15 Testing your Spring app

This chapter covers

- Why testing apps is important
- How tests work
- Implementing unit tests for Spring apps
- Implementing Spring integration tests

In this chapter, you'll learn to implement tests for your Spring apps. A test is a small piece of logic whose purpose is to validate that a specific capability your app implements works as expected. We'll classify the tests into two categories:

- *Unit tests*—Focus only on an isolated piece of logic
- *Integration tests*—Focus on validating that multiple components correctly interact with each other

But when I only use the term "test," I refer to both these categories.

Tests are essential for any application. They ensure that the changes we make during the app's development process don't break existing capabilities (or at least they make errors less likely) and also serve as documentation. Many developers (unfortunately) disregard tests because they are not directly part of the app's business logic, and, of course, it takes some time to write them. Because of this, tests seem to not have a significant impact. Indeed, their impact is not usually visible in the short term, but trust me, tests are invaluable in the long term. I can't stress enough how important it is to make sure you properly test your app's logic.

Why should you write a test instead of relying on manually testing a capability?

- Because you can run that test over and over again to validate things are work-ing as expected with minimum effort (validates the app behaves correctly continuously)
- Because by reading the test steps you can easily understand the use-case purpose (serves as documentation)
- Because tests provide early feedback about new application issues during the development process

Why wouldn't the app's capabilities work a second time if they initially worked?

- Because we continuously change the app's source code to fix bugs or add new features. When you change the source code, you might break previously implemented capabilities.

If you write tests for those capabilities, you can run them any time you change the app to validate things are still working as expected. If you affected some existing functionality, you'd find out what happened before delivering your code to production. *Regression testing* is the approach of constantly testing existing functionality to validate it still works correctly.

A good approach is making sure you test all the relevant scenarios for any specific capability you implement. You can then run the tests any time you change something to validate the previously implemented capabilities were not affected by your changes.

Today, we don't rely only on developers running the tests manually, but we make their execution part of the app's build process. Generally, development teams use what we call a *continuous integration* (CI) approach: they configure a tool such as Jenkins or TeamCity to run a build process every time a developer makes changes. A continuous integration tool is software we use to execute the steps needed to build and sometimes install the apps we implement during the development process. This CI tool also runs the tests and notifies the developers if something has been broken (figure 15.1).

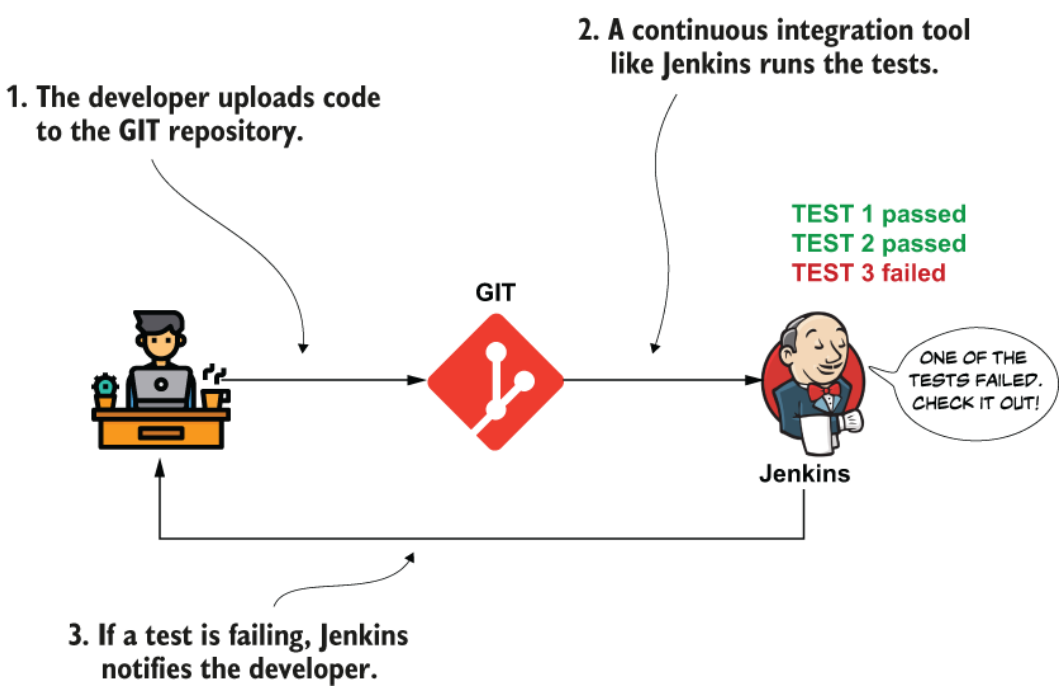

Figure 15.1 A CI tool, such as Jenkins or TeamCity, runs the tests every time a developer changes the app. If any of the tests fail, the CI tool notifies the developers to check which capability doesn't work as expected and correct the problem.

In section 15.1, we start by drawing a big picture of what a unit test is and how it works. In section 15.2, we discuss the two most encountered types of tests you'll find used with Spring apps: unit and integration tests. We'll take examples of capabilities we implemented throughout this book and implement tests for them.

Before diving deep into this chapter, I'd like to make you aware that testing is a complex subject, and we'll focus only on the essential knowledge you need to have when testing Spring apps. But testing is a subject that deserves its own bookshelf. I recommend you also read the book *JUnit in Action* (Manning, 2020) by Cătălin Tudose, which reveals more testing jams that you'll find valuable.

15.1 Writing correctly implemented tests

In this section, we discuss how tests work and what a correctly implemented test is. You'll learn how to write your app's code to make it easy to test, and you'll observe that there's a strong connection between making the app testable and making it maintainable (i.e., easy to change to implement new features and correct errors). Testability and maintainability are software qualities that help one another. By designing your app to be testable, you also help to make it maintainable.

We write tests to validate the logic implemented by a specific method in the project works in the desired way. When you test a given method, usually you need to validate multiple scenarios (ways in which the app behaves depending on different inputs). For each scenario, you write a test method in a test class. In a Maven project (such as the examples we implemented throughout the book), you write the test classes in the project's test folder (figure 15.2).

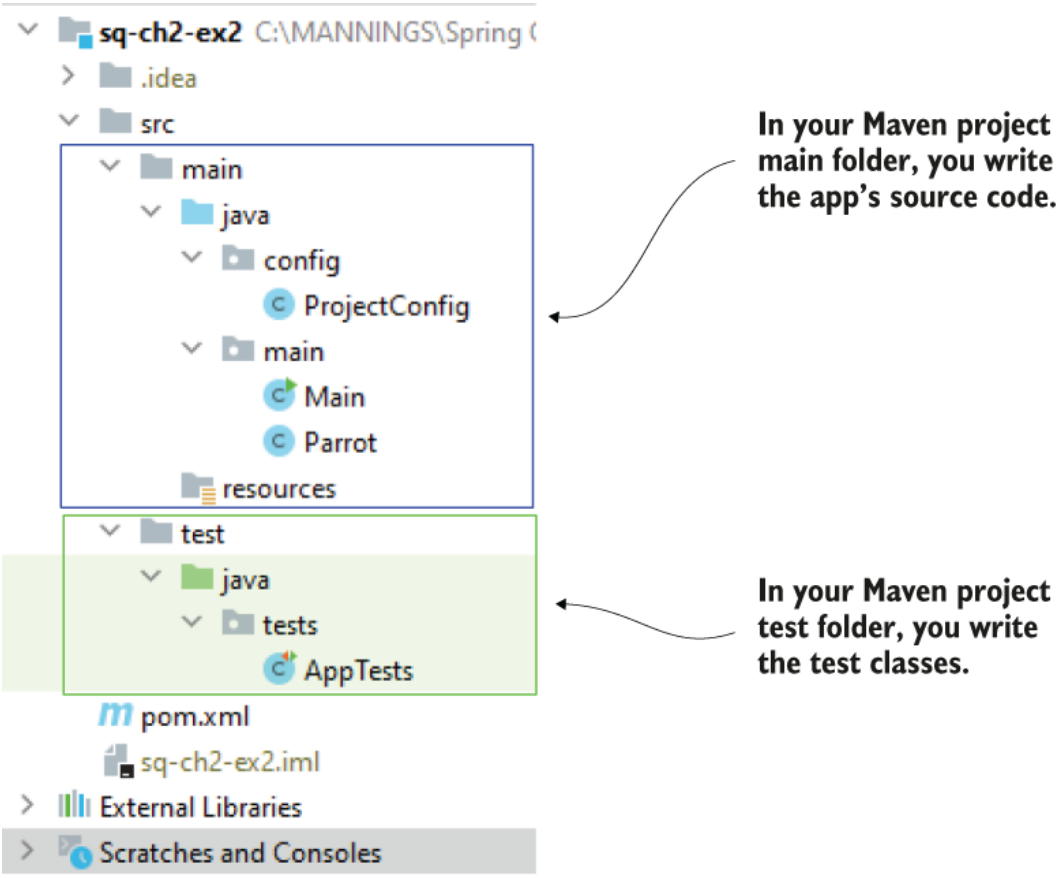

Figure 15.2 In a Maven project, you write the test classes in the project's test folder.

A test class should focus only on a particular method whose logic you test. Even simple logic generates various scenarios. For each scenario, you'll write a method in the test class that validates that specific case.

Let's take an example. Remember the money transfer use case we discussed in chapters 13 and 14? This was our simple implementation of transferring a given amount between two different accounts. The use case had only four steps:

1. Find the source account details in the database.
2. Find the destination account details in the database.
3. Calculate the new amounts for the two accounts after the transfer.
4. Update the accounts' amount values in the database.

Even with only these steps, we can still find multiple scenarios relevant for testing:

1. Test what happens if the app can't find the source account details.
2. Test what happens if the app can't find the destination account details.
3. Test what happens if the source account doesn't have enough money.
4. Test what happens if the amounts update fails.
5. Test what happens if all the steps work fine.

For each test scenario, you need to understand how the app should behave and write a test method to validate it works as desired. For example, if for test case 3, you don't want to allow a transfer to happen if the source account doesn't have enough money, you'll test that the app throws a specific exception and the transfer doesn't happen. But depending on the app's requirements, you could allow a defined credit limit for the source account. In such a case, your test needs to take this limit into consideration as well.

The test scenario implementation is strongly related to how the app should work, but technically, the idea is the same in any app: you identify the test scenarios, and you write a test method for each (figure 15.3).

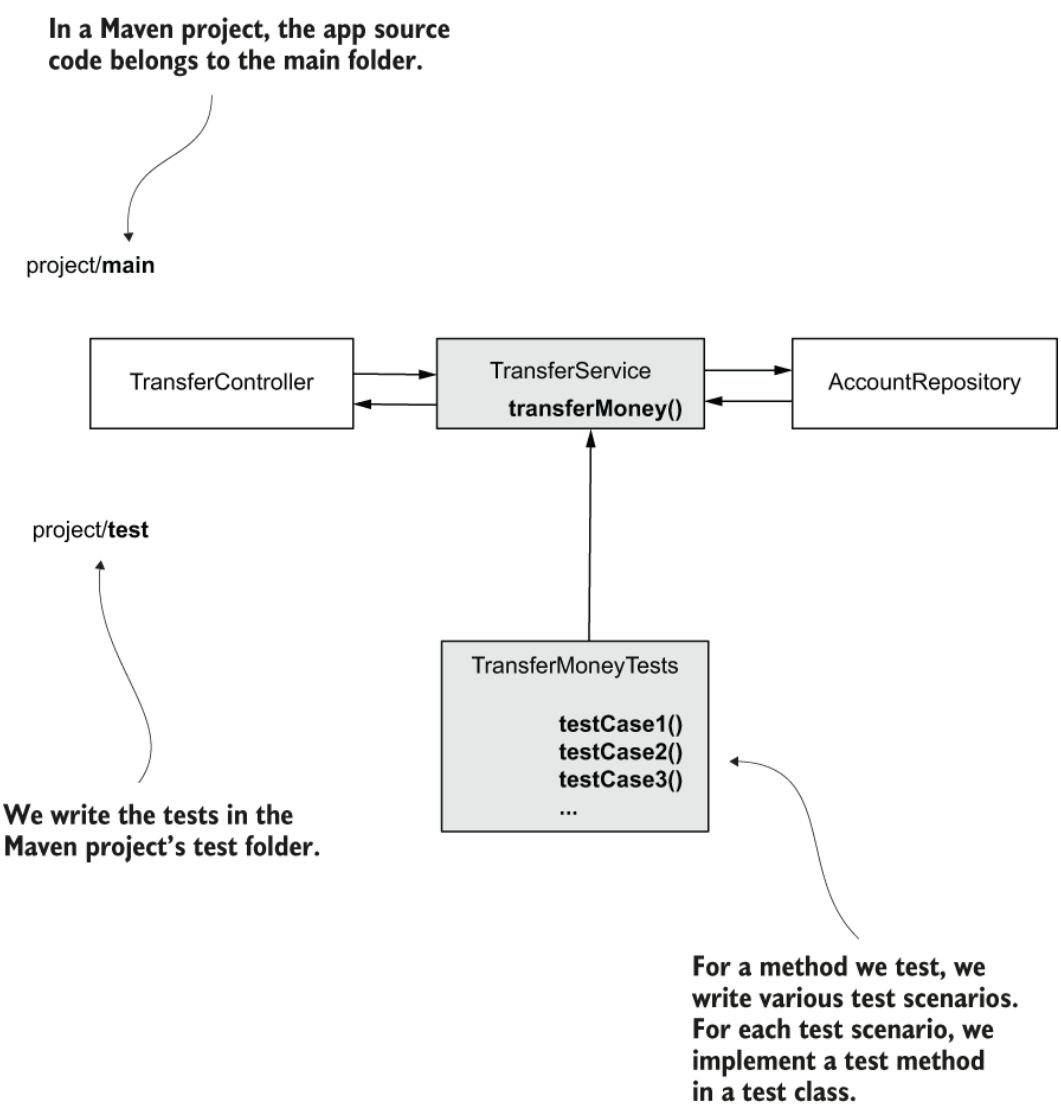

Figure 15.3 For any piece of logic you test, you need to find the relevant test scenarios. For each test scenario, you write a test method in a test class. You add the test classes in the app's Maven project test folder. In this figure, the `TransferMoneyTests` class is a test class that contains the test scenarios for the `transferMoney()` method. `TransferMoneyTests` defines multiple test case methods to test each relevant scenario in the `transferMoney()`'s method logic.

A critical thing to observe is that we can find multiple relevant test scenarios, even for a small method—another reason to keep the methods in your application small! If you write large methods with many code lines and parameters that focus on multiple things simultaneously, identifying the relevant test scenario becomes extremely difficult. We say that the app's testability decreases when you fail to separate the different responsibilities into small and easy-to-read methods.

15.2 Implementing tests in Spring apps

In this section, we use two testing techniques for Spring apps you often encounter in real-world projects. We'll demonstrate each technique by

considering a use case we implemented in the previous chapters and write the tests for it. These techniques are (in my perspective) a must-know for any developer:

- *Writing unit tests to validate a method's logic*. The unit tests are short, fast to execute, and focus on only one flow. These tests are a way to focus on validating a small piece of logic by eliminating all the dependencies.
- *Writing Spring integration tests to validate a method's logic and its integration with specific capabilities the framework provides*. These tests help you make sure your app's capabilities still work when you upgrade dependencies.

In section 15.2.1, you'll learn about unit tests. We'll discuss why unit tests are important and the steps you consider when writing a unit test, and we'll write a couple of unit tests as examples for use cases we implemented in the previous chapters. In section 15.2.2, you'll learn to implement integration tests, how these are different from unit tests, and how they complement the unit tests in a Spring app.

15.2.1 Implementing unit tests

In this section, we discuss unit tests. Unit tests are methods that call a certain use case in specific conditions to validate behavior. The unit test method defines the conditions in which the use case executes and validates the behavior defined by the app's requirements. They eliminate all the dependencies of the capability they test, covering only a specific, isolated piece of logic.

Unit tests are valuable because when one fails, you know something is wrong with a particular piece of code, and you're shown exactly where you need to correct it. A unit test is like one of your car's dashboard indicators. If you try starting your car and it fails to start, it might be because you ran out of gas or your battery isn't working properly. A car is a complex system (same as an app), and you have no clue what the problem is unless you have an indicator. If the car's indicator shows you're out of gas, then you immediately identified the problem!

Unit tests' purpose is to validate a single unit of logic's behavior, and like a car's indicators, they help you identify problems in a specific component.

Let's look at one of the use cases we wrote in chapter 14: the money transfer use case. The steps in this piece of logic are as follows:

1. Find the sender's account details.
2. Find the destination account details.
3. Calculate the new amounts for each account.
4. Update the sender's account amount.
5. Update the destination account amount.

The following listing shows you the use case implementation as we worked it in the project "sq-ch14-ex1."

Listing 15.1 The implementation of the money transfer use case

```
@Transactional
public void transferMoney(
  long idSender,
  long idReceiver,
  BigDecimal amount) {

  Account sender = accountRepository.findById(idSender)
    .orElseThrow(() -> new AccountNotFoundException());

  Account receiver = accountRepository.findById(idReceiver)
    .orElseThrow(() -> new AccountNotFoundException());

  BigDecimal senderNewAmount = sender.getAmount().subtract(amount);
  BigDecimal receiverNewAmount = receiver.getAmount().add(amount);

  accountRepository
    .changeAmount(idSender, senderNewAmount);
  accountRepository
    .changeAmount(idReceiver, receiverNewAmount);
}
```

❶ We find the details of the sender's account.

❷ We find the details of the destination account.

❸ We calculate the accounts' amounts.

❹ We update the new amount in the sender account.

❺ We update the new amount in the destination account.

Usually, the most obvious scenarios and the first we write tests for are the *happy flows*: an execution that encountered no exceptions or errors. Figure 15.4 visually represents the happy flow of our money transfer use case.

Figure 15.4 The happy flow: an execution for which no errors or exceptions are encountered. Usually, the happy flows are the first to write tests because they are the most obvious scenarios.

Let's write a unit test for this happy flow of the money transfer use case. Any test has three main parts (figure 15.5):

1. *Assumptions*—We need to define any input and find any dependency of the logic we need to control to achieve the desired flow scenario. For this point, we'll answer the following questions: what inputs should we provide, and how should dependencies behave for the tested logic to act in the specific way we want?
2. *Call/Execution*—We need to call the logic we test to validate its behavior.
3. *Validations*—We need to define all the validations that need to be done for the given piece of logic. We'll answer this question: what should happen when this piece of logic is called in the given conditions?

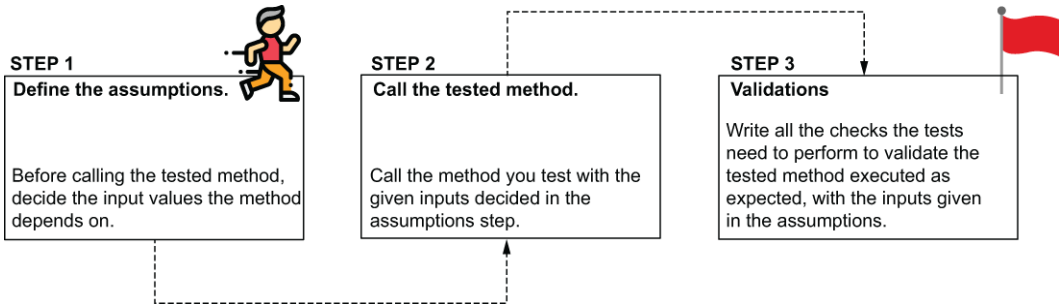

Figure 15.5 The steps for writing a unit test. Write the assumptions by defining the method inputs. Call the method with the defined assumptions and write the checks the tests need to perform to validate the method's behavior is correct.

NOTE Sometimes, you'll find these three steps (assumptions, call, and validations) named a bit differently: "arrange, act, and assert" or "given, when, and then." Regardless of how you prefer to name them, the idea of how you write the tests stays the same.

In the test's *assumptions*, we identify the dependencies for the test case we write the test for. We choose the inputs and how the dependencies behave to make the tested logic act in a certain way.

Which are the dependencies for the money transfer use case? Dependencies are anything the method uses but doesn't create itself:

- The method's parameters
- Object instances the method uses but that are not created by it

In figure 15.6, we identify these dependencies for our example.

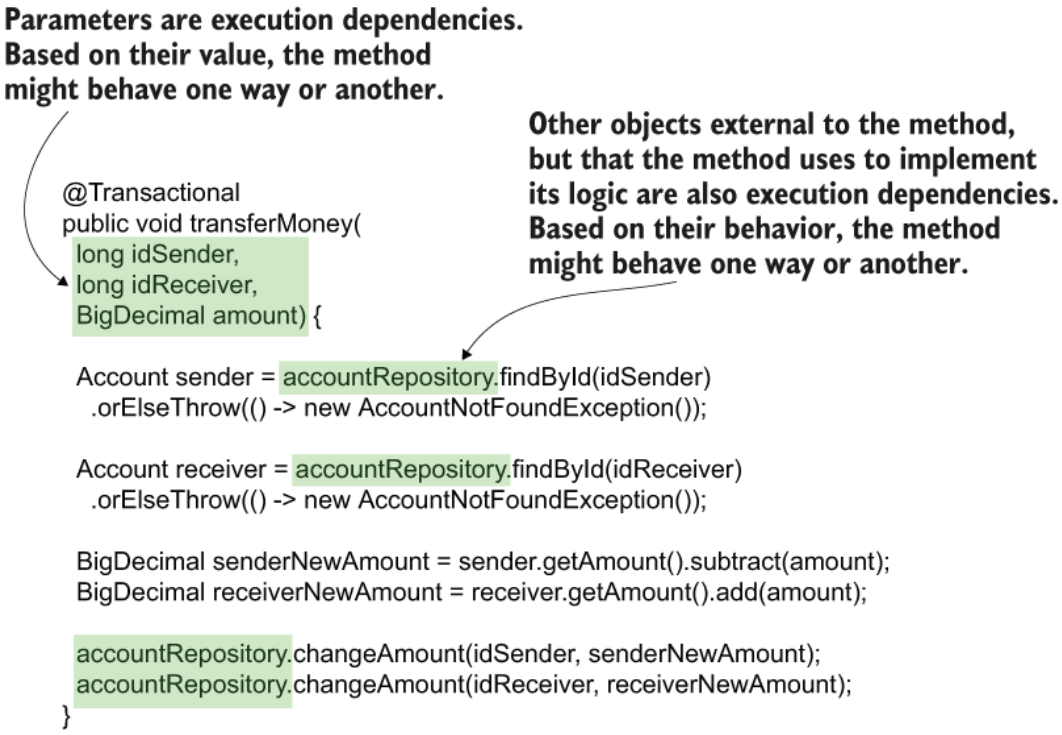

Figure 15.6 A unit test validates a use case logic in isolation from any dependency. To write the test, we need to make sure we know the dependencies and how to control them. For our scenario, the parameters and the `AccountRepository` object are dependencies we need to control for the test.

When we call the method to test it, we can provide any values for its three parameters to control the execution flow. But the `AccountRepository` instance is a bit more complicated. The `transferMoney()` method execution depends on how the `findById()` method of the `AccountRepository` instance behaves.

But remember, a unit test focuses on only one piece of logic, so it should not call the `findById()` method. The unit test should assume `findById()` works in a given way and assert that the tested method's execution does what's expected for the given situation.

But the tested method calls `findById()`. How could we control it? To control such a dependency, we use *mocks*: a fake object whose behavior we can control. In this case, instead of using the real `AccountRepository` object, we'll make sure the tested method uses this fake object. We'll take advantage of controlling how this fake object behaves to induce all the different executions of the `transferMoney()` method that we want to test.

Figure 15.7 shows you what we want to do. We replace the `AccountRepository` object with a mock to eliminate the tested object's dependency.

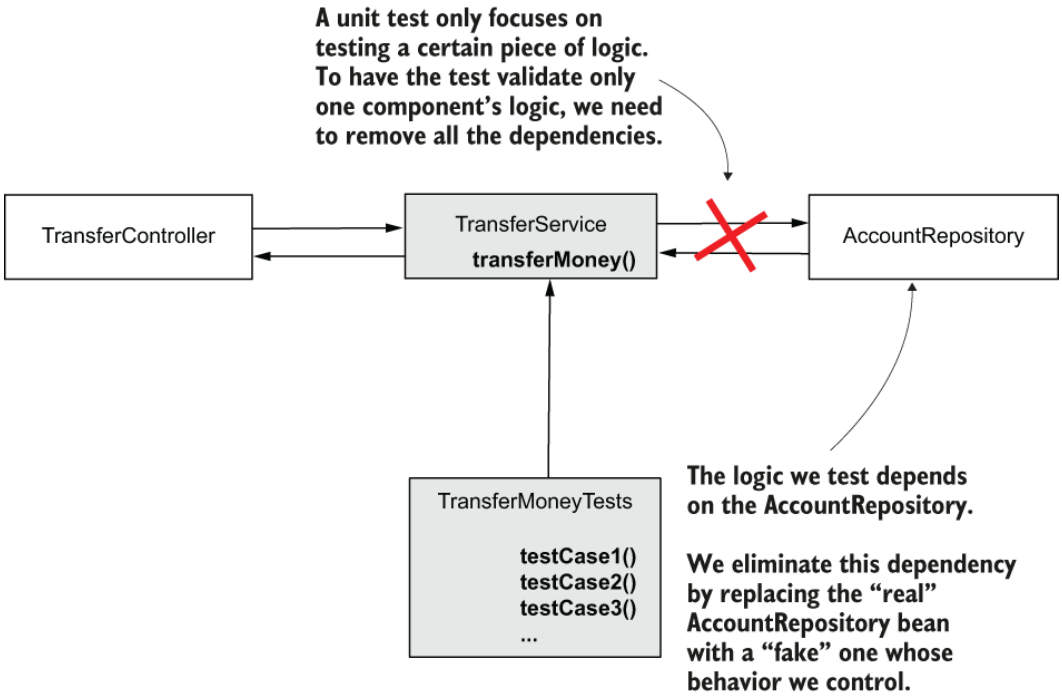

Figure 15.7 To allow the unit test to focus only on the `transferMoney()` method's logic, we eliminate the dependency to the `AccountRepository` object. We use a mock object to replace the real `AccountRepository` instance, and we control this fake instance to test how the `transferMoney()` method behaves in different situations.

In listing 15.2, we start implementing the unit test. After creating a new class in the Maven's project test folder, we start implementing the first test scenario by writing a new method we annotate with the `@Test` annotation.

NOTE For the examples in this book, we use JUnit 5 Jupiter, the latest JUnit version, to implement the unit and integration tests. However, in real-world apps, you might also find JUnit 4 used often. This is one more reason I recommend you also read books that focus on testing. Chapter 4

of *JUnit in Action* (Manning, 2020) by Cătălin Tudose focuses on the differences between JUnit 4 and JUnit 5.

We create a `TransferService` instance to call the `transferMoney()` method that we want to test. Instead of using a real `AccountRepository` instance, we create a mock object that we can control. To create such a mock object, we use a method named `mock()`. This `mock()` method is provided by a dependency named Mockito (often used with JUnit to implement tests).

Listing 15.2 Creating the object whose method we want to unit test

```
public class TransferServiceUnitTests {

  @Test
  public void moneyTransferHappyFlow() {
    AccountRepository accountRepository =
      mock(AccountRepository.class);              ❶

    TransferService transferService =             ❷
      new TransferService(accountRepository);
  }
}
```

❶ We use the Mockito mock() method to create a mock instance for the AccountRepository object.

❷ We create an instance of the TransferService object whose method we want to test. Instead of a real AccountRepository instance, we create the object using a mock AccountRepository. This way, we replace the dependency with something we can control.

We can now specify how the mock object should behave, then call the tested method and prove it works as expected in the given conditions. You control the mock's behavior using the `given()` method, as shown in listing 15.3. Using the `given()` method, you tell the mock how to behave when one of its methods is called. In our case, we want the `AccountRepository`'s `findById()` method to return a specific `Account` instance for a given parameter value.

NOTE In a real-world app, a good practice is using the `@DisplayName` annotation to describe the test scenario (as you see in the next listing). In our examples, I took out the `@DisplayName` annotation to save space and allow you focus on the test logic. However, using it in a real-world app

can help you, but also other developers on the team, better understand the implemented test scenario..

Listing 15.3 A unit test validating the happy flow

```
public class TransferServiceUnitTests {

  @Test
  @DisplayName("Test the amount is transferred " +
   "from one account to another if no exception occurs.")
  public void moneyTransferHappyFlow() {
    AccountRepository accountRepository =
      mock(AccountRepository.class);
    TransferService transferService =
      new TransferService(accountRepository);

    Account sender = new Account();                          ❶
    sender.setId(1);
    sender.setAmount(new BigDecimal(1000));

    Account destination = new Account();                     ❶
    destination.setId(2);
    destination.setAmount(new BigDecimal(1000));

    given(accountRepository.findById(sender.getId()))        ❷
      .willReturn(Optional.of(sender));

    given(accountRepository.findById(destination.getId()))   ❸
      .willReturn(Optional.of(destination));

    transferService.transferMoney(                           ❹
                       sender.getId(),
                       destination.getId(),
                       new BigDecimal(100)
                     );

  }

}
```

❶ We create the sender and the destination Account instances, which hold the Account details, which we assume the app would find in the database.

❷ We control the mock's findById() method to return the sender account instance when it gets the sender account ID. You can read this line as "If

one calls the findById() method with the sender ID parameter, then return the sender account instance."

❸ We control the mock's findById() method to return the destination account instance when it gets the destination account ID. You can read this line as "If one calls the findById() method with the destination ID parameter, then return the destination account instance."

❹ We call the method we want to test with the sender ID, destination ID, and the value to be transferred.

The only thing we still need to do is tell the test what should happen when the tested method executes. What do we expect? We know this method's purpose is to transfer the money from one given account to another. So, we expect that it calls the repository instance to change the amounts with the right values. In listing 15.4, we add the test instructions that verify the method correctly called the repository instance's methods to change the amounts.

To verify a mock's object's method has been called, you use the `verify()` method, as presented in the following listing.

Listing 15.4 A unit test validating the happy flow

```
public class TransferServiceUnitTests {

  @Test
  public void moneyTransferHappyFlow() {
    AccountRepository accountRepository =
      mock(AccountRepository.class);
    TransferService transferService =
      new TransferService(accountRepository);

    Account sender = new Account();
    sender.setId(1);
    sender.setAmount(new BigDecimal(1000));

    Account destination = new Account();
    destination.setId(2);
    destination.setAmount(new BigDecimal(1000));

    given(accountRepository.findById(sender.getId()))
      .willReturn(Optional.of(sender));

    given(accountRepository.findById(destination.getId()))
```

```
        .willReturn(Optional.of(destination));

      transferService.transferMoney(
                          sender.getId(),
                          destination.getId(),
                          new BigDecimal(100)
                        );

      verify(accountRepository)                    ❶
        .changeAmount(1, new BigDecimal(900));     ❶

      verify(accountRepository)                    ❶
        .changeAmount(2, new BigDecimal(1100));    ❶


    }

  }
```

❶ Verify that the changeAmount() method in the AccountRepository was called with the expected parameters.

If you run the test now (usually in an IDE by right-clicking on the test class and selecting the “Run tests” option), you should observe the tests succeed. When a test succeeds, the IDE displays them in green, and the console doesn’t show any exception message. If a test fails, it’s usually displayed in either red or yellow in an IDE (fig-ure 15.8).

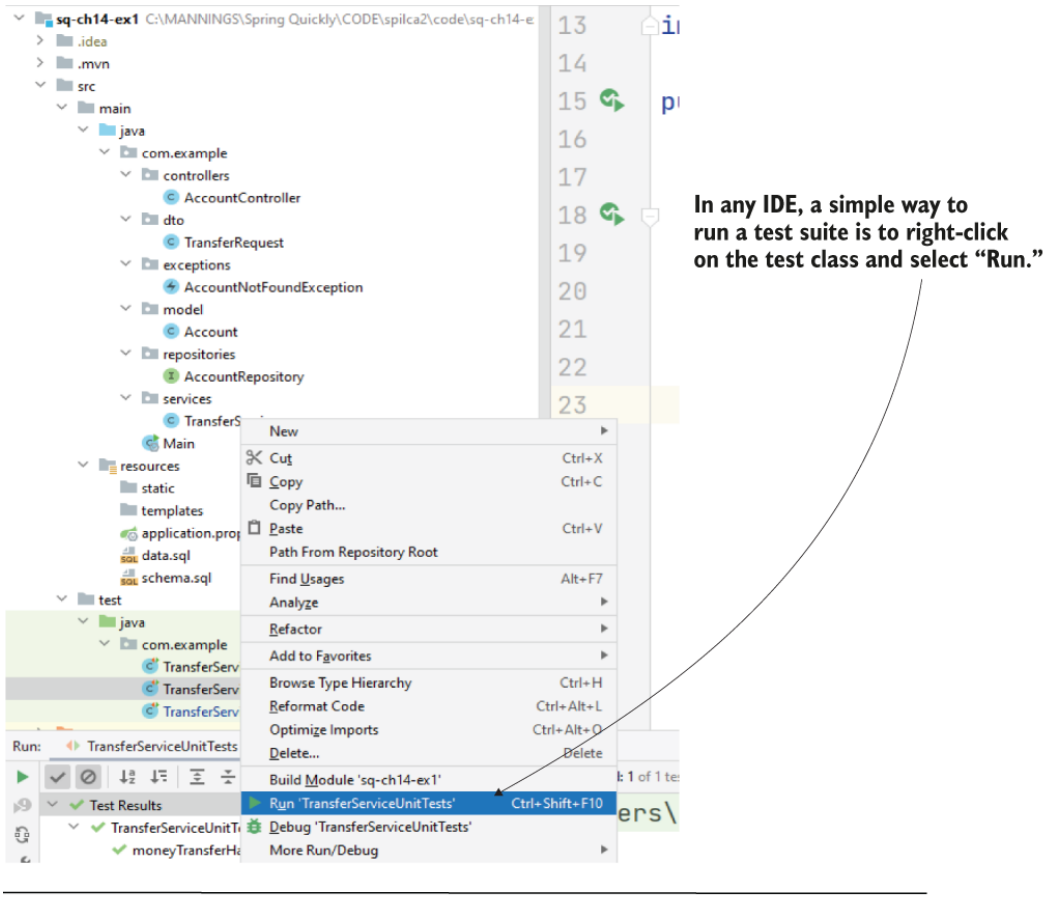

Figure 15.8 Running a test. An IDE usually offers several ways you can run a test. One of them is to right-click on the test class and select "Run." You can also run all the project tests by right-clicking on the project name and select "Run tests." Different IDEs might have slightly different graphical interfaces, but they all look similar to what you see in this figure. After running the tests, the IDE shows the status for each test.

Even if you find, in many cases, the `mock()` method being declared inside the method, as presented in listings 15.2 through 15.4, I usually prefer a different approach to create the mock object. It's not necessarily better or more often used, but I consider it a cleaner way to use annotations to create the mock and the tested object, as presented in listing 15.5.

Listing 15.5 Using annotations for mock dependencies

```
@ExtendWith(MockitoExtension.class)                                    ❶
public class TransferServiceWithAnnotationsUnitTests {

  @Mock                                                                ❷
  private AccountRepository accountRepository;

  @InjectMocks                                                         ❸
  private TransferService transferService;

  @Test
  public void moneyTransferHappyFlow() {
    Account sender = new Account();
    sender.setId(1);
    sender.setAmount(new BigDecimal(1000));

    Account destination = new Account();
    destination.setId(2);
    destination.setAmount(new BigDecimal(1000));

    given(accountRepository.findById(sender.getId()))
        .willReturn(Optional.of(sender));

    given(accountRepository.findById(destination.getId()))
        .willReturn(Optional.of(destination));

    transferService.transferMoney(1, 2, new BigDecimal(100));

    verify(accountRepository)
      .changeAmount(1, new BigDecimal(900));

    verify(accountRepository)
      .changeAmount(2, new BigDecimal(1100));
  }
}
```

❶ Enable the use of @Mock and @InjectMocks annotations.

❷ Use the @Mock annotation to create a mock object and inject it into the test class's annotated field.

❸ Use the @InjectMocks to create the tested object and inject it into the class's annotated field.

Observe how, instead of declaring these objects inside the test method, I took them out as the class parameters and annotated them with `@Mock` and `@InjectMocks`. When you use the `@Mock` annotation, the frame-

work creates and injects a mock object in the annotated attribute. With `@InjectMocks` annotation, you create the object to test and instruct the framework to inject all the mocks (created with `@Mock`) in its parameters.

For the `@Mock` and `@InjectMocks` annotations to work, you also need to annotate the test class with the `@ExtendWith(MockitoExtension.-class)` annotation. When annotating the class this way, you enable an extension that allows the framework to read the `@Mock` and `@Inject-Mocks` annotations and control the annotated fields.

Figure 15.9 summarizes the test we built. In this visual, you find the steps and code we wrote to solve each of the steps we enumerated when we started writing the test:

1. *Assumptions*—Enumerate and control the dependencies.
2. *Call*—Execute the tested method.
3. *Validations*—Verify the executed method had the expected behavior

Figure 15.9 The main parts of the test implementation. (1) Define and control the dependencies, (2) execute the tested method, and (3) verify the method behaved as expected.

Remember that happy flows are not the only ones you need to test. You also want to know that the method executes in the desired way when it encounters an exception. Such a flow is called an *exception flow*. In our example, an exception flow could happen if either the sender or the destination account details are not found for the given ID, as presented in figure 15.10.

Listing 15.6 shows you how to write the unit test for an exception flow. If you want to check that the method throws an exception, you use `assertThrows()`. You specify the exception you expect the method will throw and specify the tested method. The `assertThrows()` method calls the tested method and validates that it throws the expected exception.

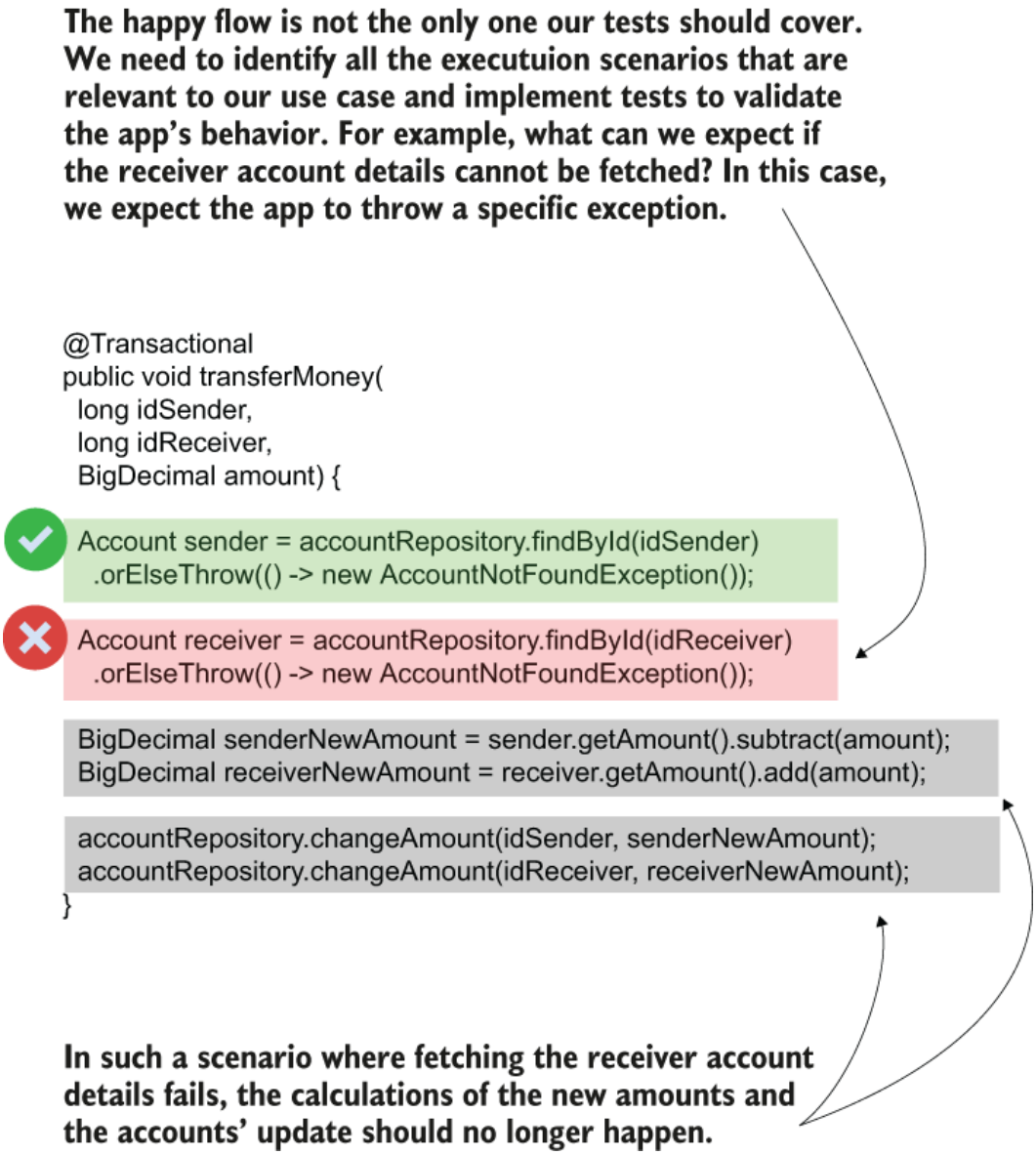

Figure 15.10 An exception flow is an execution that encountered an error or an exception. For example, if the receiver account details were not found, the app should throw an `AccountNotFoundException`, and the `changeAmount()` method should not be called. Exception flows are important as

well, and we need to implement tests for these scenarios the same as we do for happy flows.

Listing 15.6 Testing an exception flow

```
@ExtendWith(MockitoExtension.class)
public class TransferServiceWithAnnotationsUnitTests {

  @Mock
  private AccountRepository accountRepository;

  @InjectMocks
  private TransferService transferService;

  @Test
  public void moneyTransferDestinationAccountNotFoundFlow() {
    Account sender = new Account();
    sender.setId(1);
    sender.setAmount(new BigDecimal(1000));

    given(accountRepository.findById(1L))
        .willReturn(Optional.of(sender));

    given(accountRepository.findById(2L))
        .willReturn(Optional.empty());          ❶

    assertThrows(
        AccountNotFoundException.class,         ❷
        () -> transferService.transferMoney(1, 2, new BigDecimal(100)
    );

    verify(accountRepository, never())          ❸
        .changeAmount(anyLong(), any());
  }
}
```

❶ We control the mock AccountRepository to return an empty Optional when the findById() method is called for the destination account.

❷ We assert that the method throws an AccountNotFoundException in the given scenario.

❸ We use the verify() method with the never() conditional to assert that the changeAmount() method hasn't been called.

TESTING THE VALUE A METHOD RETURNS

An often encountered case is needing to check the value a method returns. The next listing shows a method we implemented in chapter 9, in the project "sq-ch9-ex1." How would you implement a unit test for this method, considering you need to test the scenario where the user provides the correct credentials to log in?

Listing 15.7 The implementation of the login controller action we want to unit test

```
@PostMapping("/")
  public String loginPost(
      @RequestParam String username,
      @RequestParam String password,
      Model model
  ) {
    loginProcessor.setUsername(username);
    loginProcessor.setPassword(password);
    boolean loggedIn = loginProcessor.login();

    if (loggedIn) {
      model.addAttribute("message", "You are now logged in.");
    } else {
      model.addAttribute("message", "Login failed!");
    }

    return "login.html";
  }
```

You follow the same steps you learned in this section:

1. Identify and control the dependencies.
2. Call the tested method.
3. Verify the tested method execution behaved as expected.

Listing 15.8 shows the unit test implementation. Observe that we mocked the dependencies whose behavior we want to control or verify: the `Model` and the `LoginProcessor` objects. We instruct the `LoginProcessor` mock object to return true (which is equivalent to assuming the user provided the correct credentials), and we call the method we want to test.

We verify the following:

- That the method returned the string “login.html.” We use an assert method to validate the method returned a value. As shown in listing 15.8, we can use the `assertEquals()` method, which compares an expected value with the value the method returned.
- The `Model` instance contains the valid message “You are now logged in.” We use the `verify()` method to validate the `addAttribute()` method of the `Model` instance has been called with the correct value as a parameter.

Listing 15.8 Testing the returned value in a unit test

```
@ExtendWith(MockitoExtension.class)
class LoginControllerUnitTests {

  @Mock
  private Model model;                                          ❶

  @Mock
  private LoginProcessor loginProcessor;                        ❶

  @InjectMocks
  private LoginController loginController;                      ❶

  @Test
  public void loginPostLoginSucceedsTest() {
    given(loginProcessor.login())                               ❷
      .willReturn(true);

    String result =                                             ❸
      loginController.loginPost("username", "password", model);


    assertEquals("login.html", result);                         ❹

    verify(model)                                               ❺
     .addAttribute("message", "You are now logged in.");
  }
}
```

❶ We define the mock objects and inject them into the instance whose behavior we test.

❷ We control the LoginProcessor mock instance, telling it to return true when its method login() is called.

❸ We call the tested method with the given assumptions.

❹ We verify the tested method returned value.

❺ We verify the message attribute was added with the correct value on the model object.

By controlling the inputs (the parameter values and how the mock objects behave), you can also test what happens in different scenarios. In the next listing, we make the `LoginProcessor` mock object's `login()` method return false to test what would happen if the login fails.

Listing 15.9 Adding the test to validate the failed login scenario

```
@ExtendWith(MockitoExtension.class)
class LoginControllerUnitTests {

  // Omitted code

  @Test
  public void loginPostLoginFailsTest() {
    given(loginProcessor.login())
      .willReturn(false);

    String result =
      loginController.loginPost("username", "password", model);

    assertEquals("login.html", result);

    verify(model)
      .addAttribute("message", "Login failed!");
  }
}
```

15.2.2 Implementing integration tests

In this section, we discuss integration tests. An integration test is very similar to a unit test. We'll even continue writing them with JUnit. But instead of focusing on how a particular component works, an integration test focuses on how two or more components interact.

Remember the analogy with the car's dashboard indicators? If the car's tank is full, but something breaks in the gas distribution between the tank and the engine, the car still won't start. Unfortunately, this time the gas indicator won't show you something is wrong because the tank has enough gas, and as an isolated component, it works correctly. In such a case, we don't know why the car doesn't work. The same problem might

happen to an app. Even if some components work correctly when they are isolated one from another, they don't correctly "talk" to each other. Writing integration tests helps us mitigate problems that could happen when components work correctly independently but don't communicate correctly.

We'll take the same example we used for the unit tests for this example: the money transfer use case we implemented in chapter 14 (project "sq-ch14-ex1").

What kind of integrations can we test? We have a few possibilities:

- *Integration between two (or more) objects of your app*. Testing that the objects interact correctly helps you identify problems in how they collaborate if you change one of them.
- *Integration of an object of your app with some capability the framework enhances it with*. Testing how an object interacts with some capability the framework provides helps you identify issues that can occur when you upgrade the framework to a new version. The integration test helps you immediately identify if something changed in the framework and the capability the object relies on doesn't work the same way.
- *Integration of the app with its persistence layer (the database)*. Testing how the repository works with the database ensures you quickly identify problems that might occur when upgrading or changing a dependency that helps your app work with persisted data (such as the JDBC driver).

An integration test looks very similar to a unit test. You still follow the same steps of identifying the assumptions, calling the tested method, and validating the results. The difference is that now the test doesn't focus on an isolated piece of logic, so you don't necessarily have to mock all the dependencies. You might allow a method you test to call another's real object's (not a mock's) method because you want to test the two objects communicate correctly. So, if for a unit test it was mandatory to mock the repository, for an integration test that is no longer mandatory. You can still mock it if the test you write doesn't care about how the service interacts with that repository, but if you want to test how these two objects communicate, you can let the real object be called (figure 15.11).

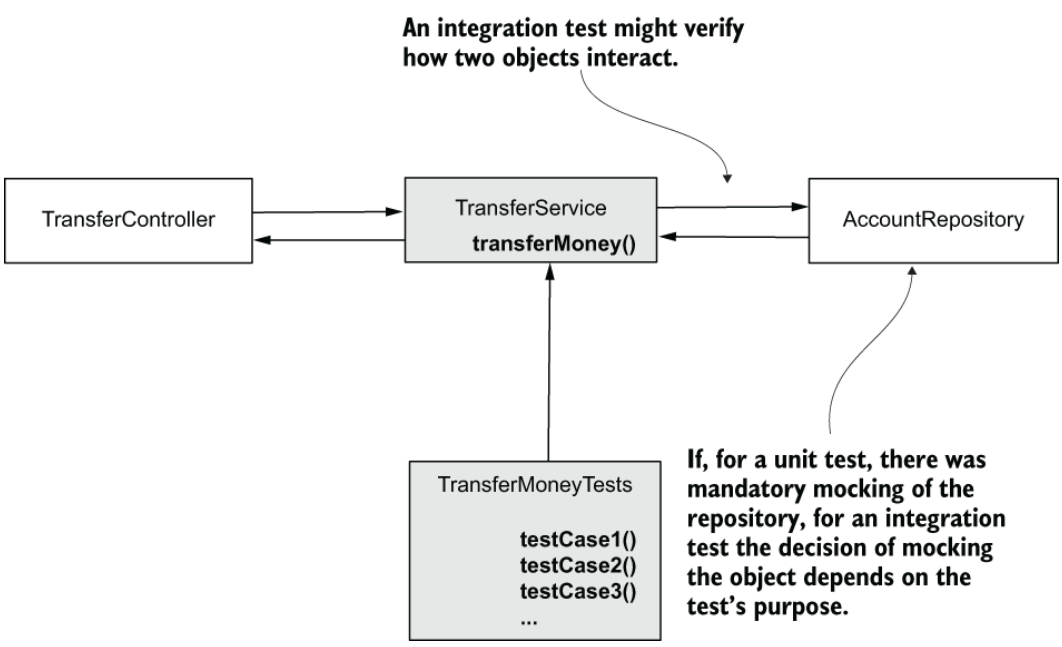

Figure 15.11 In a unit test's case, all the dependencies needed to be mocked. If the integration test's purpose is to verify how the `TestService` and `AccountRepository` interact, the repository can be the real object. An integration test can still mock an object if its purpose doesn't verify the integration with a specific component.

NOTE If you decide not to mock the repository in an integration test, you should use an in-memory database such as H2 instead of the real database. This will help you keep your tests independent of the infrastructure that runs the app. Using the real database could cause latencies in test executions and even make tests fail in case of infrastructure or networking problems. Since you test the application and not the infrastructure, you should avoid all this trouble by using a mock in-memory database.

With a Spring app, you'll generally use integration tests to verify that your app's behavior correctly interacts with the capabilities Spring provides. We name such a test a "Spring integration test." Unlike a unit test, an integration test enables Spring to create the beans and configure the context (just as it does when running the app).

Listing 15.10 shows you how simple it is to transform a unit test into a Spring integration test. Observe that we can use the `@MockBean` annotation to create a mock object in our Spring Boot application. This annotation is quite similar to the `@Mock` annotation we used for unit tests, but it also makes sure the mock object is added to the application context. This way, you can simply use `@Autowired` (as you learned in chapter 3) to inject the object whose behavior you test.

Listing 15.10 Implementing a Spring integration test

```
@SpringBootTest
class TransferServiceSpringIntegrationTests {

  @MockBean                                                    ❶
  private AccountRepository accountRepository;

  @Autowired                                                   ❷
  private TransferService transferService;

  @Test
  void transferServiceTransferAmountTest() {
    Account sender = new Account();                            ❸
    sender.setId(1);                                           ❸
    sender.setAmount(new BigDecimal(1000));                    ❸

    Account receiver = new Account();                          ❸
    receiver.setId(2);                                         ❸
    receiver.setAmount(new BigDecimal(1000));                  ❸

    when(accountRepository.findById(1L))                       ❸
      .thenReturn(Optional.of(sender));                        ❸
    when(accountRepository.findById(2L))                       ❸
      .thenReturn(Optional.of(receiver));                      ❸

    transferService
       .transferMoney(1, 2, new BigDecimal(100));              ❹

    verify(accountRepository)                                  ❺
      .changeAmount(1, new BigDecimal(900));                   ❺
    verify(accountRepository)                                  ❺
      .changeAmount(2, new BigDecimal(1100));                  ❺
  }

}
```

❶ Create a mock object that is also part of the Spring context.

❷ Inject the real object from the Spring context whose behavior you'll test.

❸ Define all the assumptions for the test.

❹ Call the tested method.

❺ Validate the tested method call has the expected behavior.

NOTE The `@MockBean` annotation is a Spring Boot annotation. If you have a plain Spring app and not a Spring Boot one as presented here, you won't be able to use `@MockBean`. However, you can still use the same approach by annotating the configuration class with `@ExtendsWith(SpringExtension.class)`. An example using this annotation is in the project "sq-ch3-ex1."

You run the test the same way you do for any other test. However, even if it looks very similar to a unit test, Spring now knows the tested object and manages it as it would in a running app. For example, if we upgraded the Spring version and, for some reason, the dependency injection no longer worked, the test would fail even if we didn't change anything in the tested object. The same applies to any capability Spring offers to the tested method: transactionality, security, caching, and so on. You would be able to test your method's integration to any of these capabilities the method uses in your app.

NOTE In a real-world app, use unit tests to validate components' behavior and the Spring integration tests to validate the necessary integration scenarios. Even if a Spring integration test could be used to validate the component's behavior (implement all the test scenarios for the method's logic), it's not a good idea to use integration tests for this purpose. Integration tests take a longer time to execute because they have to configure the Spring context. Every method call also triggers several Spring mechanisms Spring needs, depending on what capabilities it offers to that specific method. It doesn't make sense to spend time and resources to execute these for every scenario of your app's logic. To save time, the best approach is to rely on unit tests to validate your apps' components' logic and use the integration tests only to validate how they integrate with the framework.

Summary

- A test is a small piece of code you write to validate the behavior of certain logic implemented in your app. Tests are necessary because they help you ensure that future app developments don't break existing capabilities. Tests also help as documentation.
- Tests fall into two categories: unit tests and integration tests. Each has its purposes.

    - A unit test only focuses on an isolated piece of logic and validates how one simple component works without checking how it integrates with other features. Unit tests are helpful because they execute fast and point us directly to the problem a specific component might face.
    - An integration test focuses on validating the interaction between two or more components. They are essential because sometimes two components might work correctly in isolation but don't communicate well. Integration tests help us mitigate problems generated by such cases.
- Sometimes in tests you want to eliminate dependencies to some components to allow the test to focus on how some but not all parts interact. In such cases, we replace the components we don't want to test with "mocks": fake objects you control to eliminate dependencies you don't want to test and allow the test to focus only on specific interactions.
- Any test has three main parts:
    - *Assumptions*—Define the input values and the way the mock objects behave.
    - *Call/Execution*—Call the method you want to test.
    - *Validations*—Verify the way the tested method behaved.