6 Test doubles and mocks

This chapter covers

- Using stubs, fakes, and mocks to simplify testing
- Understanding what to mock, when to mock, and when not to mock
- How to mock the unmockable

Until now, we have been testing classes and methods that were isolated from each other. We passed the inputs to a single method call and asserted its output. Or, when a class was involved, we set up the state of the class, called the method under test, and asserted that the class was in the expected state.

But some classes depend on other classes to do their job. Exercising (or testing) many classes together may be desirable. We often break down complex behavior into multiple classes to improve maintainability, each with a small part of the business logic. We still want to ensure, however, that the whole thing works together; we will discuss this in chapter 9. This chapter focuses on testing that unit in an isolated fashion without caring too much about its dependencies. But why would we want that?

The answer is simple: because exercising the class under test together with its concrete dependencies might be too slow, too hard, or too much work. As an example, consider an application that handles invoices. This system has a class called

`IssuedInvoices`, which handles the database and contains lots of SQL queries. Other parts of the system (such as the `InvoiceGenerationService` class, which generates new invoices) depend on this `IssuedInvoices` class to persist the generated invoice in the database. This means that whenever we test `InvoiceGenerationService`, this class will consequently call `IssuedInvoices`, which will then communicate with a database.

In other words, the `InvoiceGenerationService` class indirectly depends on the database that stores the issued invoices. This means testing the `InvoiceGenerationService` requires setting up a database, making sure it contains all the right data, and so on. That is clearly much more work than writing tests that do not require a database. Figure 6.1 shows a more generic illustration of this problem. How do we test a class that depends on many other classes, some of which may involve databases and other complicated things?

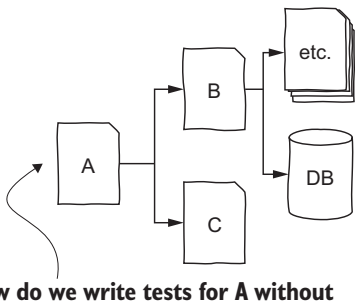


Figure 6.1 A simple illustration of the challenges we face when testing a class that depends on many other classes

But when systematically testing the `InvoiceGenerationService` class, maybe we do not want to test whether the SQL query in the `IssuedInvoices` class is correct. We only want to ensure that, for example, the invoice is generated correctly or contains all the right values. Testing whether the SQL query works will be the responsibility of the `IssuedInvoicesTest` test suite, not `InvoiceGenerationServiceTest`. We will write integration tests for SQL queries in chapter 9.

We must figure out how to test a class that depends on another class without using that dependency. This is where *test doubles* come in handy. We create an object to mimic the behavior of component `B` ("it looks like `B`, but it is not `B`"). Within the test, we have full control over what this fake component `B` does, so we can make it behave as `B` would *in the context of this test* and thus cut the dependency on the real object.

In the previous example, suppose `A` is a plain Java class that depends on `IssuedInvoices` to retrieve values from a database. We can implement a fake `IssuedInvoices` that returns a hard-coded list of values rather than retrieving them from an external database. This means we can control the environment around `A` so we can check how `A` behaves without dealing with complex dependencies. I show examples of how this works later in the chapter.

Using objects that simulate the behavior of other objects has the following advantages:

- *We have more control.* We can easily tell these fake objects what to do. If we want a method to throw an exception, we tell the mock method to throw it. There is no need for complicated setups to force the dependency to throw the exception. Think of how hard it is to force a class to throw an exception or return a fake date. This effort is close to zero when we simulate the dependencies with mock objects.
- *Simulations are faster.* Imagine a dependency that communicates with a web service or a database. A method in one of these classes might take a few seconds to process. On the other hand, if we simulate the dependency, it will no longer need to communicate with a database or web service and wait for a response. The simulation will return what it was configured to return, and it will cost nothing in terms of time.
- When used as a design technique, mocks enable developers to *reflect on how classes should interact with each other*, what their contracts should be, and the conceptual boundaries. Therefore, mocks can be used to make testing easier and support developers in designing code.

NOTE While some of the schools of thought in testing prefer to see mocks as a design technique, in this book, I talk about stubs and mocks mostly from a testing perspective, as our goal is to use mocks to ease our lives when looking for bugs. If you are interested in mocking as a design technique, I strongly recommend Freeman and Pryce's 2009 book, which is the canonical reference for the subject.

I sorted mocks into the unit testing section of my testing flow (go back to figure 1.4 in chapter 1) because our goal is to focus on a single unit without caring much about the other units of the system. Note, however, that we still care about the contracts of the dependencies, as our simulations must follow and do the same things that the simulated class promises.

6.1 Dummies, fakes, stubs, spies, and mocks

Before we dive into how to simulate objects, let's first discuss the different types of simulations we can create. Meszaros, in his book (2007), defines five different types: dummy objects, fake objects, stubs, spies, and mocks. Each makes sense in a specific situation.

6.1.1 Dummy objects

Dummy objects are passed to the class under test but never used. This is common in business applications where you need to fill a long list of parameters, but the test exercises only a few of them. Think of a unit test for a `Customer` class. Maybe this class depends on several other classes like `Address`, `Email`, and so on. Maybe a specific test

case A wants to exercise a behavior, and this behavior does not care which `Address` this `Customer` has. In this case, a tester can set up a dummy `Address` object and pass it to the `Customer` class.

6.1.2 Fake objects

Fake objects have real working implementations of the class they simulate. However, they usually do the same task in a much simpler way. Imagine a fake database class that uses an array list instead of a real database. This fake object is simpler to control than the real database. A common example in real life is to use a simpler database during testing.

In the Java world, developers like to use HSQLDB (HyperSQL database, http://hsqldb.org), an in-memory database that is much faster and easier to set up in the test code than a real database. We will talk more about in-memory databases when we discuss integration testing in chapter 9.

6.1.3 Stubs

Stubs provide hard-coded answers to the calls performed during the test. Unlike fake objects, stubs do not have a working implementation. If the code calls a stubbed method `getAllInvoices`, the stub will return a hard-coded list of invoices.

Stubs are the most popular type of simulation. In most cases, all you need from a dependency is for it to return a value so the method under test can continue its execution. If we were testing a method that depends on this `getAllInvoices` method, we could stub it to return an empty list, then return a list with one element, then return a list with many elements, and so on. This would enable us to assert how the method under test would work for lists of various lengths being returned from the database.

6.1.4 Mocks

Mock objects act like stubs in the sense that you can configure how they reply if a method is called: for example, to return a list of invoices when `getAllInvoices` is called. However, mocks go beyond that. They save all the interactions and allow you to make assertions afterward. For example, maybe we only want the `getAllInvoices` method to be called once. If the method is called twice by the class under test, this is a bug, and the test should fail. At the end of our test, we can write an assertion along the lines of "verify that `getAllInvoices` was called just once."

Mocking frameworks let you assert all sorts of interactions, such as "the method was never called with this specific parameter" or "the method was called twice with parameter A and once with parameter B." Mocks are also popular in industry since they can provide insight into how classes interact.

6.1.5 Spies

As the name suggests, spies spy on a dependency. They wrap themselves around the real object and observe its behavior. Strictly speaking, we are not simulating the object but rather recording all the interactions with the underlying object we are spying on.

Spies are used in very specific contexts, such as when it is much easier to use the real implementation than a mock but you still want to assert how the method under test interacts with the dependency. Spies are less common in the wild.

6.2 An introduction to mocking frameworks

Mocking frameworks are available for virtually all programming languages. While they may differ in their APIs, the underlying idea is the same. Here, I will use Mockito (https://site.mockito.org), one of the most popular stubbing and mocking libraries for Java. Mockito offers a simple API, enabling developers to set up stubs and define expectations in mock objects with just a few lines of code. (Mockito is an extensive framework, and we cover only part of it in this chapter. To learn more, take a look at its documentation.)

Mockito is so simple that knowing the following three methods is often enough:

- `mock(<class>)`—Creates a mock object/stub of a given class. The class can be specified by `<ClassName>.class`.
- `when(<mock>.<method>).thenReturn(<value>)`—A chain of method calls that defines the (stubbed) behavior of the method. In this case `<value>` is returned. For example, to make the `all` method of an `issuedInvoices` mock return a list of invoices, we write `when(issuedInvoices.all()).thenReturn(someListHere)`.
- `verify(<mock>).<method>`—Asserts that the interactions with the mock object happened in the expected way. For example, if we want to ensure that the method `all` of an `issuedInvoices` mock was invoked, we use `verify(issuedInvoices).all()`.

Let's dive into concrete examples to illustrate Mockito's main features and show you how developers use mocking frameworks in practice. If you are already familiar with Mockito, you can skip this section.

6.2.1 Stubbing dependencies

Let's learn how to use Mockito and set up stubs with a practical example. Suppose we have the following requirement:

> The program must return all the issued invoices with values smaller than 100. The collection of invoices can be found in our database. The class `IssuedInvoices` already contains a method that retrieves all the invoices.

The code in listing 6.1 is a possible implementation of this requirement. Note that `IssuedInvoices` is a class responsible for retrieving all the invoices from a real database (for example, MySQL). For now, suppose it has a method `all()` (not shown) that returns all the invoices in the database. The class sends SQL queries to the database and returns invoices. You can check the (naive) implementation in the book's code repository.

Listing 6.1 `InvoiceFilter` class

```
import java.util.List;
import static java.util.stream.Collectors.toList;

public class InvoiceFilter {

  public List<Invoice> lowValueInvoices() {

    DatabaseConnection dbConnection = new DatabaseConnection();
    IssuedInvoices issuedInvoices = new IssuedInvoices(dbConnection);

    try {
      List<Invoice> all = issuedInvoices.all();

      return all.stream()
              .filter(invoice -> invoice.getValue() < 100)
              .collect(toList());
    } finally {
      dbConnection.close();
    }
  }
}
```

Instantiates the IssuedInvoices dependency. It needs a DatabaseConnection, so we also instantiate one of those.

Gets all the invoices from the database

Picks all the invoices with a value smaller than 100

Closes the connection with the database. You would probably handle it better, but this is here to remind you of all the things you need to handle when dealing with databases.

Without stubbing the `IssuedInvoices` class, testing the `InvoiceFilter` class means having to set up a database. It also means having invoices in the database so the SQL query can return them. This is a lot of work, as you can see from the (simplified) test method in listing 6.2, which exercises `InvoiceFilter` together with the concrete `IssuedInvoices` class and the database. Because the tests need a populated database up and running, we first create a connection to the database and clean up any old data it may contain. Then, in the test method, we persist a set of invoices to the database. Finally, when the test is over, we close the connection with the database, as we do not want hanging connections.

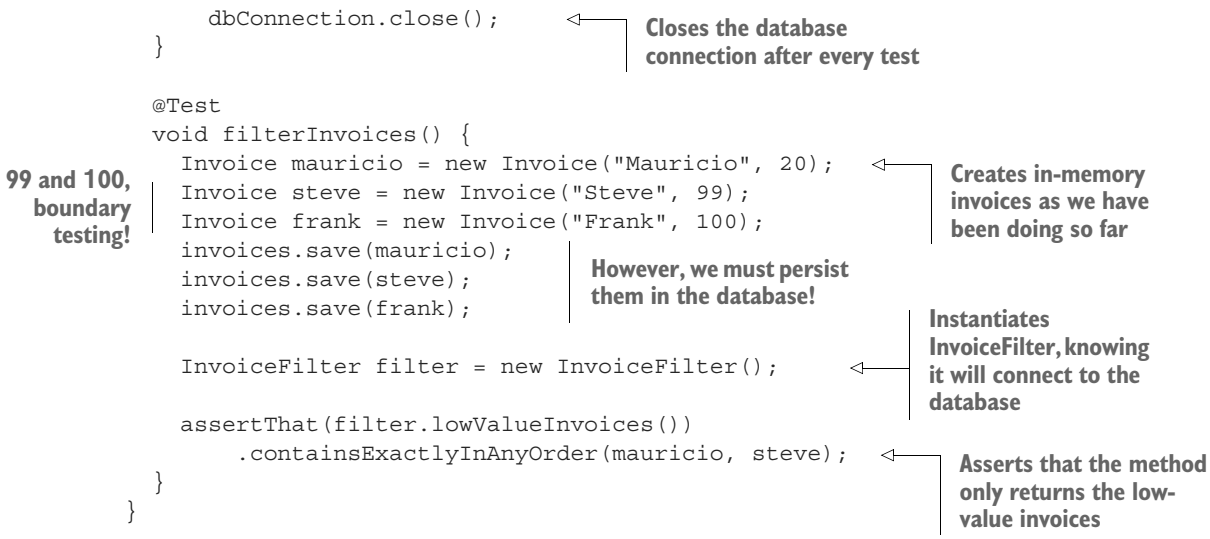
Listing 6.2 Tests for `InvoiceFilter`

```
public class InvoiceFilterTest {
  private IssuedInvoices invoices;
  private DatabaseConnection dbConnection;

  @BeforeEach
  public void open() {
    dbConnection = new DatabaseConnection();
    invoices = new IssuedInvoices(dbConnection);

    dbConnection.resetDatabase();
  }

  @AfterEach
  public void close() {
    if (dbConnection != null)
```

BeforeEach methods are executed before every test method.

Cleans up the tables to make sure old data in the database does not interfere with the test

AfterEach methods are executed after every test method.



NOTE Did you notice the `assertThat…containsExactlyInAnyOrder` assertion? This ensures that the list contains exactly the objects we pass, in any order. Such assertions do not come with JUnit 5. Without AssertJ, we would have to write a lot of code for that assertion to happen. You should get familiar with AssertJ's assertions; they are handy.

This is a small example. Imagine a larger business class with a much more complex database structure. Imagine that instead of persisting a bunch of invoices, you need to persist invoices, customers, items, shopping carts, products, and so on. This can become tedious and expensive.

Let's rewrite the test. This time we will stub the `IssuedInvoices` class and avoid the hassle with the database. First, we need a way to inject the `InvoiceFilter` stub into the class under test. Its current implementation creates an instance of `IssuedInvoices` internally (see the first lines in the `lowValueInvoices` method). This means there is no way for this class to use the stub during the test: whenever this method is invoked, it instantiates the concrete database-dependent class.

We must change our production code to make testing easier (get used to the idea of changing the production code to facilitate testing). The most direct way to do this is to have `IssuedInvoices` passed in as an explicit dependency through the class constructor, as shown in listing 6.3. The class no longer instantiates the `DatabaseConnection` and `IssuedInvoices` classes. Rather, it receives `IssuedInvoices` via constructor. Note that there is no need for the `DatabaseConnection` class to be injected, as `InvoiceFilter` does not need it. This is good: the less we need to do in our test code, the better. The new implementation works for both our tests (because we can inject an `IssueInvoices` stub) and production (because we can inject the concrete `IssueInvoices`, which will go to the database, as we expect in production).

Listing 6.3 `InvoiceFilter` class receiving `IssueInvoices` via constructor

```
public class InvoiceFilter {

  private final IssuedInvoices issuedInvoices;

  public InvoiceFilter(IssuedInvoices issuedInvoices) {
    this.issuedInvoices = issuedInvoices;
  }

  public List<Invoice> lowValueInvoices() {
    List<Invoice> all = issuedInvoices.all();

    return all.stream()
        .filter(invoice -> invoice.getValue() < 100)
        .collect(toList());
  }
}
```

Creates a field in the class to store the dependency

IssuedInvoices is now passed in the constructor.

We no longer instantiate the IssuedInvoices database class. We received it as a dependency, and we use it.

Let's change our focus to the unit test of `InvoiceFilter`. The test is very similar to the one we wrote earlier, but now we do not handle the database. Instead, we configure the `IssuedInvoices` stub as shown in the next listing. Note how easy it is to write this test: full control over the stub enables us to try different cases (even exceptional ones) quickly.

Listing 6.4 Tests for `InvoiceFilter`, stubbing `IssuedInvoices`

```
public class InvoiceFilterTest {

  @Test
  void filterInvoices() {
    IssuedInvoices issuedInvoices = mock(IssuedInvoices.class);

    Invoice mauricio = new Invoice("Mauricio", 20);
    Invoice steve = new Invoice("Steve", 99);
    Invoice frank = new Invoice("Frank", 100);
    List<Invoice> listOfInvoices = Arrays.asList(mauricio, steve, frank);

    when(issuedInvoices.all()).thenReturn(listOfInvoices);

    InvoiceFilter filter = new InvoiceFilter(issuedInvoices);

    assertThat(filter.lowValueInvoices())
        .containsExactlyInAnyOrder(mauricio, steve);
  }
}
```

Instantiates a stub for the IssuedInvoices class, using Mockito's mock method

Creates invoices as we did before

Makes the stub return the predefined list of invoices if all() is called

Instantiates the class under test, and passes the stub as a dependency (instead of the concrete database class)

Asserts that the behavior is as expected

NOTE This idea of classes not instantiating their dependencies by themselves but instead receiving them is a popular design technique. It allows us to inject mocks and also makes the production code more flexible. This idea is also

known as *dependency injection*. If you want to dive into the topic, I suggest *Dependency Injection: Principles, Practices, and Patterns* by Steven van Deursen and Mark Seemann (2019).

Note how we set up the stub using Mockito's `when()` method. In this example, we tell the stub to return a list containing `mauricio`, `frank`, and `steve`, the three invoices we instantiate as part of the test case. The test then invokes the method under test, `filter.lowValueInvoices()`. Consequently, the method under test invokes `issuedInvoices.all()`. However, at this point, `issuedInvoices` is a stub that returns the list with the three invoices. The method under test continues its execution and returns a new list with only the two invoices that are below 100, causing the assertion to pass.

Besides making the test easier to write, stubs also made the test class more cohesive and less prone to change if something other than `InvoiceFilter` changes. If `IssuedInvoices` changes—or, more specifically, if its contracts change—we may have to propagate it to the tests of `InvoiceFilter`, too. Our discussion of contracts in chapter 4 also makes sense when talking about mocks. Now `InvoiceFilterTest` only tests the `InvoiceFilter` class. It does not test the `IssuedInvoices` class. `IssuedInvoices` deserves to be tested, but in another place, using an integration test (which we'll discuss in chapter 9).

A cohesive test also has fewer chances of failing for another reason. In the old version, the `filterInvoices` test could fail because of a bug in the `InvoiceFilter` class or a bug in the `IssuedInvoices` class (imagine a bug in the SQL query that retrieves the invoices from the database). The new tests can only fail because of a bug in `InvoiceFilter`, never because of `IssuedInvoices`. This is handy, as a developer will spend less time debugging if this test fails. Our new approach for testing `InvoiceFilter` is faster, easier to write, and more cohesive.

NOTE This part of the book does not focus on systematic testing. But that is what you should do, regardless of whether you are using mocks. Look at the `filterInvoices` test method. Its goal is to filter invoices that are below 100. In our (currently only) test case, we ensure that this works, and we even exercise the 100 boundary. You may want to exercise other cases, such as empty lists, or lists with a single element, or other test cases that emerge during specification-based and structural testing. I don't do that in this chapter, but you should remember all the techniques discussed in the previous chapters.

In a real software system, the business rule implemented by `InvoiceFilter` would probably be best executed in the database. A simple SQL query would do the job with a much better performance. Try to abstract away from this simple example: whenever you have a dependency that is expensive to use during testing, stubs may come in handy.

6.2.2 Mocks and expectations

Next, let's discuss mocks. Suppose our current system has a new requirement:

> All low-valued invoices should be sent to our SAP system (a software that manages business operations). SAP offers a `sendInvoice` web service that receives invoices.

You know you probably want to test the new class without depending on a real full-blown SAP web service. So, the `SAPInvoiceSender` class (which contains the main logic of the feature) receives, via its constructor, a class that communicates with SAP. For simplicity, suppose there is a `SAP` interface. The `SAPInvoiceSender`'s main method, `sendLowValuedInvoices`, gets all the low-valued invoices using the `InvoiceFilter` class discussed in the previous section and then passes the resulting invoices to SAP.

Listing 6.5 `SAPInvoiceSender` class

```
public interface SAP {
  void send(Invoice invoice);
}

public class SAPInvoiceSender {

  private final InvoiceFilter filter;
  private final SAP sap;

  public SAPInvoiceSender(InvoiceFilter filter, SAP sap) {
    this.filter = filter;
    this.sap = sap;
  }

  public void sendLowValuedInvoices() {
    List<Invoice> lowValuedInvoices = filter.lowValueInvoices();
    for(Invoice invoice : lowValuedInvoices) {
      sap.send(invoice);
    }
  }
}
```

This interface encapsulates the communication with SAP. Note that it does not matter how the concrete implementation will work.

We have fields for both the required dependencies.

The two dependencies are required by the constructor of the class.

The logic of the method is straightforward. We first get the low-value invoices from the InvoiceFilter. Then we pass each of them to SAP.

Let's test the `SAPInvoiceSender` class (see listing 6.6 for the implementation of the test suite). For this test, we stub the `InvoiceFilter` class. For `SAPInvoiceSender`, `InvoiceFilter` is a class that returns a list of invoices. It is not the goal of the current test to test `InvoiceFilter`, so we should stub this class to facilitate testing the method we do want to test. The stub returns a list of low-valued invoices.

The main purpose of this test is to ensure that every low-valued invoice is sent to SAP. How can we assert that this is happening without having the real SAP? It is simple: we ensure that the call to SAP's `send()` method happened. How do we do that?

Mockito, behind the scenes, records all the interactions with its mocks. This means if we mock the `SAP` interface and pass it to the class under test, at the end of the test,

all we need to do is ask the mock whether the method is called. For that, we use Mockito's `verify` assertion (listing 6.6). Note the syntax: we repeat the method we expect to be called. We can even pass the specific parameters we expect. In the case of this test method, we expect the `send` method to be called for both the `mauricio` and `frank` invoices.

Listing 6.6 Tests for the `SAPInvoiceSender` class

```
public class SAPInvoiceSenderTest {

  private InvoiceFilter filter = mock(InvoiceFilter.class);
  private SAP sap = mock(SAP.class);

  private SAPInvoiceSender sender =
    new SAPInvoiceSender(filter, sap);

  @Test
  void sendToSap() {

    Invoice mauricio = new Invoice("Mauricio", 20);
    Invoice frank = new Invoice("Frank", 99);

    List<Invoice> invoices = Arrays.asList(mauricio, frank);

    when(filter.lowValueInvoices()).thenReturn(invoices);

    sender.sendLowValuedInvoices();

    verify(sap).send(mauricio);
    verify(sap).send(frank);
  }
}
```

Passes the mock and the stub to the class under test

Instantiates all the mocks as fields. Nothing changes in terms of behavior. JUnit instantiates a new class before running each of the test methods. This is a matter of taste, but I usually like to have my mocks as fields, so I do not need to instantiate them in every test method.

Sets up the InvoiceFilter stub. It will return two invoices whenever lowValueInvoices() is called.

Calls the method under test, knowing that these two invoices will be sent to SAP

Ensures that the send method was called for both invoices

Again, note how we define the expectations of the mock object. We know exactly how the `InvoiceFilter` class should interact with the mock. When the test is executed, Mockito checks whether these expectations were met and fails the test if they were not.

If you want to see Mockito in action, comment out the call to `sap.send()` in the `sendLowValuedInvoices` method to see the test fail. Mockito will say something like what you see in listing 6.7. Mockito expected the `send` method to be called to the "mauricio" invoice, but it was not. Mockito even complements the message and says that it did not see any interactions with this mock. This is an extra tip to help you debug the failing test.

Listing 6.7 Mockito's verify-failing message

```
Wanted but not invoked:
sap.send(
    Invoice{customer='Mauricio', value=20}
);

Actually, there were zero interactions with this mock.
```

send() was not invoked for this invoice!

This example illustrates the main difference between stubbing and mocking. Stubbing means returning hard-coded values for a given method call. Mocking means not only defining what methods do but also explicitly defining the interactions with the mock.

Mockito enables us to define even more specific expectations. For example, look at the following expectations.

Listing 6.8 More Mockito expectations

```
verify(sap, times(2)).send(any(Invoice.class));
verify(sap, times(1)).send(mauricio);
verify(sap, times(1)).send(frank);
```

Verifies that the send method was called precisely twice for any invoice

Verifies that the send method was called precisely once for the "mauricio" invoice

Verifies that the send method was called precisely once for the "frank" invoice

These expectations are more restrictive than the earlier ones. We now expect the SAP mock to have its `send` method invoked precisely two times (for any given `Invoice`). We then expect the `send` method to be called once for the `mauricio` invoice and once for the `frank` invoice.

Let's write one more test so you become more familiar with Mockito. Let's exercise the case where there are no low-valued invoices. The code is basically the same as in the previous test, but we make our stub return an empty list when the `lowValueInvoices()` method of `InvoiceFilter` is called. We then expect no interactions with the `SAP` mock. That can be accomplished through the `Mockito.never()` and `Mockito.any()` methods in combination with `verify()`.

Listing 6.9 Test for when there are no low-value invoices

```
@Test
void noLowValueInvoices() {
  List<Invoice> invoices = emptyList();
  when(filter.lowValueInvoices()).thenReturn(invoices);

  sender.sendLowValuedInvoices();

  verify(sap, never()).send(any(Invoice.class));
}
```

This time, the stub will return an empty list.

The important part of this test is the assertion. We ensure that the send() method was not invoked for any invoice.

You may wonder why I did not put this new SAP sending functionality in the existing `InvoiceFilter` class. The `lowValueInvoices` method would then be both a command and a query. Mixing both concepts in a single method is not a good idea, as it may confuse developers who call this method. An advantage of separating commands from queries is that, from a mocking perspective, you know what to do. You should stub the queries, as you now know that queries return values and do not change the object's

state; and you should mock commands, as you know they change the world outside the object under test.

> **NOTE** If you want to learn more, search for "command-query separation" (CQS) or read Fowler's wiki entry on CQS (2005). As you get used to testing and writing tests, you will see that the better the code, the easier it is to test it. In chapter 7, we will discuss code decisions you can make in your production code to facilitate testing.

To learn more about the differences between mocks and stubs, see the article "Mocks Aren't Stubs," by Martin Fowler (2007).

6.2.3 Capturing arguments

Imagine a tiny change in the requirements of sending the invoice to the SAP feature:

> Instead of receiving the `Invoice` entity directly, SAP now requires the data to be sent in a different format. SAP requires the customer's name, the value of the invoice, and a generated ID.
>
> The ID should have the following format: <date><customer code>.
>
> - The date should always be in the "MMddyyyy" format: <month><day><year with 4 digits>.
> - The customer code should be the first two characters of the customer's first name. If the customer's name has fewer than two characters, it should be "X".

Implementation-wise, we change the `SAP` interface to receive a new `SapInvoice` entity. This entity has three fields: `customer`, `value`, and `id`. We then modify the `SAPInvoiceSender` so for each low-value invoice, it creates a new `SapInvoice` entity with the correct `id` and sends it to SAP. The next listing contains the new implementation.

Listing 6.10 Changing the SAP-related classes to support the new required format

```
public class SapInvoice {
  private final String customer;
  private final int value;
  private final String id;

  public SapInvoice(String customer, int value, String id) {
    // constructor
  }

  // getters
}

public interface SAP {
  void send(SapInvoice invoice);
}
```

A new entity to represent the new format

SAP receives this new SapInvoice entity.

```
public class SAPInvoiceSender {

  private final InvoiceFilter filter;
  private final SAP sap;

  public SAPInvoiceSender(InvoiceFilter filter, SAP sap) {
    this.filter = filter;
    this.sap = sap;
  }

  public void sendLowValuedInvoices() {
    List<Invoice> lowValuedInvoices = filter.lowValueInvoices();

    for(Invoice invoice : lowValuedInvoices) {
      String customer = invoice.getCustomer();
      int value = invoice.getValue();
      String sapId = generateId(invoice);
      SapInvoice sapInvoice =
        new SapInvoice(customer, value, sapId);

      sap.send(sapInvoice);
    }
  }

  private String generateId(Invoice invoice) {
    String date = LocalDate.now().format(
      ➥ DateTimeFormatter.ofPattern("MMddyyyy"));
    String customer = invoice.getCustomer();

    return date +
      (customer.length()>=2 ? customer.substring(0,2) : "X");
  }
}
```

The constructor is the same as before.

Instantiates the new SAPInvoice object

Sends the new entity to SAP

Generates the required ID as in the requirements

Returns the date plus the customer's code

When it comes to testing, we know that we should stub the `InvoiceFilter` class. We can also mock the `SAP` class and ensure that the `send()` method was called, as shown next.

Listing 6.11 Test for the new implementation of `SAPInvoiceSender`

```
@Test
void sendSapInvoiceToSap() {
  Invoice mauricio = new Invoice("Mauricio", 20);


  List<Invoice> invoices = Arrays.asList(mauricio);
  when(filter.lowValueInvoices()).thenReturn(invoices);

  sender.sendLowValuedInvoices();

  verify(sap).send(any(SapInvoice.class));
}
```

Again, we stub InvoiceFilter.

Asserts that SAP received a SapInvoice. But which SapInvoice? Any. That is not good. We want to be more specific.

This test ensures that the `send` method of the `SAP` is called. But how do we assert that the generated `SapInvoice` is the correct one? For example, how do we ensure that the generated ID is correct?

One idea could be to extract the logic of converting an `Invoice` to a `SapInvoice`, as shown in listing 6.12. The `convert()` method receives an invoice, generates the new id, and returns a `SapInvoice`. A simple class like this could be tested via unit tests without any stubs or mocks. We can instantiate different `Invoices`, call the `convert` method, and assert that the returned `SapInvoice` is correct. I leave that as an exercise for you.

Listing 6.12 Class that converts from `Invoice` to `SapInvoice`

```
public class InvoiceToSapInvoiceConverter {

  public SapInvoice convert(Invoice invoice) {
    String customer = invoice.getCustomer();
    int value = invoice.getValue();
    String sapId = generateId(invoice);

    SapInvoice sapInvoice = new SapInvoice(customer, value, sapId);
    return sapInvoice;
  }

  private String generateId(Invoice invoice) {
    String date = LocalDate.now()
      .format(DateTimeFormatter.ofPattern("MMddyyyy"));
    String customer = invoice.getCustomer();

    return date +
      (customer.length()>=2 ? customer.substring(0,2) : "X");
  }
}
```

This method is straightforward. It does not depend on any complex classes, so we can write unit tests for it as we have done previously.

The same generateId method we saw before

In chapter 10, we further discuss refactorings you can apply to your code to facilitate testing. I strongly recommend doing so. But for the sake of argument, let's suppose this is not a possibility. How can we get the `SapInvoice` object generated in the current implementation of `SAPInvoiceSender` and assert it? This is our chance to use another of Mockito's features: the argument captor.

Mockito allows us to get the specific objects passed to its mocks. We then ask the `SAP` mock to give us the `SapInvoice` passed to it during the execution of the method, to make assertions on it (see listing 6.13). Instead of using `any(SAPInvoice.class)`, we pass an instance of an `ArgumentCaptor`. We then capture its value, which in this case is an instance of `SapInvoice`. We make traditional assertions on the contents of this object.

Listing 6.13 Test using the `ArgumentCaptor` feature of Mockito

```
@ParameterizedTest
@CsvSource({
    "Mauricio,Ma",
    "M,X"}
)
void sendToSapWithTheGeneratedId(String customer, String customerCode) {
  Invoice mauricio = new Invoice(customer, 20);

  List<Invoice> invoices = Arrays.asList(mauricio);
  when(filter.lowValueInvoices()).thenReturn(invoices);

  sender.sendLowValuedInvoices();

  ArgumentCaptor<SapInvoice> captor =
    ArgumentCaptor.forClass(SapInvoice.class);

  verify(sap).send(captor.capture());

  SapInvoice generatedSapInvoice = captor.getValue();

  String date = LocalDate.now().format(DateTimeFormatter.
    ofPattern("MMddyyyy"));
  assertThat(generatedSapInvoice)
    .isEqualTo(new SapInvoice(customer, 20, date + customerCode));
}
```

Passes the two test cases. The test method is executed twice: once for "Mauricio" and once for "M".

Instantiates an ArgumentCaptor with the type of the object we are expecting to capture

Calls the verify method and passes the argument captor as the parameter of the method

The argument was already captured. Now we extract it.

Uses a traditional assertion, ensuring that the ID matches what is expected

Note that we have at least two different test cases to ensure that the generated ID is correct: one where the customer's name is longer than two characters and another where it is shorter than two characters. Given that the structure of the test method would be the same for both methods, I decided to use a parameterized test. I also used the `CsvSource` to pass the different test cases to the test method. The CSV source enables us to pass the inputs via comma-separated values. I usually go for CSV sources whenever the inputs are simple and easily written, as in this case.

Interestingly, although my first option is always to try to refactor the code so I can write simple unit tests, I use argument captors often. In practice, it is common to have such classes, where most of what you do is coordinate the data flow between different components, and objects that need to be asserted may be created on the fly by the method but not returned to the caller.

> **NOTE** There is another test I find fundamental in the `sendToSapWithTheGeneratedId` method: we are missing proper boundary testing. The length of the customer's name (two) is a boundary, so I would test with a customer name that is precisely of length two. Again, we are discussing mocks, but when it comes to designing test cases, all the techniques we have discussed apply.

6.2.4 Simulating exceptions

The developer realizes that SAP's `send` method may throw a `SapException` if a problem occurs. This leads to a new requirement:

> The system should return the list of invoices that failed to be sent to SAP. A failure should not make the program stop. Instead, the program should try to send all the invoices, even though some of them may fail.

One easy way to implement this is to try to catch any possible exceptions. If an exception happens, we store the failed invoice as shown in the following listing.

Listing 6.14 Catching a possible `SAPException`

```
public List<Invoice> sendLowValuedInvoices() {
  List<Invoice> failedInvoices = new ArrayList<>();

  List<Invoice> lowValuedInvoices = filter.lowValueInvoices();
  for(Invoice invoice : lowValuedInvoices) {
    String customer = invoice.getCustomer();
    int value = invoice.getValue();
    String sapId = generateId(invoice);

    SapInvoice sapInvoice = new SapInvoice(customer, value, sapId);

    try {
      sap.send(sapInvoice);
    } catch(SAPException e) {
      failedInvoices.add(invoice);
    }
  }

  return failedInvoices;
}
```

Catches the possible SAPException. If that happens, we store the failed invoice in a list.

Returns the list of failed invoices

How do we test this? By now, you probably see that all we need to do is to force our `sap` mock to throw an exception for one of the invoices. We should use Mockito's `doThrow().when()` chain of calls. This is similar to the `when()` API you already know, but now we want it to throw an exception (see listing 6.15). Note that we configure the mock to throw an exception for the `frank` invoice. Then we assert that the list of failed invoices returned by the new `sendLowValuedInvoices` contains that invoice and that SAP was called for both the `mauricio` and the `frank` invoices. Also, because the `SAP` interface receives a `SapInvoice` and not an `Invoice`, we must instantiate three invoices (Maurício's, Frank's, and Steve's) before asserting that the `send` method was called.

Listing 6.15 Mocks that throw exceptions

```
@Test
void returnFailedInvoices() {
  Invoice mauricio = new Invoice("Mauricio", 20);
  Invoice frank = new Invoice("Frank", 25);
  Invoice steve = new Invoice("Steve", 48);
```

```
  List<Invoice> invoices = Arrays.asList(mauricio, frank, steve);
  when(filter.lowValueInvoices()).thenReturn(invoices);

  String date = LocalDate.now()
    .format(DateTimeFormatter.ofPattern("MMddyyyy"));
  SapInvoice franksInvoice = new SapInvoice("Frank", 25, date + "Fr");
  doThrow(new SAPException())
    .when(sap).send(franksInvoice);

  List<Invoice> failedInvoices = sender.sendLowValuedInvoices();
  assertThat(failedInvoices).containsExactly(frank);

  SapInvoice mauriciosInvoice =
    new SapInvoice("Mauricio", 20, date + "Ma");
  verify(sap).send(mauriciosInvoice);

  SapInvoice stevesInvoice =
    new SapInvoice("Steve", 48, date + "St");
  verify(sap).send(stevesInvoice);
}
```

Configures the mock to throw an exception when it receives Frank's invoice. Note the call to doThrow().when(): this is the first time we use it.

Gets the returned list of failed invoices and ensures that it only has Frank's invoice

Asserts that we tried to send both Maurício's and Steve's invoices

NOTE Creating `SapInvoices` is becoming a pain, given that we always need to get the current date, put it in the MMddyyyy format, and concatenate it with the first two letters of the customer's name. You may want to extract all this logic to a helper method or a helper class. Helper methods are widespread in test code. Remember, test code is as important as production code. All the best practices you follow when implementing your production code should be applied to your test code, too. We will discuss test code quality in chapter 10.

Configuring mocks to throw exceptions enables us to test how our systems would behave in unexpected scenarios. This is perfect for many software systems that interact with external systems, which may not behave as expected. Think of a web service that is not available for a second: would your application behave correctly if this happened? How would you test the program behavior without using mocks or stubs? How would you force the web service API to throw you an exception? Doing so would be harder than telling the mock to throw an exception.

The requirement says one more thing: "A failure should not make the program stop; rather, the program should try to send all the invoices, even though some of them may fail." We also tested that in our test method. We ensured that `steve`'s invoice—the one after `frank`'s invoice, which throws the exception—is sent to SAP.

6.3 Mocks in the real world

Now that you know how to write mocks and stubs and how you can write powerful tests with them, it is time to discuss best practices. As you can imagine, some developers are big fans of mocking. Others believe mocks should not be used. It is a fact that mocks make your tests less real.

When should we mock? When is it best not to mock? What other best practices should I follow? I tackle those questions next.

6.3.1 The disadvantages of mocking

I have been talking a lot about the advantages of mocks. However, as I hinted before, a common (and heated) discussion among practitioners is whether to use mocks. Let's look at possible disadvantages.

Some developers strongly believe that using mocks may lead to test suites that *test the mock, not the code*. That can happen. When you use mocks, you are naturally making your test less realistic. In production, your code will use the concrete implementation of the class you mocked during the test. Something may go wrong in the way the classes communicate in production, for example, and you may miss it because you mocked them.

Consider a class `A` that depends on class `B`. Suppose class `B` offers a method `sum()` that always returns positive numbers (that is, the post-condition of `sum()`). When testing class `A`, the developer decides to mock `B`. Everything seems to work. Months later, a developer changes the post-conditions of `B`'s `sum()`: now it also returns negative numbers. In a common development workflow, a developer would apply these changes in `B` and update `B`'s tests to reflect the change. It is easy to forget to check whether `A` handles this new post-condition well. Even worse, `A`'s test suite will still pass! `A` mocks `B`, and the mock does not know that `B` changed. In large-scale software, it can be easy to lose control of your mocks in the sense that mocks may not represent the real contract of the class.

For mock objects to work well on a large scale, developers must design careful (and hopefully stable) contracts. If contracts are well designed and stable, you do not need to be afraid of mocks. And although we use the example of a contract break as a disadvantage of mocks, it is part of the coder's job to find the dependencies of the contract change and check that the new contract is covered, mocks or not.

Another disadvantage is that tests that use mocks are naturally more coupled with the code they test than tests that do not use mocks. Think of all the tests we have written without mocks. They call a method, and they assert the output. They do not know anything about the actual implementation of the method. Now, think of all the tests we wrote in this chapter. The test methods know some things about the production code. The tests we wrote for `SAPInvoiceSender` know that the class uses `InvoiceFilter`'s `lowValueInvoices` method and that `SAP`'s `send` method must be called for all the invoices. This is a lot of information about the class under test.

What is the problem with the test knowing so much? It may be harder to change. If the developer changes how the `SAPInvoiceSender` class does its job and, say, stops using the `InvoiceFilter` class or uses the same filter differently, the developer may also have to change the tests. The mocks and their expectations may be completely different.

Therefore, although mocks simplify our tests, they increase the coupling between the test and the production code, which may force us to change the test whenever we

change the production code. Spadini and colleagues, including me, observed this through empirical studies in open source systems (2019). *Can you avoid such coupling?* Not really, but at least now you are aware of it.

Interestingly, developers consider this coupling a major drawback of mocks. But I appreciate that my tests break when I change how a class interacts with other classes. The broken tests make me reflect on the changes I am making. Of course, my tests do not break as a result of every minor change I make in my production code. I also do not use mocks in every situation. I believe that when mocks are properly used, the coupling with the production code is not a big deal.

> **Mocking as a design technique**
>
> Whenever I say that mocks increase coupling with production code, I am talking about using mocks from a *testing* perspective: not using mocks as a way to design the code, but in the sense of "This is the code we have: let's test it." In this case, mocks are naturally coupled with the code under test, and changes in the code will impact the mocks.
>
> If you are using mocks as a design technique (as explained in Freeman and Pryce's 2009 book), you should look at it from a different angle. You want your mocks to be coupled with the code under test because you *care* about how the code does its job. If the code changes, you want your mocks to change.

6.3.2 What to mock and what not to mock

Mocks and stubs are useful tools for simplifying the process of writing unit tests. However, *mocking too much* might also be a problem. A test that uses the real dependencies is more real than a test that uses doubles and, consequently, is more prone to find real bugs. Therefore, we do not want to mock a dependency that should not be mocked. Imagine you are testing class `A`, which depends on class `B`. How do we know whether we should mock or stub `B` or whether it is better to use the real, concrete implementation?

Pragmatically, developers often mock or stub the following types of dependencies:

- *Dependencies that are too slow*—If the dependency is too slow for any reason, it might be a good idea to simulate it. We do not want slow test suites. Therefore, I mock classes that deal with databases or web services. Note that I still do integration tests to ensure that these classes work properly, but I use mocks for all the other classes that depend on these slow classes.
- *Dependencies that communicate with external infrastructure*—If the dependency talks to (external) infrastructure, it may be too slow or too complex to set up the required infrastructure. So, I apply the same principle: whenever testing a class that depends on a class that handles external infrastructure, I mock the dependency (as we mocked the `IssuedInvoices` class when testing the `InvoiceFilter` class). I then write integration tests for these classes.

- *Cases that are hard to simulate*—If we want to force the dependency to behave in a hard-to-simulate way, mocks or stubs can help. A common example is when we would like the dependency to throw an exception. Forcing an exception might be tricky when using the real dependency but is easy to do with a stub.

On the other hand, developers tend not to mock or stub the following dependencies:

- *Entities*—Entities are classes that represent business concepts. They consist primarily of data and methods that manipulate this data. Think of the `Invoice` class in this chapter or the `ShoppingCart` class from previous chapters. In business systems, entities commonly depend on other entities. This means, whenever testing an entity, we need to instantiate other entities.

  For example, to test a `ShoppingCart`, we may need to instantiate `Products` and `Items`. One possibility would be to mock the `Product` class when the focus is to test the `ShoppingCart`. However, this is not something I recommend. Entities are classes that are simple to manipulate. Mocking them may require more work. Therefore, I prefer to never mock them. If my test needs three entities, I instantiate them.

  I make exceptions for heavy entities. Some entities require dozens of other entities. Think of a complex `Invoice` class that depends on 10 other entities: `Customer`, `Product`, and so on. Mocking this complex `Invoice` class may be easier.
- *Native libraries and utility methods*—It is also not common to mock or stub libraries that come with our programming language and utility methods. For example, why would we mock `ArrayList` or a call to `String.format`? Unless you have a very good reason, avoid mocking them.
- *Things that are simple enough*—Simple classes may not be worth mocking. If you feel a class is too simple to be mocked, it probably is.

Interestingly, I always followed those rules, because they made sense to me. In 2018–2019, Spadini, myself, and colleagues performed a study to see how developers mock in the wild. Our findings were surprisingly similar to this list.

Let me illustrate with a code example. Consider a `BookStore` class with the following requirement:

> Given a list of books and their respective quantities, the program should return the total price of the cart.
>
> If the bookstore does not have all the requested copies of the book, it includes all the copies it has in stock in the final cart and lets the user know about the missing ones.

The implementation (listing 6.16) uses a `BookRepository` class to check whether the book is available in the store. If not enough copies are available, it keeps track of the unavailable ones in the `Overview` class. For the available books, the store notifies `BuyBookProcess`. In the end, it returns the `Overview` class containing the total amount to be paid and the list of unavailable copies.

Listing 6.16 Implementation of `BookStore`

```
class BookStore {

  private BookRepository bookRepository;
  private BuyBookProcess process;

  public BookStore(BookRepository bookRepository, BuyBookProcess process)
  {
    this.bookRepository = bookRepository;
    this.process = process;
  }

  private void retrieveBook(String ISBN, int amount, Overview overview) {
    Book book = bookRepository.findByISBN(ISBN);

    if (book.getAmount() < amount) {
      overview.addUnavailable(book, amount - book.getAmount());
      amount = book.getAmount();
    }

    overview.addToTotalPrice(amount * book.getPrice());
    process.buyBook(book, amount);
  }

  public Overview getPriceForCart(Map<String, Integer> order) {
    if(order==null)
      return null;

    Overview overview = new Overview();

    for (String ISBN : order.keySet()) {
      retrieveBook(ISBN, order.get(ISBN), overview);
    }

    return overview;
  }
}
```

We know we must mock and stub things, so we inject the dependencies.

Searches for the book using its ISBN

Adds the available copies to the final price

Notifies the buy book process

If the number of copies in stock is less than the number of copies the user wants, we keep track of the missing ones.

Processes each book in the order

Let's discuss the main dependencies of the `BookStore` class:

- The `BookRepository` class is responsible for, among other things, searching for books in the database. This means the concrete implementation of this class sends SQL queries to a database, parses the result, and transforms it into `Book` classes. Using the concrete `BookRepository` implementation in the test might be too painful: we would need to set up the database, ensure that it had the books we wanted persisted, clean the database afterward, and so on. This is a good dependency to mock.
- The `BuyBookProcess` class is responsible for the process of someone buying a book. We do not know exactly what it does, but it sounds complex. `BuyBookProcess` deserves its own test suite, and we do not want to mix that with the `BookStore` tests. This is another good dependency to mock.

- The `Book` class represents a book. The implementation of `BookStore` gets the books that are returned by `BookRepository` and uses that information to know the book's price and how many copies the bookstore has in stock. This is a simple class, and there is no need to mock it since it is easy to instantiate a concrete `Book`.
- The `Overview` class is also a simple, plain old Java object that stores the total price of the cart and the list of unavailable books. Again, there is no need to mock it.
- The `Map<String, Integer>` that the `getPriceForCart` receives as an input is a `Map` object. `Map` and its concrete implementation `HashMap` are part of the Java language. They are simple data structures that also do not need to be mocked.

Now that we have decided what should be mocked and what should not be mocked, we write the tests. The following test exercises the behavior of the program with a more complex order.

Listing 6.17 Test for `BookStore`, only mocking what needs to be mocked

```
@Test
void moreComplexOrder() {
  BookRepository bookRepo = mock(BookRepository.class);
  BuyBookProcess process = mock(BuyBookProcess.class);

  Map<String, Integer> orderMap = new HashMap<>();

  orderMap.put("PRODUCT-ENOUGH-QTY", 5);
  orderMap.put("PRODUCT-PRECISE-QTY", 10);
  orderMap.put("PRODUCT-NOT-ENOUGH", 22);

  Book book1 = new Book("PRODUCT-ENOUGH-QTY", 20, 11); // 11 > 5
  when(bookRepo.findByISBN("PRODUCT-ENOUGH-QTY"))
    .thenReturn(book1);

  Book book2 = new Book("PRODUCT-PRECISE-QTY", 25, 10); // 10 == 10
  when(bookRepo.findByISBN("PRODUCT-PRECISE-QTY"))
    .thenReturn(book2);

  Book book3 = new Book("PRODUCT-NOT-ENOUGH", 37, 21); // 21 < 22
  when(bookRepo.findByISBN("PRODUCT-NOT-ENOUGH"))
    .thenReturn(book3);

  BookStore bookStore = new BookStore(bookRepo, process);
  Overview overview = bookStore.getPriceForCart(orderMap);

  int expectedPrice =
      5*20 + // from the first product
          10*25 + // from the second product
          21*37; // from the third product

  assertThat(overview.getTotalPrice()).isEqualTo(expectedPrice);
```

No need to mock HashMap

As agreed, BookRepository and BuyBookProcess should be mocked.

The order has three books: one where there is enough quantity, one where the available quantity is precisely what is requested in the order, and one where there is not enough quantity.

Stubs the BookRepository to return the three books

Injects the mocks and stubs into BookStore

Ensures that the total price is correct

```
  verify(process).buyBook(book1, 5);
  verify(process).buyBook(book2, 10);
  verify(process).buyBook(book3, 21);

  assertThat(overview.getUnavailable())
      .containsExactly(entry(book3, 1));
}
```

Ensures that BuyBookProcess was called for three books with the right amounts

Ensures that the list of unavailable books contains the one missing book

Could we mock everything? Yes, we could—but doing so would not make sense. You should only stub and mock what is needed. But whenever you mock, you reduce the reality of the test. It is up to you to understand this trade-off.

6.3.3 Date and time wrappers

Software systems often use date and time information. For example, you might need the current date to add a special discount to the customer's shopping cart, or you might need the current time to start a batch processing job. To fully exercise some pieces of code, our tests need to provide different dates and times as input.

Given that date and time operations are common, a best practice is to wrap them into a dedicated class (often called `Clock`). Let's show that using an example:

> The program should give a 15% discount on the total amount of an order if the current date is Christmas. There is no discount on other dates.

A possible implementation for this requirement is shown next.

Listing 6.18 `ChristmasDiscount` implementation

```
public class ChristmasDiscount {

  public double applyDiscount(double amount) {
    LocalDate today = LocalDate.now();

    double discountPercentage = 0;
    boolean isChristmas = today.getMonth() == Month.DECEMBER
      && today.getDayOfMonth() == 25;

    if(isChristmas)
      discountPercentage = 0.15;

    return amount - (amount * discountPercentage);
  }
}
```

Gets the current date. Note the static call.

If it is Christmas, we apply the discount.

The implementation is straightforward; given the characteristics of the class, unit testing seems to be a perfect fit. The question is, how can we write unit tests for it? To test both cases (Christmas/not Christmas), we need to be able to control/stub the `LocalDate` class, so it returns the dates we want. Right now, this is not easy to do, given that the

method makes explicit, direct calls to `LocalDate.now()`. The problem is analogous when `InvoiceFilter` instantiated the `IssuedInvoices` class directly: we could not stub it.

We can then ask a more specific question: how can we stub Java's Time API? In particular, how can we do so for the static method call to `LocalDate.now()`? Mockito allows developers to mock static methods (http://mng.bz/g48n), so we could use this Mockito feature.

Another solution (which is still popular in code bases) is to encapsulate all the date and time logic into a class. In other words, we create a class called `Clock`, and this class handles these operations. The rest of our system only uses this class when it needs dates and times. This new `Clock` class is passed as a dependency to all classes that need it and can therefore be stubbed. The new version of `ChristmasDiscount` is in the following listing.

Listing 6.19 The `Clock` abstraction

```
public class Clock {
  public LocalDate now() {
    return LocalDate.now();
  }

  // any other date and time operation here...
}

public class ChristmasDiscount {

  private final Clock clock;

  public ChristmasDiscount(Clock clock) {
    this.clock = clock;
  }

  public double applyDiscount(double rawAmount) {
    LocalDate today = clock.now();

    double discountPercentage = 0;
    boolean isChristmas = today.getMonth() == Month.DECEMBER
        && today.getDayOfMonth() == 25;

    if(isChristmas)
      discountPercentage = 0.15;

    return rawAmount - (rawAmount * discountPercentage);
  }
}
```

Encapsulates the static call. This seems too simple, but think of other, more complex operations you will encapsulate in this class.

Clock is a plain old dependency that we store in a field and receive via the constructor.

Calls the clock whenever we need, for example, the current date

Testing it should be easy, given that we can stub the `Clock` class (see listing 6.20). We have two tests: one for when it is Christmas (where we set the clock to December 25 of any year) and another for when it is not Christmas (where we set the clock to any other date).

Listing 6.20 Testing the new `ChristmasDiscount`

```
public class ChristmasDiscountTest {
  private final Clock clock = mock(Clock.class);
  private final ChristmasDiscount cd = new ChristmasDiscount(clock);

  @Test
  public void christmas() {
    LocalDate christmas = LocalDate.of(2015, Month.DECEMBER, 25);
    when(clock.now()).thenReturn(christmas);

    double finalValue = cd.applyDiscount(100.0);
    assertThat(finalValue).isCloseTo(85.0, offset(0.001));
  }

  @Test
  public void notChristmas() {
    LocalDate notChristmas = LocalDate.of(2015, Month.DECEMBER, 26);
    when(clock.now()).thenReturn(notChristmas);

    double finalValue = cd.applyDiscount(100.0);
    assertThat(finalValue).isCloseTo(100.0, offset(0.001));
  }
}
```

Clock is a stub.

Stubs the now() method to return the Christmas date

Stubs the now() method. It now returns a date that is not Christmas.

As I said, creating an abstraction on top of date and time operations is common. The idea is that having a class that encapsulates these operations will facilitate the testing of the other classes in the system, because they are no longer handling date and time operations. And because these classes now receive this clock abstraction as a dependency, it can be easily stubbed. Martin Fowler's wiki even has an entry called `ClockWrapper`, which explains the same thing.

Is it a problem to use Mockito's ability to mock static methods? As always, there are no right and wrong answers. If your system does not have complex date and time operations, stubbing them using Mockito's `mockStatic()` API should be fine. Pragmatism always makes sense.

6.3.4 Mocking types you do not own

Mocking frameworks are powerful. They even allow you to mock classes you do not own. For example, we could stub the `LocalDate` class if we wanted to. We can mock any classes from any library our software system uses. The question is, do we want to?

When mocking, it is a best practice to avoid mocking types you do not own. Imagine that your software system uses a library. This library is costly, so you decide to mock it 100% of the time. In the long run, you may face the following complications:

- If this library ever changes (for example, a method stops doing what it was supposed to do), you will not have a breaking test. The entire behavior of that library was mocked. You will only notice it in production. Remember that you want your tests to break whenever something goes wrong.

- It may be difficult to mock external libraries. Think about the library you use to access a database such as Hibernate. Mocking all the API calls to Hibernate is too complex. Your tests will quickly become difficult to maintain.

What is the solution? When you need to mock a type you do not own, you create an abstraction on top of it that encapsulates all the interactions with that type of library. In a way, the `Clock` class we discussed is an example. We do not own the Time API, so we created an abstraction that encapsulates it. These abstractions will let you hide all the complexity of that type, offering a much simpler API to the rest of your software system (which is good for the production code). At the same time, we can easily stub these abstractions.

If the behavior of your class changes, you do not have any failing tests anyway, as your classes depend on the abstraction, not on the real thing. This is not a problem if you apply the right test levels. In all the classes of the system that depend on this abstraction, you can mock or stub the dependency. At this point, a change in the type you do not own will not be caught by the test suite. The abstraction depends on the contracts of the type before it changed. However, the abstraction itself needs to be tested using integration tests. These integration tests will break if the type changes.

Suppose you encapsulate all the behavior of a specific XML parser in an `XmlWriter` class. The abstraction offers a single method: `write(Invoice)`. All the classes of the system that depend on `XmlWriter` have `write` mocked in their unit tests. The `XmlWriter` class, which calls the XML parser, will not mock the library. Rather, it will make calls to the real library and see how it reacts. It will make sure the XML is written as expected. If the library changes, this one test will break. It will then be up to the developer to understand what to do, given the new behavior of the type. See figure 6.2 for an illustration.

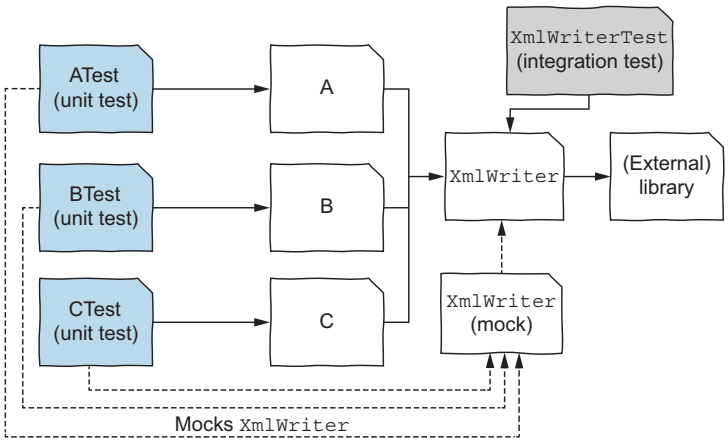


Figure 6.2 `XmlWriter` is mocked when the developer is testing classes that use it (`A`, `B`, and `C`, in the example). `XmlWriter` is then tested via integration tests, exercising the library.

In practice, unit tests are fast and easy to write and do not depend on external libraries. Integration tests ensure that the interaction with the library happens as expected, and they capture any changes in the behavior.

Creating abstractions on top of dependencies that you do not own, as a way to gain more control, is a common technique among developers. (The idea of only mocking types you own was suggested by Freeman et al. in the paper that introduced the concept of mock objects [2004] and by Mockito.) Doing so increases the overall complexity of the system and requires maintaining another abstraction. But does the ease in testing the system that we get from adding the abstraction compensate for the cost of the increased complexity? Often, the answer is yes: it does pay off.

6.3.5 What do others say about mocking?

As I said, some developers favor mocking, and others do not. *Software Engineering at Google*, edited by Winters, Manshreck, and Wright (2020), has an entire chapter dedicated to test doubles. Here's what I understood from it, along with my own point of view:

- *Using test doubles requires the system to be designed for testability.* Indeed, as we saw, if you use mocks, you need to make sure the class under test can receive the mock.
- *Building test doubles faithful to the real implementation is challenging. Test doubles must be as faithful as possible.* If your mocks do not offer the same contracts and expectations of the production class, your tests may all pass, but the software system will fail in production. Whenever you are mocking, make sure your mocks faithfully represent the class you are mocking.
- *Prefer realism over isolation. When possible, opt for the real implementation instead of fakes, stubs, or mocks.* I fully agree with this. Although I did my best to convince you about the usefulness of mocking (that was the point of this chapter), realism always wins over isolation. I am pragmatic about it, though. If it is getting too hard to test with the real dependency, I mock it.
- The following are trade-offs to consider when deciding whether to use a test double:
  - *The execution time of the real implementation*—I also take the execution time of the dependency into account when deciding to mock or not. I usually mock slow dependencies.
  - *How much non-determinism we would get from using the real implementation*—While I did not discuss non-deterministic behavior, dependencies that present such behavior may be good candidates for mocking.
- *When using the real implementation is not possible or too costly, prefer fakes over mocks.* I do not fully agree with this recommendation. In my opinion, you either use the real implementation or mock it. A fake implementation may end up having the same problems as a mock. How do you ensure that the fake implementation has the same behavior as the real implementation? I rarely use fakes.
- *Excessive mocking can be dangerous, as tests become unclear (hard to comprehend), brittle (may break too often), and less effective (reduced ability to detect faults).* I agree. If you are mocking too much or the class under test forces you to mock too much, that may be a sign that the production class is poorly designed.

- *When mocking, prefer state testing rather than interaction testing.* Google says you should make sure you are asserting a change of state and/or the consequence of the action under test, rather than the precise interaction that the action has with the mocked object. Google's point is similar to what we discussed about mocks and coupling. Interaction testing tends to be too coupled with the implementation of the system under test.

  While I agree with this point, properly written interaction tests are useful. They tell you when the interaction changed. This is my rule of thumb: if what matters in the class I am testing is the interaction between classes, I do interaction testing (my assertions check that the interactions are as expected). When what matters is the result of processing, I do state testing (my assertions check the return value or whether the state of the class is as expected).
- *Avoid over-specified interaction tests. Focus on the relevant arguments and functions.* This is a good suggestion and best practice. Make sure you only mock and stub what needs to be mocked and stubbed. Only verify the interactions that make sense for that test. Do not verify every single interaction that happens.
- *Good interaction testing requires strict guidelines when designing the system under test. Google engineers tend not to do this.* Using mocks properly is challenging even for senior developers. Focus on training and team education, and help your developer peers do better interaction testing.

Exercises

6.1 Mocks, stubs, and fakes. What are they, and how do they differ from each other?

6.2 The following `InvoiceFilter` class is responsible for returning the invoices for an amount smaller than 100.0. It uses the `IssuedInvoices` type, which is responsible for communication with the database.

```
public class InvoiceFilter {

  private IssuedInvoices invoices;

  public InvoiceFilter(IssuedInvoices invoices) {
    this.invoices = invoices;
  }

  public List<Invoice> filter() {
    return invoices.all().stream()
        .filter(invoice -> invoice.getValue() < 100.0)
        .collect(toList());
  }
}
```

Which of the following statements about this class is false?

A Integration testing is the only way to achieve 100% branch coverage.

B Its implementation allows for dependency injection, which enables mocking.

C It is possible to write completely isolated unit tests by, for example, using mocks.

D The `IssuedInvoices` type (a direct dependency of `InvoiceFilter`) should be tested using integration tests.

6.3 You are testing a system that triggers advanced events based on complex combinations of external, boolean conditions relating to the weather (outside temperature, amount of rain, wind, and so on). The system has been designed cleanly and consists of a set of cooperating classes, each of which has a single responsibility. You use specification-based testing for this logic and test it using mocks.

Which of the following is a valid test strategy?

A You use mocks to support observing the external conditions.

B You create mock objects to represent each variant you need to test.

C You use mocks to control the external conditions and to observe the event being triggered.

D You use mocks to control the triggered events.

6.4 Class `A` depends on a static method in class `B`. If you want to test class `A`, which of the following two actions should you apply to do so properly?

Approach 1: Mock class `B` to control the behavior of the methods in class `B`.

Approach 2: Refactor class `A`, so the outcome of the method of class `B` is now used as a parameter.

A Only approach 1

B Neither

C Only approach 2

D Both

6.5 According to the guidelines provided in the book, what types of classes should you mock, and which should you not mock?

6.6 Now that you know the advantages and disadvantages of test doubles, what are your thoughts about them? Do you plan to use mocks and stubs, or do you prefer to focus on integration tests?

Summary

- Test doubles help us test classes that depend on slow, complex, or external components that are hard to control and observe.
- There are different types of test doubles. Stubs are doubles that return hard-coded values whenever methods are called. Mocks are like stubs, but we can define how we expect a mock to interact with other classes.
- Mocking can help us in testing, but it also has disadvantages. The mock may differ from the real implementation, and that would cause our tests to pass while the system would fail.

- Tests that use mocks are more coupled with the production code than tests that do not use mocks. When not carefully planned, such coupling can be problematic.
- Production classes should allow for the mock to be injected. One common approach is to require all dependencies via the constructor.
- You do not have to (and should not) mock everything, even when you decide to go for mocks. Only mock what is necessary.