

    sut.AddComment(text, author, now);

    sut.ShouldContainNumberOfComments(1)
        .WithComment(text, author, now);
}
```

Helper methods

Another way to shorten a state-based test is to define equality members in the class that is being asserted. In listing 6.6, that's the `Comment` class. You could turn it into a *value object* (a class whose instances are compared by value and not by reference), as shown next; this would also simplify the test, especially if you combined it with an assertion library like Fluent Assertions.

Listing 6.6 `Comment` compared by value

```
[Fact]
public void Adding_a_comment_to_an_article()
{
    var sut = new Article();
    var comment = new Comment(
        "Comment text",
        "John Doe",
        new DateTime(2019, 4, 1));

    sut.AddComment(comment.Text, comment.Author, comment.DateCreated);

    sut.Comments.Should().BeEquivalentTo(comment);
}
```

This test uses the fact that comments can be compared as whole values, without the need to assert individual properties in them. It also uses the `BeEquivalentTo` method from Fluent Assertions, which can compare entire collections, thereby removing the need to check the collection size.

This is a powerful technique, but it works only when the class is inherently a *value* and can be converted into a value object. Otherwise, it leads to *code pollution* (polluting production code base with code whose sole purpose is to enable or, as in this case, simplify unit testing). We'll discuss code pollution along with other unit testing antipatterns in chapter 11.

As you can see, these two techniques—using helper methods and converting classes into value objects—are applicable only occasionally. And even when these techniques are applicable, state-based tests still take up more space than output-based tests and thus remain less maintainable.

Maintainability of communication-based tests

Communication-based tests score worse than output-based and state-based tests on the maintainability metric. Communication-based testing requires setting up test doubles and interaction assertions, and that takes up a lot of space. Tests become even larger and less maintainable when you have *mock chains* (mocks or stubs returning other mocks, which also return mocks, and so on, several layers deep).

6.2.4 Comparing the styles: The results

Let's now compare the styles of unit testing using the attributes of a good unit test. Table 6.1 sums up the comparison results. As discussed in section 6.2.1, all three styles score equally with the metrics of protection against regressions and feedback speed; hence, I'm omitting these metrics from the comparison.

Output-based testing shows the best results. This style produces tests that rarely couple to implementation details and thus don't require much due diligence to maintain proper resistance to refactoring. Such tests are also the most maintainable due to their conciseness and lack of out-of-process dependencies.

Table 6.1 The three styles of unit testing: The comparisons

	Output-based	State-based	Communication-based
Due diligence to maintain resistance to refactoring	Low	Medium	Medium
Maintainability costs	Low	Medium	High

State-based and communication-based tests are worse on both metrics. These are more likely to couple to a leaking implementation detail, and they also incur higher maintenance costs due to being larger in size.

Always prefer output-based testing over everything else. Unfortunately, it's easier said than done. This style of unit testing is only applicable to code that is written in a functional way, which is rarely the case for most object-oriented programming languages. Still, there are techniques you can use to transition more of your tests toward the output-based style.

The rest of this chapter shows how to transition from state-based and collaboration-based testing to output-based testing. The transition requires you to make your code more purely functional, which, in turn, enables the use of output-based tests instead of state- or communication-based ones.

6.3 Understanding functional architecture

Some groundwork is needed before I can show how to make the transition. In this section, you'll see what functional programming and functional architecture are and how the latter relates to the hexagonal architecture. Section 6.4 illustrates the transition using an example.

Note that this isn't a deep dive into the topic of functional programming, but rather an explanation of the basic principles behind it. These basic principles should be enough to understand the connection between functional programming and output-based testing. For a deeper look at functional programming, see Scott Wlaschin's website and books at https://fsharpforfunandprofit.com/books.

6.3.1 What is functional programming?

As I mentioned in section 6.1.1, the output-based unit testing style is also known as *functional*. That's because it requires the underlying production code to be written in a purely functional way, using functional programming. So, what is functional programming?

Functional programming is programming with mathematical functions. A *mathematical function* (also known as *pure function*) is a function (or method) that doesn't have any hidden inputs or outputs. All inputs and outputs of a mathematical function must be explicitly expressed in its *method signature*, which consists of the method's name, arguments, and return type. A mathematical function produces the same output for a given input regardless of how many times it is called.

Let's take the `CalculateDiscount()` method from listing 6.1 as an example (I'm copying it here for convenience):

```
public decimal CalculateDiscount(Product[] products)
{
    decimal discount = products.Length * 0.01m;
    return Math.Min(discount, 0.2m);
}
```

This method has one input (a `Product` array) and one output (the `decimal` discount), both of which are explicitly expressed in the method's signature. There are no hidden inputs or outputs. This makes `CalculateDiscount()` a mathematical function (figure 6.5).

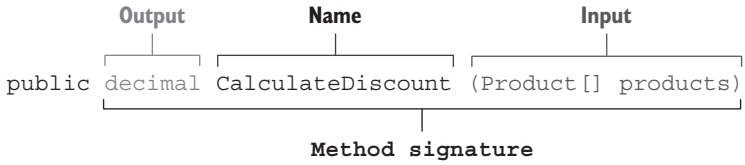

Figure 6.5 `CalculateDiscount()` has one input (a `Product` array) and one output (the `decimal` discount). Both the input and the output are explicitly expressed in the method's signature, which makes `CalculateDiscount()` a mathematical function.

Methods with no hidden inputs and outputs are called mathematical functions because such methods adhere to the definition of a function in mathematics.

DEFINITION In mathematics, a *function* is a relationship between two sets that for each element in the first set, finds exactly one element in the second set.

Figure 6.6 shows how for each input number *x*, function `f(x) = x + 1` finds a corresponding number *y*. Figure 6.7 displays the `CalculateDiscount()` method using the same notation as in figure 6.6.

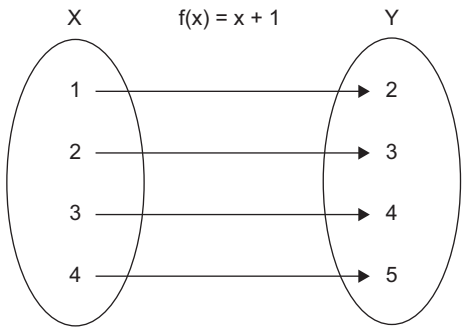

Figure 6.6 A typical example of a function in mathematics is `f(x) = x + 1`. For each input number *x* in set X, the function finds a corresponding number *y* in set Y.

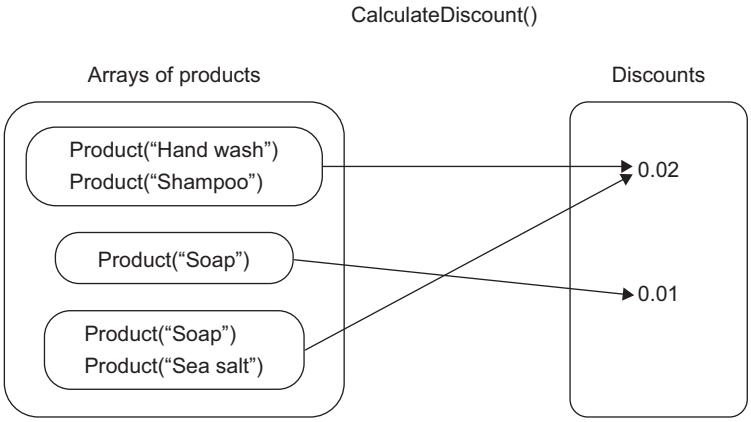

Figure 6.7 The `CalculateDiscount()` method represented using the same notation as the function `f(x) = x + 1`. For each input array of products, the method finds a corresponding discount as an output.

Explicit inputs and outputs make mathematical functions extremely testable because the resulting tests are short, simple, and easy to understand and maintain. Mathematical functions are the only type of methods where you can apply output-based testing, which has the best maintainability and the lowest chance of producing a false positive.

On the other hand, hidden inputs and outputs make the code less testable (and less readable, too). Types of such hidden inputs and outputs include the following:

- *Side effects*—A *side effect* is an output that isn't expressed in the method signature and, therefore, is hidden. An operation creates a side effect when it mutates the state of a class instance, updates a file on the disk, and so on.
- *Exceptions*—When a method throws an exception, it creates a path in the program flow that bypasses the contract established by the method's signature. The thrown exception can be caught anywhere in the call stack, thus introducing an additional output that the method signature doesn't convey.
- *A reference to an internal or external state*—For example, a method can get the current date and time using a static property such as `DateTime.Now`. It can query data from the database, or it can refer to a private mutable field. These are all inputs to the execution flow that aren't present in the method signature and, therefore, are hidden.

A good rule of thumb when determining whether a method is a mathematical function is to see if you can replace a call to that method with its return value without changing the program's behavior. The ability to replace a method call with the corresponding value is known as *referential transparency*. Look at the following method, for example:

```
public int Increment(int x)
{
    return x + 1;
}
```

This method is a mathematical function. These two statements are equivalent to each other:

```
int y = Increment(4);
int y = 5;
```

On the other hand, the following method is *not* a mathematical function. You can't replace it with the return value because that return value doesn't represent all of the method's outputs. In this example, the hidden output is the change to field `x` (a side effect):

```
int x = 0;
public int Increment()
{
    x++;
    return x;
}
```

Side effects are the most prevalent type of hidden outputs. The following listing shows an `AddComment` method that looks like a mathematical function on the surface but actually isn't one. Figure 6.8 shows the method graphically.

Listing 6.7 Modification of an internal state

```
public Comment AddComment(string text)
{
    var comment = new Comment(text);
    _comments.Add(comment);          <-- Side effect
     return comment;
}
```

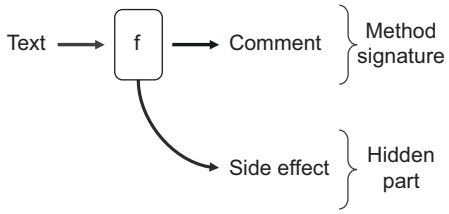

Figure 6.8 Method `AddComment` (shown as `f`) has a `text` input and a `Comment` output, which are both expressed in the method signature. The side effect is an additional hidden output.

6.3.2 What is functional architecture?

You can't create an application that doesn't incur any side effects whatsoever, of course. Such an application would be impractical. After all, side effects are what you create all applications for: updating the user's information, adding a new order line to the shopping cart, and so on.

The goal of functional programming is not to eliminate side effects altogether but rather to introduce a separation between code that handles business logic and code that incurs side effects. These two responsibilities are complex enough on their own; mixing them together multiplies the complexity and hinders code maintainability in the long run. This is where functional architecture comes into play. It separates business logic from side effects by *pushing those side effects to the edges of a business operation.*

> **DEFINITION** *Functional architecture* maximizes the amount of code written in a purely functional (immutable) way, while minimizing code that deals with side effects. *Immutable* means unchangeable: once an object is created, its state can't be modified. This is in contrast to a *mutable* object (changeable object), which can be modified after it is created.

The separation between business logic and side effects is done by segregating two types of code:

- *Code that makes a decision*—This code doesn't require side effects and thus can be written using mathematical functions.
- *Code that acts upon that decision*—This code converts all the decisions made by the mathematical functions into visible bits, such as changes in the database or messages sent to a bus.

The code that makes decisions is often referred to as a *functional core* (also known as an *immutable core*). The code that acts upon those decisions is a *mutable shell* (figure 6.9).

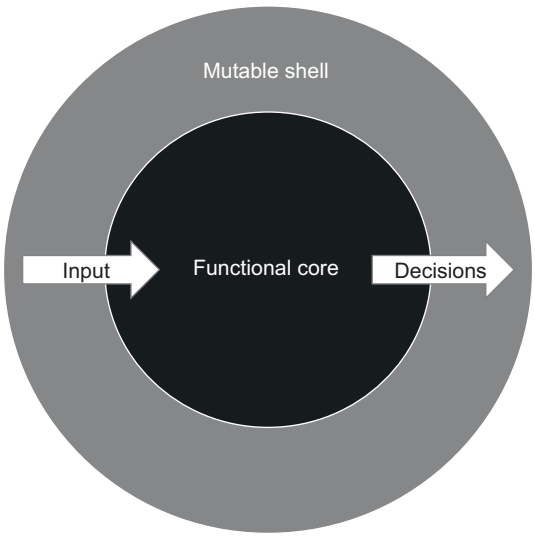

Figure 6.9 In functional architecture, the functional core is implemented using mathematical functions and makes all decisions in the application. The mutable shell provides the functional core with input data and interprets its decisions by applying side effects to out-of-process dependencies such as a database.

The functional core and the mutable shell cooperate in the following way:

- The mutable shell gathers all the inputs.
- The functional core generates decisions.
- The shell converts the decisions into side effects.

To maintain a proper separation between these two layers, you need to make sure the classes representing the decisions contain enough information for the mutable shell to act upon them without additional decision-making. In other words, the mutable shell should be as dumb as possible. The goal is to cover the functional core extensively with output-based tests and leave the mutable shell to a much smaller number of integration tests.

Encapsulation and immutability

Like encapsulation, functional architecture (in general) and immutability (in particular) serve the same goal as unit testing: enabling sustainable growth of your software project. In fact, there's a deep connection between the concepts of encapsulation and immutability.

As you may remember from chapter 5, *encapsulation* is the act of protecting your code against inconsistencies. Encapsulation safeguards the class's internals from corruption by

- Reducing the API surface area that allows for data modification
- Putting the remaining APIs under scrutiny

Immutability tackles this issue of preserving invariants from another angle. With immutable classes, you don't need to worry about state corruption because it's impossible to corrupt something that cannot be changed in the first place. As a consequence, there's no need for encapsulation in functional programming. You only need to validate the class's state once, when you create an instance of it. After that, you can freely pass this instance around. When all your data is immutable, the whole set of issues related to the lack of encapsulation simply vanishes.

There's a great quote from Michael Feathers in that regard:

> *Object-oriented programming makes code understandable by encapsulating moving parts. Functional programming makes code understandable by minimizing moving parts.*

6.3.3 Comparing functional and hexagonal architectures

There are a lot of similarities between functional and hexagonal architectures. Both of them are built around the idea of separation of concerns. The details of that separation vary, though.

As you may remember from chapter 5, the hexagonal architecture differentiates the domain layer and the application services layer (figure 6.10). The *domain layer* is accountable for business logic while the *application services layer*, for communication with

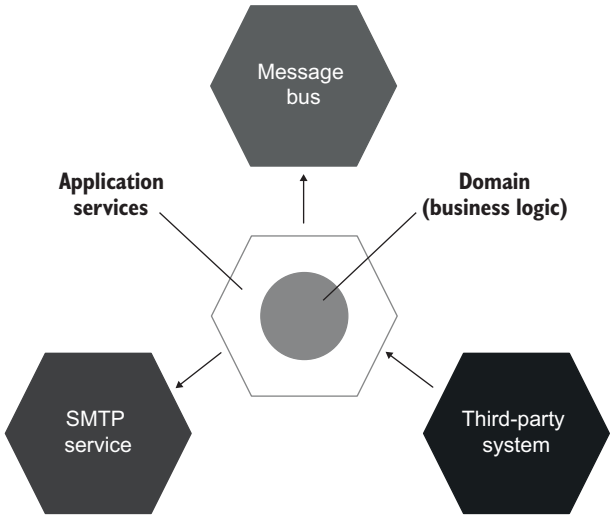

Figure 6.10 Hexagonal architecture is a set of interacting applications—hexagons. Your application consists of a domain layer and an application services layer, which correspond to a functional core and a mutable shell in functional architecture.

external applications such as a database or an SMTP service. This is very similar to functional architecture, where you introduce the separation of decisions and actions.

Another similarity is the one-way flow of dependencies. In the hexagonal architecture, classes inside the domain layer should only depend on each other; they should not depend on classes from the application services layer. Likewise, the immutable core in functional architecture doesn't depend on the mutable shell. It's self-sufficient and can work in isolation from the outer layers. This is what makes functional architecture so testable: you can strip the immutable core from the mutable shell entirely and simulate the inputs that the shell provides using simple values.

The difference between the two is in their treatment of side effects. Functional architecture pushes *all* side effects out of the immutable core to the edges of a business operation. These edges are handled by the mutable shell. On the other hand, the hexagonal architecture is fine with side effects made by the domain layer, as long as they are limited to that domain layer only. All modifications in hexagonal architecture should be contained within the domain layer and *not* cross that layer's boundary. For example, a domain class instance can't persist something to the database directly, but it can change its own state. An application service will then pick up this change and apply it to the database.

NOTE Functional architecture is a subset of the hexagonal architecture. You can view functional architecture as the hexagonal architecture taken to an extreme.

6.4 Transitioning to functional architecture and output-based testing

In this section, we'll take a sample application and refactor it toward functional architecture. You'll see two refactoring stages:

- Moving from using an out-of-process dependency to using mocks
- Moving from using mocks to using functional architecture

The transition affects test code, too! We'll refactor state-based and communication-based tests to the output-based style of unit testing. Before starting the refactoring, let's review the sample project and tests covering it.

6.4.1 Introducing an audit system

The sample project is an audit system that keeps track of all visitors in an organization. It uses flat text files as underlying storage with the structure shown in figure 6.11. The system appends the visitor's name and the time of their visit to the end of the most recent file. When the maximum number of entries per file is reached, a new file with an incremented index is created.

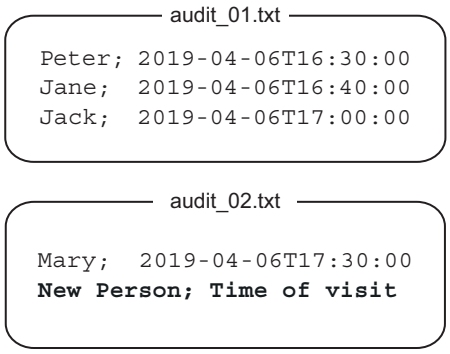

Figure 6.11 The audit system stores information about visitors in text files with a specific format. When the maximum number of entries per file is reached, the system creates a new file.

The following listing shows the initial version of the system.

Listing 6.8 Initial implementation of the audit system

```
public class AuditManager
{
    private readonly int _maxEntriesPerFile;
    private readonly string _directoryName;

    public AuditManager(int maxEntriesPerFile, string directoryName)
    {
        _maxEntriesPerFile = maxEntriesPerFile;
        _directoryName = directoryName;
    }
```

```
    public void AddRecord(string visitorName, DateTime timeOfVisit)
    {
        string[] filePaths = Directory.GetFiles(_directoryName);
        (int index, string path)[] sorted = SortByIndex(filePaths);

        string newRecord = visitorName + ';' + timeOfVisit;

        if (sorted.Length == 0)
        {
            string newFile = Path.Combine(_directoryName, "audit_1.txt");
            File.WriteAllText(newFile, newRecord);
            return;
        }

        (int currentFileIndex, string currentFilePath) = sorted.Last();
        List<string> lines = File.ReadAllLines(currentFilePath).ToList();

        if (lines.Count < _maxEntriesPerFile)
        {
            lines.Add(newRecord);
            string newContent = string.Join("\r\n", lines);
            File.WriteAllText(currentFilePath, newContent);
        }
        else
        {
            int newIndex = currentFileIndex + 1;
            string newName = $"audit_{newIndex}.txt";
            string newFile = Path.Combine(_directoryName, newName);
            File.WriteAllText(newFile, newRecord);
        }
    }
}
```

The code might look a bit large, but it's quite simple. `AuditManager` is the main class in the application. Its constructor accepts the maximum number of entries per file and the working directory as configuration parameters. The only public method in the class is `AddRecord`, which does all the work of the audit system:

- Retrieves a full list of files from the working directory
- Sorts them by index (all filenames follow the same pattern: audit_{index}.txt [for example, audit_1.txt])
- If there are no audit files yet, creates a first one with a single record
- If there are audit files, gets the most recent one and either appends the new record to it or creates a new file, depending on whether the number of entries in that file has reached the limit

The `AuditManager` class is hard to test as-is, because it's tightly coupled to the file-system. Before the test, you'd need to put files in the right place, and after the test finishes, you'd read those files, check their contents, and clear them out (figure 6.12).

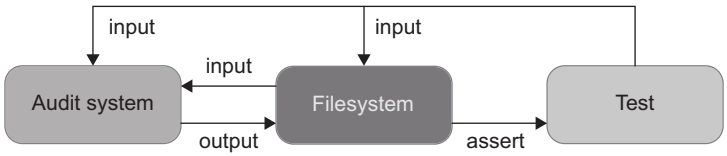

Figure 6.12 Tests covering the initial version of the audit system would have to work directly with the filesystem.

You won't be able to parallelize such tests—at least, not without additional effort that would significantly increase maintenance costs. The bottleneck is the filesystem: it's a shared dependency through which tests can interfere with each other's execution flow.

The filesystem also makes the tests slow. Maintainability suffers, too, because you have to make sure the working directory exists and is accessible to tests—both on your local machine and on the build server. Table 6.2 sums up the scoring.

Table 6.2 The initial version of the audit system scores badly on two out of the four attributes of a good test.

	Initial version
Protection against regressions	Good
Resistance to refactoring	Good
Fast feedback	Bad
Maintainability	Bad

By the way, tests working directly with the filesystem don't fit the definition of a unit test. They don't comply with the second and the third attributes of a unit test, thereby falling into the category of integration tests (see chapter 2 for more details):

- A unit test verifies a single unit of behavior,
- Does it quickly,
- And does it in isolation from other tests.

6.4.2 Using mocks to decouple tests from the filesystem

The usual solution to the problem of tightly coupled tests is to mock the filesystem. You can extract all operations on files into a separate class (`IFileSystem`) and inject that class into `AuditManager` via the constructor. The tests will then mock this class and capture the writes the audit system do to the files (figure 6.13).

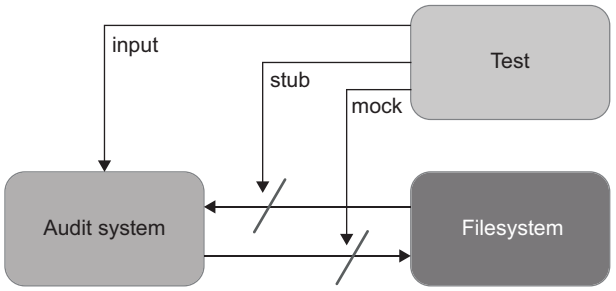

Figure 6.13 Tests can mock the filesystem and capture the writes the audit system makes to the files.

The following listing shows how the filesystem is injected into `AuditManager`.

Listing 6.9 Injecting the filesystem explicitly via the constructor

```
public class AuditManager
{
    private readonly int _maxEntriesPerFile;
    private readonly string _directoryName;
    private readonly IFileSystem _fileSystem;

    public AuditManager(
        int maxEntriesPerFile,
        string directoryName,
        IFileSystem fileSystem)
    {
        _maxEntriesPerFile = maxEntriesPerFile;
        _directoryName = directoryName;
        _fileSystem = fileSystem;
    }
}
```

The new interface represents the filesystem.

And next is the `AddRecord` method.

Listing 6.10 Using the new `IFileSystem` interface

The new interface in action

```
public void AddRecord(string visitorName, DateTime timeOfVisit)
{
    string[] filePaths = _fileSystem
        .GetFiles(_directoryName);
    (int index, string path)[] sorted = SortByIndex(filePaths);

    string newRecord = visitorName + ';' + timeOfVisit;

    if (sorted.Length == 0)
    {
        string newFile = Path.Combine(_directoryName, "audit_1.txt");
        _fileSystem.WriteAllText(
            newFile, newRecord);
        return;
    }
```

```
    (int currentFileIndex, string currentFilePath) = sorted.Last();
    List<string> lines = _fileSystem
        .ReadAllLines(currentFilePath);

    if (lines.Count < _maxEntriesPerFile)
    {
        lines.Add(newRecord);
        string newContent = string.Join("\r\n", lines);
        _fileSystem.WriteAllText(
            currentFilePath, newContent);
    }
    else
    {
        int newIndex = currentFileIndex + 1;
        string newName = $"audit_{newIndex}.txt";
        string newFile = Path.Combine(_directoryName, newName);
        _fileSystem.WriteAllText(
            newFile, newRecord);
    }
}
```

The new interface in action

In listing 6.10, `IFileSystem` is a new custom interface that encapsulates the work with the filesystem:

```
public interface IFileSystem
{
    string[] GetFiles(string directoryName);
    void WriteAllText(string filePath, string content);
    List<string> ReadAllLines(string filePath);
}
```

Now that `AuditManager` is decoupled from the filesystem, the shared dependency is gone, and tests can execute independently from each other. Here's one such test.

Listing 6.11 Checking the audit system's behavior using a mock

```
[Fact]
public void A_new_file_is_created_when_the_current_file_overflows()
{
    var fileSystemMock = new Mock<IFileSystem>();
    fileSystemMock
        .Setup(x => x.GetFiles("audits"))
        .Returns(new string[]
        {
            @"audits\audit_1.txt",
            @"audits\audit_2.txt"
        });
    fileSystemMock
        .Setup(x => x.ReadAllLines(@"audits\audit_2.txt"))
        .Returns(new List<string>
        {
            "Peter; 2019-04-06T16:30:00",
            "Jane; 2019-04-06T16:40:00",
```

```
            "Jack; 2019-04-06T17:00:00"
        });
    var sut = new AuditManager(3, "audits", fileSystemMock.Object);

    sut.AddRecord("Alice", DateTime.Parse("2019-04-06T18:00:00"));

    fileSystemMock.Verify(x => x.WriteAllText(
        @"audits\audit_3.txt",
        "Alice;2019-04-06T18:00:00"));
}
```

This test verifies that when the number of entries in the current file reaches the limit (3, in this example), a new file with a single audit entry is created. Note that this is a legitimate use of mocks. The application creates files that are visible to end users (assuming that those users use another program to read the files, be it specialized software or a simple notepad.exe). Therefore, communications with the filesystem and the side effects of these communications (that is, the changes in files) are part of the application's observable behavior. As you may remember from chapter 5, that's the only legitimate use case for mocking.

This alternative implementation is an improvement over the initial version. Since tests no longer access the filesystem, they execute faster. And because you don't need to look after the filesystem to keep the tests happy, the maintenance costs are also reduced. Protection against regressions and resistance to refactoring didn't suffer from the refactoring either. Table 6.3 shows the differences between the two versions.

Table 6.3 The version with mocks compared to the initial version of the audit system

	Initial version	With mocks
Protection against regressions	Good	Good
Resistance to refactoring	Good	Good
Fast feedback	Bad	Good
Maintainability	Bad	Moderate

We can still do better, though. The test in listing 6.11 contains convoluted setups, which is less than ideal in terms of maintenance costs. Mocking libraries try their best to be helpful, but the resulting tests are still not as readable as those that rely on plain input and output.

6.4.3 Refactoring toward functional architecture

Instead of hiding side effects behind an interface and injecting that interface into `AuditManager`, you can move those side effects out of the class entirely. `AuditManager` is then only responsible for making a decision about what to do with the files. A new class, `Persister`, acts on that decision and applies updates to the filesystem (figure 6.14).

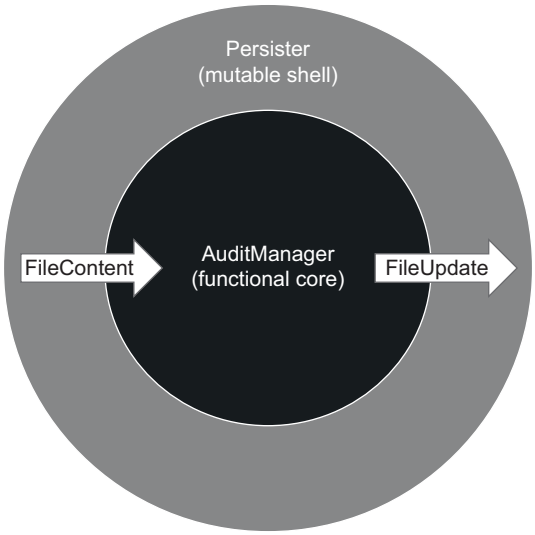

Figure 6.14 `Persister` and `AuditManager` form the functional architecture. `Persister` gathers files and their contents from the working directory, feeds them to `AuditManager`, and then converts the return value into changes in the filesystem.

`Persister` in this scenario acts as a mutable shell, while `AuditManager` becomes a functional (immutable) core. The following listing shows `AuditManager` after the refactoring.

Listing 6.12 The `AuditManager` class after refactoring

```
public class AuditManager
{
    private readonly int _maxEntriesPerFile;

    public AuditManager(int maxEntriesPerFile)
    {
        _maxEntriesPerFile = maxEntriesPerFile;
    }

    public FileUpdate AddRecord(
        FileContent[] files,
        string visitorName,
        DateTime timeOfVisit)
    {
        (int index, FileContent file)[] sorted = SortByIndex(files);

        string newRecord = visitorName + ';' + timeOfVisit;

        if (sorted.Length == 0)
        {
            return new FileUpdate(                    Returns an update
                "audit_1.txt", newRecord);            instruction
        }

        (int currentFileIndex, FileContent currentFile) = sorted.Last();
        List<string> lines = currentFile.Lines.ToList();
```

```
        if (lines.Count < _maxEntriesPerFile)
        {
            lines.Add(newRecord);
            string newContent = string.Join("\r\n", lines);
            return new FileUpdate(
                currentFile.FileName, newContent);
        }
        else
        {
            int newIndex = currentFileIndex + 1;
            string newName = $"audit_{newIndex}.txt";
            return new FileUpdate(
                newName, newRecord);
        }
    }
}
```

Returns an update instruction

Instead of the working directory path, `AuditManager` now accepts an array of `FileContent`. This class includes everything `AuditManager` needs to know about the filesystem to make a decision:

```
public class FileContent
{
    public readonly string FileName;
    public readonly string[] Lines;

    public FileContent(string fileName, string[] lines)
    {
        FileName = fileName;
        Lines = lines;
    }
}
```

And, instead of mutating files in the working directory, `AuditManager` now returns an instruction for the side effect it would like to perform:

```
public class FileUpdate
{
    public readonly string FileName;
    public readonly string NewContent;

    public FileUpdate(string fileName, string newContent)
    {
        FileName = fileName;
        NewContent = newContent;
    }
}
```

The following listing shows the `Persister` class.

Listing 6.13 The mutable shell acting on `AuditManager`'s decision

```
public class Persister
{
    public FileContent[] ReadDirectory(string directoryName)
    {
        return Directory
            .GetFiles(directoryName)
            .Select(x => new FileContent(
                Path.GetFileName(x),
                File.ReadAllLines(x)))
            .ToArray();
    }

    public void ApplyUpdate(string directoryName, FileUpdate update)
    {
        string filePath = Path.Combine(directoryName, update.FileName);
        File.WriteAllText(filePath, update.NewContent);
    }
}
```

Notice how trivial this class is. All it does is read content from the working directory and apply updates it receives from `AuditManager` back to that working directory. It has no branching (no `if` statements); all the complexity resides in the `AuditManager` class. *This is the separation between business logic and side effects in action.*

To maintain such a separation, you need to keep the interface of `FileContent` and `FileUpdate` as close as possible to that of the framework's built-in file-interaction commands. All the parsing and preparation should be done in the functional core, so that the code outside of that core remains trivial. For example, if .NET didn't contain the built-in `File.ReadAllLines()` method, which returns the file content as an array of lines, and only has `File.ReadAllText()`, which returns a single string, you'd need to replace the `Lines` property in `FileContent` with a `string` too and do the parsing in `AuditManager`:

```
public class FileContent
{
    public readonly string FileName;
    public readonly string Text; // previously, string[] Lines;
}
```

To glue `AuditManager` and `Persister` together, you need another class: an application service in the hexagonal architecture taxonomy, as shown in the following listing.

Listing 6.14 Gluing together the functional core and mutable shell

```
public class ApplicationService
{
    private readonly string _directoryName;
    private readonly AuditManager _auditManager;
    private readonly Persister _persister;
```

```
    public ApplicationService(
        string directoryName, int maxEntriesPerFile)
    {
        _directoryName = directoryName;
        _auditManager = new AuditManager(maxEntriesPerFile);
        _persister = new Persister();
    }

    public void AddRecord(string visitorName, DateTime timeOfVisit)
    {
        FileContent[] files = _persister.ReadDirectory(_directoryName);
        FileUpdate update = _auditManager.AddRecord(
            files, visitorName, timeOfVisit);
        _persister.ApplyUpdate(_directoryName, update);
    }
}
```

Along with gluing the functional core together with the mutable shell, the application service also provides an entry point to the system for external clients (figure 6.15). With this implementation, it becomes easy to check the audit system's behavior. All tests now boil down to supplying a hypothetical state of the working directory and verifying the decision `AuditManager` makes.

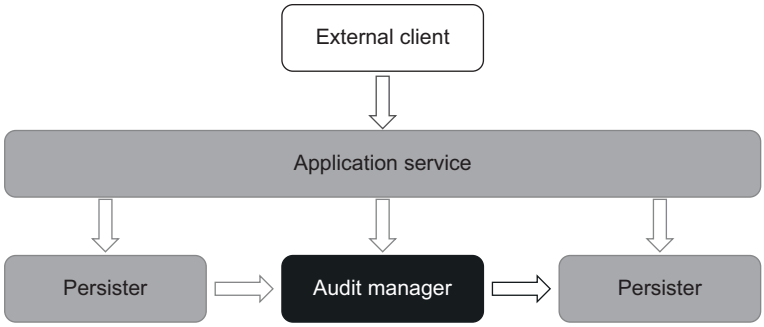

Figure 6.15 `ApplicationService` **glues the functional core (**`AuditManager`**) and the mutable shell (**`Persister`**) together and provides an entry point for external clients. In the hexagonal architecture taxonomy,** `ApplicationService` **and** `Persister` **are part of the application services layer, while** `AuditManager` **belongs to the domain model.**

Listing 6.15 The test without mocks

```
[Fact]
public void A_new_file_is_created_when_the_current_file_overflows()
{
    var sut = new AuditManager(3);
    var files = new FileContent[]
    {
        new FileContent("audit_1.txt", new string[0]),
```

```
        new FileContent("audit_2.txt", new string[]
        {
            "Peter; 2019-04-06T16:30:00",
            "Jane; 2019-04-06T16:40:00",
            "Jack; 2019-04-06T17:00:00"
        })
    };

    FileUpdate update = sut.AddRecord(
        files, "Alice", DateTime.Parse("2019-04-06T18:00:00"));

    Assert.Equal("audit_3.txt", update.FileName);
    Assert.Equal("Alice;2019-04-06T18:00:00", update.NewContent);
}
```

This test retains the improvement the test with mocks made over the initial version (fast feedback) but also further improves on the maintainability metric. There's no need for complex mock setups anymore, only plain inputs and outputs, which helps the test's readability a lot. Table 6.4 compares the output-based test with the initial version and the version with mocks.

Table 6.4 The output-based test compared to the previous two versions

	Initial version	With mocks	Output-based
Protection against regressions	Good	Good	Good
Resistance to refactoring	Good	Good	Good
Fast feedback	Bad	Good	Good
Maintainability	Bad	Moderate	Good

Notice that the instructions generated by a functional core are always a *value* or a *set of values*. Two instances of such a value are interchangeable as long as their contents match. You can take advantage of this fact and improve test readability even further by turning `FileUpdate` into a value object. To do that in .NET, you need to either convert the class into a `struct` or define custom equality members. That will give you comparison by value, as opposed to the comparison by reference, which is the default behavior for classes in C#. Comparison by value also allows you to compress the two assertions from listing 6.15 into one:

```
Assert.Equal(
    new FileUpdate("audit_3.txt", "Alice;2019-04-06T18:00:00"),
    update);
```

Or, using Fluent Assertions,

```
update.Should().Be(
    new FileUpdate("audit_3.txt", "Alice;2019-04-06T18:00:00"));
```

6.4.4 Looking forward to further developments

Let's step back for a minute and look at further developments that could be done in our sample project. The audit system I showed you is quite simple and contains only three branches:

- Creating a new file in case of an empty working directory
- Appending a new record to an existing file
- Creating another file when the number of entries in the current file exceeds the limit

Also, there's only one use case: addition of a new entry to the audit log. What if there were another use case, such as deleting all mentions of a particular visitor? And what if the system needed to do validations (say, for the maximum length of the visitor's name)?

Deleting all mentions of a particular visitor could potentially affect several files, so the new method would need to return multiple file instructions:

```
public FileUpdate[] DeleteAllMentions(
    FileContent[] files, string visitorName)
```

Furthermore, business people might require that you not keep empty files in the working directory. If the deleted entry was the last entry in an audit file, you would need to remove that file altogether. To implement this requirement, you could rename `FileUpdate` to `FileAction` and introduce an additional `ActionType` enum field to indicate whether it was an update or a deletion.

Error handling also becomes simpler and more explicit with functional architecture. You could embed errors into the method's signature, either in the `FileUpdate` class or as a separate component:

```
public (FileUpdate update, Error error) AddRecord(
    FileContent[] files,
    string visitorName,
    DateTime timeOfVisit)
```

The application service would then check for this error. If it was there, the service wouldn't pass the update instruction to the persister, instead propagating an error message to the user.

6.5 Understanding the drawbacks of functional architecture

Unfortunately, functional architecture isn't always attainable. And even when it is, the maintainability benefits are often offset by a performance impact and increase in the size of the code base. In this section, we'll explore the costs and the trade-offs attached to functional architecture.

6.5.1 Applicability of functional architecture

Functional architecture worked for our audit system because this system could gather all the inputs up front, before making a decision. Often, though, the execution flow is less straightforward. You might need to query additional data from an out-of-process dependency, based on an intermediate result of the decision-making process.

Here's an example. Let's say the audit system needs to check the visitor's access level if the number of times they have visited during the last 24 hours exceeds some threshold. And let's also assume that all visitors' access levels are stored in a database. You can't pass an `IDatabase` instance to `AuditManager` like this:

```
public FileUpdate AddRecord(
    FileContent[] files, string visitorName,
    DateTime timeOfVisit, IDatabase database
)
```

Such an instance would introduce a hidden input to the `AddRecord()` method. This method would, therefore, cease to be a mathematical function (figure 6.16), which means you would no longer be able to apply output-based testing.

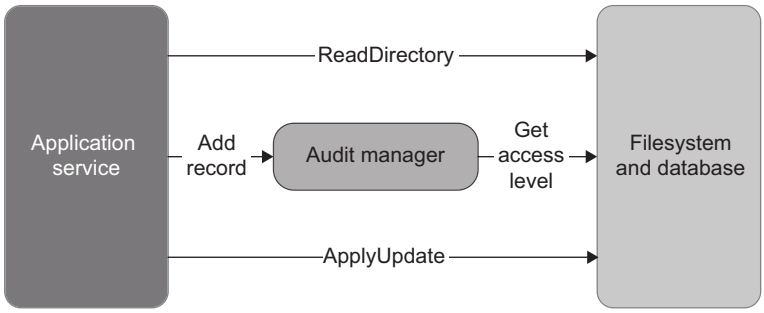

Figure 6.16 A dependency on the database introduces a hidden input to `AuditManager`. Such a class is no longer purely functional, and the whole application no longer follows the functional architecture.

There are two solutions in such a situation:

- You can gather the visitor's access level in the application service up front, along with the directory content.
- You can introduce a new method such as `IsAccessLevelCheckRequired()` in `AuditManager`. The application service would call this method before `AddRecord()`, and if it returned `true`, the service would get the access level from the database and pass it to `AddRecord()`.

Both approaches have drawbacks. The first one concedes performance—it unconditionally queries the database, even in cases when the access level is not required. But this approach keeps the separation of business logic and communication with external

systems fully intact: all decision-making resides in `AuditManager` as before. The second approach concedes a degree of that separation for performance gains: the decision as to whether to call the database now goes to the application service, not `AuditManager`.

Note that, unlike these two options, making the domain model (`AuditManager`) depend on the database isn't a good idea. I'll explain more about keeping the balance between performance and separation of concerns in the next two chapters.

Collaborators vs. values

You may have noticed that `AuditManager`'s `AddRecord()` method has a dependency that's not present in its signature: the `_maxEntriesPerFile` field. The audit manager refers to this field to make a decision to either append an existing audit file or create a new one.

Although this dependency isn't present among the method's arguments, it's not hidden. It can be derived from the class's constructor signature. And because the `_maxEntriesPerFile` field is immutable, it stays the same between the class instantiation and the call to `AddRecord()`. In other words, that field is a *value*.

The situation with the `IDatabase` dependency is different because it's a *collaborator*, not a value like `_maxEntriesPerFile`. As you may remember from chapter 2, a collaborator is a dependency that is one or the other of the following:

- Mutable (allows for modification of its state)
- A proxy to data that is not yet in memory (a shared dependency)

The `IDatabase` instance falls into the second category and, therefore, is a collaborator. It requires an additional call to an out-of-process dependency and thus precludes the use of output-based testing.

NOTE A class from the functional core should work not with a collaborator, but with the product of its work, a value.

6.5.2 Performance drawbacks

The performance impact on the system as a whole is a common argument against functional architecture. Note that it's not the performance of tests that suffers. The output-based tests we ended up with work as fast as the tests with mocks. It's that the system itself now has to do more calls to out-of-process dependencies and becomes less performant. The initial version of the audit system didn't read all files from the working directory, and neither did the version with mocks. But the final version does in order to comply with the read-decide-act approach.

The choice between a functional architecture and a more traditional one is a trade-off between performance and code maintainability (both production and test code). In some systems where the performance impact is not as noticeable, it's better to go with functional architecture for additional gains in maintainability. In others, you might need to make the opposite choice. There's no one-size-fits-all solution.

6.5.3 Increase in the code base size

The same is true for the size of the code base. Functional architecture requires a clear separation between the functional (immutable) core and the mutable shell. This necessitates additional coding initially, although it ultimately results in reduced code complexity and gains in maintainability.

Not all projects exhibit a high enough degree of complexity to justify such an initial investment, though. Some code bases aren't that significant from a business perspective or are just plain too simple. It doesn't make sense to use functional architecture in such projects because the initial investment will never pay off. Always apply functional architecture strategically, taking into account the complexity and importance of your system.

Finally, don't go for purity of the functional approach if that purity comes at too high a cost. In most projects, you won't be able to make the domain model fully immutable and thus can't rely solely on output-based tests, at least not when using an OOP language like C# or Java. In most cases, you'll have a combination of output-based and state-based styles, with a small mix of communication-based tests, and that's fine. The goal of this chapter is not to incite you to transition *all* your tests toward the output-based style; the goal is to transition as many of them as reasonably possible. The difference is subtle but important.

Summary

- *Output-based* testing is a style of testing where you feed an input to the SUT and check the output it produces. This style of testing assumes there are no hidden inputs or outputs, and the only result of the SUT's work is the value it returns.
- *State-based* testing verifies the state of the system after an operation is completed.
- In *communication-based* testing, you use mocks to verify communications between the system under test and its collaborators.
- The classical school of unit testing prefers the state-based style over the communication-based one. The London school has the opposite preference. Both schools use output-based testing.
- Output-based testing produces tests of the highest quality. Such tests rarely couple to implementation details and thus are resistant to refactoring. They are also small and concise and thus are more maintainable.
- State-based testing requires extra prudence to avoid brittleness: you need to make sure you don't expose a private state to enable unit testing. Because state-based tests tend to be larger than output-based tests, they are also less maintainable. Maintainability issues can sometimes be mitigated (but not eliminated) with the use of helper methods and value objects.
- Communication-based testing also requires extra prudence to avoid brittleness. You should only verify communications that cross the application boundary and whose side effects are visible to the external world. Maintainability of

communication-based tests is worse compared to output-based and state-based tests. Mocks tend to occupy a lot of space, and that makes tests less readable.

- *Functional programming* is programming with mathematical functions.
- A *mathematical function* is a function (or method) that doesn't have any hidden inputs or outputs. Side effects and exceptions are hidden outputs. A reference to an internal or external state is a hidden input. Mathematical functions are explicit, which makes them extremely testable.
- The goal of functional programming is to introduce a separation between business logic and side effects.
- Functional architecture helps achieve that separation by pushing side effects to the edges of a business operation. This approach maximizes the amount of code written in a purely functional way while minimizing code that deals with side effects.
- Functional architecture divides all code into two categories: functional core and mutable shell. The *functional core* makes decisions. The *mutable shell* supplies input data to the functional core and converts decisions the core makes into side effects.
- The difference between functional and hexagonal architectures is in their treatment of side effects. Functional architecture pushes *all* side effects out of the domain layer. Conversely, hexagonal architecture is fine with side effects made by the domain layer, as long as they are limited to that domain layer only. Functional architecture is hexagonal architecture taken to an extreme.
- The choice between a functional architecture and a more traditional one is a trade-off between performance and code maintainability. Functional architecture concedes performance for maintainability gains.
- Not all code bases are worth converting into functional architecture. Apply functional architecture strategically. Take into account the complexity and the importance of your system. In code bases that are simple or not that important, the initial investment required for functional architecture won't pay off.