

JUnit IN ACTION

THIRD EDITION

Cătălin Tudose
Petar Tahchiev
Felipe Leme
Vincent Massol
Gary Gregory



MANNING

MEAP



WOW! eBook
www.wowebook.org



MEAP Edition
Manning Early Access Program
jUnit in Action
Third Edition
Version 3

Copyright 2020 Manning Publications

For more information on this and other Manning titles go to
manning.com

©Manning Publications Co. To comment go to [liveBook](#)

WOW! eBook
www.wowebook.org

welcome

Thank you for purchasing the MEAP edition of *JUnit in Action, Third Edition*.

To get the most benefit from this book, you will need to have some established skills in programming: knowledge of object-oriented programming concepts; intermediate Java 8 language skills; basic understanding of the Apache Maven tool; basic skills to open a Java program in and IDE, edit it, and launch it in execution.

For some particular dedicated chapters, you will need to have the basic understanding of some other techniques. We include here: the purpose of the Spring and Spring Boot frameworks; the basic understanding of the Representational State Transfer (REST) software architecture; the basic understanding of J2EE, CDI and the dependency injection principle; the basic understanding of main relational databases concepts (tables, relationships, primary keys, foreign keys) and of the main JPA concepts (object-relational mapping functionality, annotations).

The book topic is automated software testing with the help of JUnit 5. Automated software testing is a technique that will greatly increase the development speed and that will remove much of the debugging nightmare - everything with the help of JUnit 5 and its new features.

You will understand how to write safer code and how to introduce new functionality much quicker. JUnit 5 tests will drive the programmer to implement the code that will do what it is supposed to do and will prevent introducing bugs into the well working existing code. To quote Martin Fowler: "Never in the field of software development have so many owed so much to so few lines of code".

By reading this book, you will discover the features and architecture of JUnit 5 and you'll learn how to develop safe and flexible automatically tested Java applications. You will be able to put in practice the steps towards Test Driven Development and Behavior Driven Development, making sure that your software is both doing the things right and the right things. You will also understand how to create a testing strategy pyramid – an application tested into buckets of different granularity, at different levels.

It is a big book for a big library, and I hope you find it as useful to read as I did to write it. Please post any questions, comments, or suggestions you have about the book in the [liveBook's Discussion Forum](#). Your feedback is essential in developing the best book possible.

—Cătălin Tudose, PhD

brief contents

PART 1: JUNIT 5 ESSENTIALS

- 1 *JUnit jump-start*
- 2 *Exploring core JUnit*
- 3 *JUnit architecture*
- 4 *Migrating from JUnit 4 to JUnit 5*
- 5 *Software testing principles*

PART 2: DIFFERENT TESTING STRATEGIES

- 6 *Tests quality*
- 7 *Coarse-grained testing with stubs*
- 8 *Testing with mock objects*
- 9 *In-container testing*

PART 3: WORKING WITH JUNIT 5 AND OTHER TOOLS

- 10 *Running JUnit tests from Maven 3*
- 11 *Running JUnit tests from Gradle 5*
- 12 *JUnit 5 IDE support*
- 13 *Continuous integration with JUnit 5*

PART 4: WORKING WITH MODERN FRAMEWORKS AND JUNIT 5

- 14 *JUnit 5 extension model*
- 15 *Presentation layer testing*
- 16 *Testing Spring applications*
- 17 *Testing Spring Boot applications*
- 18 *Testing a REST API*
- 19 *Testing database applications*

PART 5: DEVELOPING APPLICATIONS WITH JUNIT 5

- 20 *Test Driven Development with JUnit 5*
- 21 *Behavior Driven Development with JUnit 5*
- 22 *Implementing a test strategy pyramid with JUnit 5*

1

JUnit jump-start

This chapter covers

- Understanding JUnit
- Installing JUnit
- Writing your first tests
- Running tests

Never in the field of software development was so much owed by so many to so few lines of code.

—Martin Fowler

All code is tested. During development, the first thing we do is run our own programmer’s acceptance test. We code, compile, and run. When we run, we test. The test may just consist of clicking a button to see whether it brings up the expected menu or looking at a result to compare it with the expected value. Nevertheless, every day, we code, we compile, we run, and we *test*.

When we test, we often find issues, especially during early runs. So we code, compile, run, and test again.

Most of us quickly develop a pattern for our informal tests: add a record, view a record, edit a record, and delete a record. Running a little test suite like this by hand is easy enough to do, so we do it—over and over again.

Some programmers like doing this type of repetitive testing. It can be a pleasant break from deep thought and hardcoding. When our little click-through tests finally succeed, we have a real feeling of accomplishment (“Eureka! I found it!”).

Other programmers dislike this type of repetitive work. Rather than run the tests by hand, they prefer to create a small program that runs the tests automatically. Play-testing code is one thing; running automated tests is another.

If you're a "play-test" developer, this book is for you. It shows you that creating automated tests can be easy, effective, and even fun.

If you're already test-infected,¹ this book is also for you. I cover the basics in part 1 and move on to tough, real-life problems in parts 2–5.

1.1 Proving that a program works

Some developers feel that automated tests are essential parts of the development process: you cannot *prove* that a component works until it passes a comprehensive series of tests. In fact, two developers felt that this type of unit testing was so important that it deserved its own framework. In 1997, Erich Gamma and Kent Beck created a simple but effective unit testing framework for Java called *JUnit*: they were on a long plane trip, and it gave them something interesting to do. Erich wanted Kent to learn Java, and Erich was interested in knowing more about the SUnit testing framework that Kent created earlier for Smalltalk, and the flight gave them time to do both.

DEFINITION *Framework*—A semi-complete application that provides a reusable common structure to share among applications.² Developers incorporate the framework into their own applications and extend it to meet their specific needs. Frameworks differ from toolkits by providing a coherent structure rather than a simple set of utility classes. A framework defines a skeleton, and the application defines its own features to fill out the skeleton. The developer code is called appropriately by the framework. Developers can worry less about whether a design is good and focus more on implementing domain-specific functions.

If you recognize the names Erich Gamma and Kent Beck, that's for a good reason. Gamma is one of the Gang of Four who gave us the now-classic *Design Patterns* book.³ Beck is equally well known for his groundbreaking work in the software discipline known as Extreme Programming (www.extremeprogramming.org).

¹ *Test-infected* is a term coined by Erich Gamma and Kent Beck in "Test-Infected: Programmers Love Writing Tests," *Java Report* 3 (7), 37–50, 1998.

² Ralph Johnson and Brian Foote, "Designing Reusable Classes," *Journal of Object-Oriented Programming* 1 (2): 22–35, 1988; www.laputan.org/drc/drc.html.

³ Erich Gamma et al., *Design Patterns* (Reading, MA: Addison-Wesley, 1995).

JUnit quickly became the de facto standard framework for developing unit tests in Java. Today, JUnit (<https://junit.org>) is open source software hosted on GitHub, with an Eclipse Public License. And the underlying testing model, known as *xUnit*, is on its way to becoming the standard framework for any language. xUnit frameworks are available for ASP, C++, C#, Eiffel, Delphi, Perl, PHP, Python, REBOL, Smalltalk, and Visual Basic—to name just a few.

The JUnit team didn't invent software testing or even unit tests, of course. Originally, the term *unit test* described a test that examined the behavior of a single unit of work: a class or a method. Over time, the use of the term *unit test* broadened. The Institute of Electrical and Electronics Engineers (IEEE), for example, has defined unit testing as “testing of individual hardware or software units *or groups of related units*” (emphasis added).⁴

In this book, I use the term *unit test* in the narrower sense to mean a test that examines a single unit in isolation from other units. I focus on the type of small, incremental test that programmers apply to their own code. Sometimes, these tests are called *programmer tests* to differentiate them from quality-assurance or customer tests (<http://c2.com/cgi/wiki?ProgrammerTest>).

Here is a generic description of a typical unit test from the perspective of this book: “Confirms that the method accepts the expected range of input and that the method returns the expected value for each input.” This description asks us to test the behavior of a method through its interface. If we give it value *x*, will it return value *y*? If we give it value *z* instead, will it throw the proper exception?

DEFINITION *Unit test*—A test that examines the behavior of a distinct unit of work. A *unit of work* is a task that is not directly dependent on the completion of any other task. Within a Java application, the distinct unit of work is often, but not always, a single method. In contrast, *integration tests* and *acceptance tests* examine how various components interact.

Unit tests often focus on testing whether a method is following the terms of its API contract. Like a written contract between people who agree to exchange certain goods or services under specific conditions, an *API contract* is a formal agreement made by the signature of a method. A method requires its callers to provide specific object references or primitive values and returns an object reference or primitive value. If the method cannot fulfill the contract, the test should throw an exception, and we say that the method has *broken* its contract.

DEFINITION *API contract*—A view of an application programming interface (API) as a formal agreement between the caller and the callee. Often, unit tests help define the API contract by demonstrating the expected behavior. The notion of an API contract arises from the practice of Design by Contract, popularized by the Eiffel programming language (<http://archive.eiffel.com/doc/manuals/technology/contract>).

⁴ IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries (New York, IEEE, 1990).

In this chapter, we walk through creating a unit test for a simple class from scratch. We start by writing a test and its minimal runtime framework so you can see how things used to be done. Then we roll out JUnit to show you how the right tools can make life much simpler.

1.2 Starting from scratch

For our first example, we will create a very simple calculator class that adds two numbers. Our calculator, shown in listing 1.1., provides an API to clients and does not contain a user interface. To test its functionality, we'll first create our own pure Java tests and later move to JUnit 5.

Listing 1.1 The test calculator class

```
public class Calculator {  
    public double add(double number1, double number2) {  
        return number1 + number2;  
    }  
}
```

Although the documentation isn't shown, the intended purpose of `Calculator`'s `add(double, double)` method is to take two doubles and return the sum as a double. The compiler can tell you that the code compiles, but you should also make sure it works at runtime. A core principle of unit testing is, "Any program feature without an automated test simply doesn't exist."⁵ The `add` method represents a core feature of the calculator. You have some code that allegedly implements the feature. What is missing is an automated test that proves the implementation works.

Isn't the add method too simple to break?

The current implementation of the `add` method is too simple to break. If `add` were a minor utility method, you might not test it directly. In that case, if `add` did fail, tests of the methods that used `add` would fail. The `add` method would be tested indirectly, but tested nonetheless. In the context of the calculator program, `add` is not just a method, but also a *program feature*. To have confidence in the program, most developers would expect there to be an automated test for the `add` feature, no matter how simple the implementation appears to be. In some cases, you can prove program features through automatic functional tests or automatic acceptance tests. For more about software tests in general, see chapter 5.

Yet testing anything at this point seems to be problematic. You do not even have a user interface with which to enter a pair of doubles. You could write a small command-line

⁵ Kent Beck, *Extreme Programming Explained: Embrace Change* (Reading, MA: Addison-Wesley, 1999).

program that waited for you to type two `double` values and then displayed the result. Then, of course, you would also be testing your own ability to type a number and add the result yourself, which is much more than you want to do. You just want to know whether this unit of work actually adds two doubles and returns the correct sum. You do not want to test whether programmers can type numbers!

Meanwhile, if you are going to go to the effort of testing your work, you should also try to preserve that effort. It is good to know that the `add(double, double)` method worked when you wrote it. What you really want to know, however, is whether the method works when you ship the rest of the application or whenever you make a subsequent modification. If we put these requirements together, we come up with the idea of writing a simple test program for the `add` method.

The test program can pass known values to the method and see whether the result matches expectations. You can also run the program again later to be sure the method continues to work as the application grows. So what is the simplest possible test program you could write? What about this `CalculatorTest` program?

Listing 1.2 A simple test calculator program

```
public class CalculatorTest {  
    public static void main(String[] args) {  
        Calculator calculator = new Calculator();  
        double result = calculator.add(10, 50);  
        if (result != 60) {  
            System.out.println("Bad result: " + result);  
        }  
    }  
}
```

`CalculatorTest` is simple indeed: it creates an instance of `Calculator`, passes two numbers to it, and checks the result. If the result does not meet your expectations, you print a message on standard output.

If you compile and run this program now, the test quietly passes, and all seems to be well. But what happens if you change the code so that it fails? You have to watch the screen carefully for the error message. You may not have to supply the input, but you are still testing your own ability to monitor the program's output. You want to test the code, not yourself!

The conventional way to signal error conditions in Java is to throw an exception. Let's throw an exception instead, to indicate a test failure.

Meanwhile, you may also want to run tests for other `Calculator` methods that you have not written yet, such as `subtract` or `multiply`. Moving to a modular design will make catching and handling exceptions easier; it will also be easier to extend the test program later. The next listing shows a slightly better `CalculatorTest` program.

Listing 1.3 A (slightly) better test calculator program

```
public class CalculatorTest {  
  
    private int nbErrors = 0;
```

```

public void testAdd() {                                #1
    Calculator calculator = new Calculator();
    double result = calculator.add(10, 50);
    if (result != 60) {
        throw new IllegalStateException("Bad result: " + result);
    }
}

public static void main(String[] args) {
    CalculatorTest test = new CalculatorTest();
    try {
        test.testAdd();                                #2
    }
    catch (Throwable e) {
        test.nbErrors++;                            #2
        e.printStackTrace();                         #2
    }
    if (test.nbErrors > 0) {                        #2
        throw new IllegalStateException("There were " + test.nbErrors
            + " error(s)");
    }
}
}

```

At #1, you move the test into its own `testAdd` method. Now it's easier to focus on what the test does. You can also add more methods with more unit tests later without making the `main` method harder to maintain. At #2, you change the `main` method to print a stack trace when an error occurs; then, if there are any errors, you end by throwing a summary exception.

Now that you have seen a simple application and its tests, you can see that even this small class and its tests can benefit from the little bit of skeleton code you created to run and manage test results. But as an application gets more complicated and the tests become more involved, continuing to build and maintain a custom testing framework becomes a burden.

Next, we take a step back and look at the general case for a unit testing framework.

1.2.1 Understanding unit testing frameworks

Unit testing has several best practices that frameworks should follow. The seemingly minor improvements in the `CalculatorTest` program in listing 1.3 highlight three rules that (in my experience) all unit testing frameworks should follow:

- Each unit test should run independently of all other unit tests.
- The framework should detect and report errors test by test.
- It should be easy to define which unit tests will run.

The “slightly better” test program comes close to following these rules but still falls short. For each unit test to be truly independent, for example, each should run in a different class instance.

1.2.2 Adding unit tests

You can add new unit tests by adding a new method and then adding a corresponding try/catch block to main. This is a step up but still short of what you would want in a real unit test suite. Experience tells us that large try-catch blocks cause maintenance problems. You could easily leave out a unit test and never know it!

It would be nice if you could just add new test methods and continue working, but if you did, how would the program know which methods to run? Well, you could have a simple registration procedure. A registration method would at least inventory which tests are running.

Another approach would be to use Java's reflection capabilities. A program could look at itself and decide to run whatever methods follow a certain naming convention, such as those that begin with test.

Making it easy to add tests (the third rule in the earlier list) sounds like another good rule for a unit testing framework. The support code that realizes this rule (via registration or reflection) would not be trivial, but it would be worthwhile. You'd have to do a lot of work up front, but that effort would pay off each time you add a new test.

Fortunately, the JUnit team has saved you the trouble. The JUnit framework already supports discovering methods. It also supports using a different class instance and class loader instance for each test and reports all errors on a test-by-test basis. The team has defined three discrete goals for the framework:

- The framework must help us write useful tests.
- The framework must help us create tests that retain their value over time.
- The framework must help us lower the cost of writing tests by reusing code.

We'll discuss these goals further in chapter 2.

Next, let's see how to set up JUnit.

1.3 Setting up JUnit

To use JUnit to write your application tests, you need to know about its dependencies. You'll work with JUnit 5, the latest version of the framework when this book was written. Version 5 of the testing framework is a modular one; you can no longer simply add a jar file to your project compilation classpath and your execution classpath. In fact, starting with version 5, the architecture is no longer monolithic (as discussed in chapter 3). Also, with the introduction of annotations in Java 5, JUnit 4 and JUnit 5 have also moved to this approach. JUnit 5 is heavily based on annotations—a contrast with the idea of extending a base class for all testing classes and using naming conventions for all testing methods to match the `textXyz` pattern, as done in previous versions.

NOTE If you are familiar with JUnit 4, you may wonder what's new in this version, as well as why and how to move toward it. JUnit 5 represents the next generation of JUnit. You'll use the programming capabilities introduced starting with Java 8; you'll be able to build tests modularly and hierarchically; and the tests will be easier to understand, maintain, and extend. Chapter 4 discusses the transition from JUnit 4 to JUnit 5 and

shows that the projects you are working on may benefit from the great features of JUnit 5. As you'll see, you can make this transition smoothly, in small steps.

To manage JUnit 5's dependencies efficiently, it's logical to work with the help of a build tool. In this book, we've chosen to use Maven, a very popular build tool. Chapter 10 is dedicated to the topic of running JUnit tests from Maven. What you need to know now are the basic ideas behind Maven: configuring your project through the pom.xml file, executing the mvn clean install command, and understanding the command's effects.

NOTE You can download Maven from <https://maven.apache.org>. When this book was being written, the latest version was 3.6.3.

The dependencies that are always needed in the pom.xml file are shown in the following listing. In the beginning, you need only junit-jupiter-api and junit-jupiter-engine.

Listing 1.4 pom.xml JUnit 5 dependencies

```
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-api</artifactId>
    <version>5.6.0</version>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-engine</artifactId>
    <version>5.6.0</version>
    <scope>test</scope>
</dependency>
```

To be able to run tests from the command prompt, make sure your pom.xml configuration file includes a JUnit provider dependency for the Maven Surefire plugin. Here's what this dependency looks like.

Listing 1.5 Maven Surefire plugin configuration in pom.xml

```
<build>
    <plugins>
        <plugin>
            <artifactId>maven-surefire-plugin</artifactId>
            <version>2.22.2</version>
        </plugin>
    </plugins>
</build>
```

As Windows is the most commonly used operating system (OS), our example configuration details use Windows 10, the latest version of it. Concepts such as the path, environment variables, and the command prompt also exist in other OSs; follow your documentation guidelines if you will be running the examples on an OS other than Windows.

To run the tests, the bin folder from the Maven directory must be on the OS path (figure 1.1). You also need to configure the `JAVA_HOME` environment variable on your OS to point to the Java installation folder (figure 1.2). In addition, your JDK version must be at least 8, as required by JUnit 5.

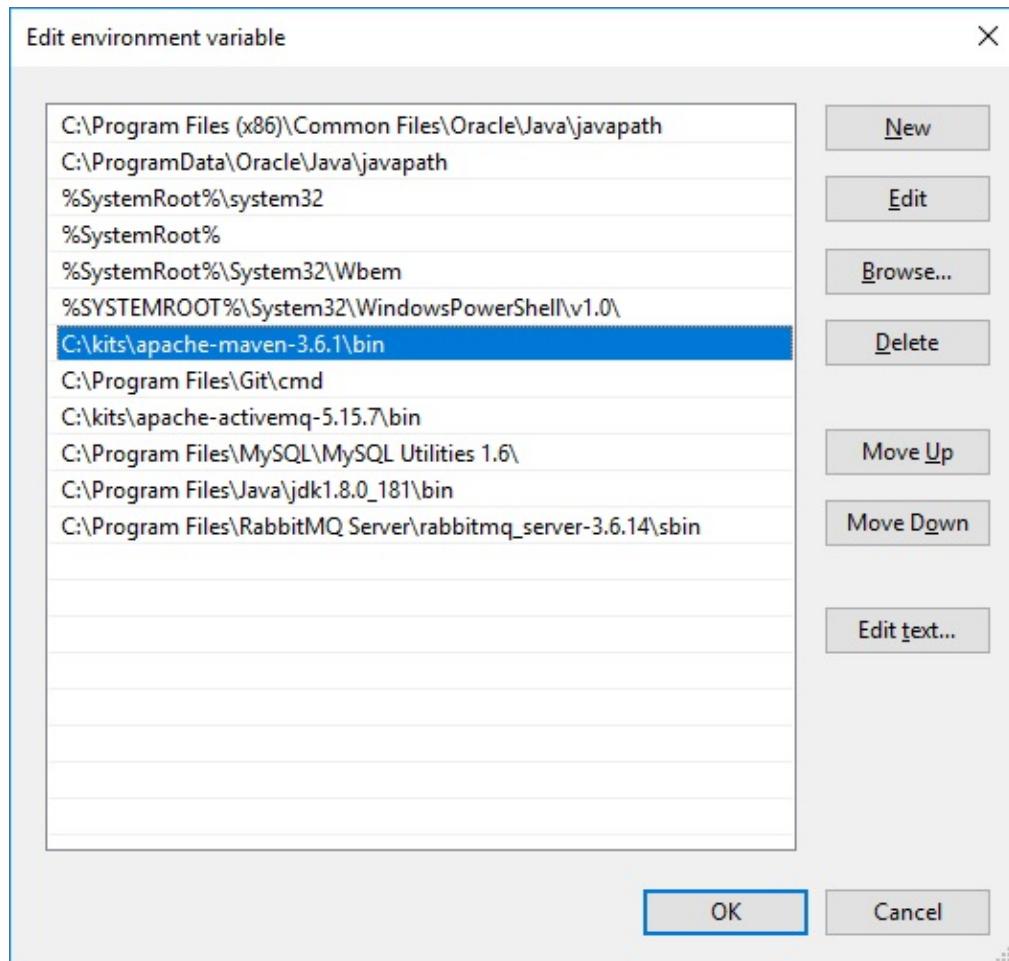


Figure 1.1 The configuration of the OS path must include the Apache Maven bin folder.

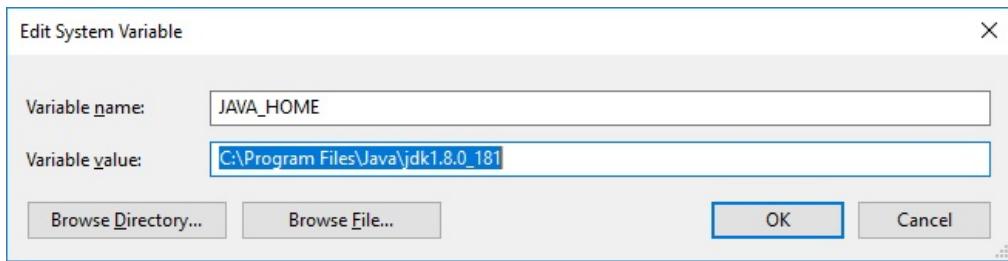


Figure 1.2 The configuration of the `JAVA_HOME` environment variable

You will need the source files from the chapter to get the following results. Open a command prompt into the project folder (the one containing the `pom.xml` file), and run the following command:

```
mvn clean install
```

This command will take the Java source code, compile it, test it and convert it into a runnable Java program (a `.jar` file in our case). Figure 1.3 shows the result of the test.

A screenshot of a Windows Command Prompt window titled 'cmd.exe'. The window displays the output of the 'mvn clean install' command. The output includes: '[INFO] Results:', '[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0', '[INFO] --- maven-jar-plugin:2.4:jar (default-jar) @ ch01-jumpstart ---', '[INFO] Building jar: C:\Work\Manning\JUnit in Action 3rd Edition\Source code\ch01-jumpstart\target\ch01-jumpstart-1.0-SNAPSHOT.jar', '[INFO] --- maven-install-plugin:2.4:install (default-install) @ ch01-jumpstart ---', '[INFO] Installing C:\Work\Manning\JUnit in Action 3rd Edition\Source code\ch01-jumpstart\target\ch01-jumpstart-1.0-SNAPSHOT.jar to C:\Users\ftudose\.m2\repository\com\manning\junitbook\ch01-jumpstart\1.0-SNAPSHOT\ch01-jumpstart-1.0-SNAPSHOT.jar', '[INFO] Installing C:\Work\Manning\JUnit in Action 3rd Edition\Source code\ch01-jumpstart\pom.xml to C:\Users\ftudose\.m2\repository\com\manning\junitbook\ch01-jumpstart\1.0-SNAPSHOT\ch01-jumpstart-1.0-SNAPSHOT.pom'.

Figure 1.3 Execution of the JUnit tests using Maven and the command prompt

Part 3 of the book provides more details about running tests with the Maven and Gradle build tools.

1.4 Testing with JUnit

JUnit has many features that make writing and running tests easy. You'll see these features at work throughout this book:

- Separate test class instances and class loaders for each unit test to prevent side effects
- JUnit annotations to provide resource initialization and cleanup methods: `@BeforeEach`, `@BeforeAll`, `@AfterEach`, and `@AfterAll` (starting from version 5); and `@Before`, `@BeforeClass`, `@After`, and `@AfterClass` (up to version 4)
- A variety of assert methods that make it easy to check the results of your tests
- Integration with popular tools such as Maven and Gradle, as well as popular integrated development environments (IDEs) such as Eclipse, NetBeans, and IntelliJ

Without further ado, this listing shows what the simple `Calculator` test looks like when written with JUnit.

Listing 1.6 JUnit CalculatorTest program

```
import static org.junit.jupiter.api.Assertions.assertEquals;
import org.junit.jupiter.api.Test;

public class CalculatorTest { #1

    @Test #2
    public void testAdd() {
        Calculator calculator = new Calculator(); #3
        double result = calculator.add(10, 50); #4
        assertEquals(60, result, 0); #5
    }
}
```

Running such a test with the help of Maven results in behavior similar to that shown in figure 1.3. The test is very simple. At #1, you define a test class. It's common practice to end the class name with `Test`. JUnit 3 required extending the `TestCase` class, but this requirement was removed with JUnit 4. Also, up to JUnit 4, the class had to be public; starting with version 5, the test class can be public or private, and you can name it whatever you want.

At #2, you mark the method as a unit test method by adding the `@Test` annotation. In the past, the usual practice was to name test methods following the `testXYZ` pattern, as was required up to JUnit 3. Now that doing so is no longer required, some programmers drop the prefix and use a descriptive phrase as the method name. You can name your methods as you like; as long as they have the `@Test` annotation, JUnit will execute them. The JUnit 5 `@Test` annotation belongs to a new package, `org.junit.jupiter.api`, and the JUnit 4 `@Test` annotation belongs to the `org.junit` package. This book uses JUnit 5's capabilities except in some clearly emphasized cases (such as to demonstrate migration from JUnit 4).

At #3, you start the test by creating an instance of the `Calculator` class (the object under test). And at #4, as before, you execute the test by calling the method to test, passing it two known values.

At #5, the JUnit framework begins to shine! To check the result of the test, you call an `assertEquals` method, which you imported with a static import on the first line of the class. The Javadoc for the `assertEquals` method is

```
/**  
 * Assert that expected and actual are equal within the non-negative delta.  
 * Equality imposed by this method is consistent with Double.equals(Object)  
 * and Double.compare(double, double). */  
public static void assertEquals(  
    double expected, double actual, double delta)
```

In listing 1.6, you passed these parameters to `assertEquals`:

```
expected = 60  
actual = result  
delta = 0
```

Because you passed the calculator the values 10 and 50, you tell `assertEquals` to expect the sum to be 60. (You pass 0 as the delta because you are expecting no floating-point errors when adding 10 and 50, as the decimal part of these numbers is 0.) When you call the `calculator` object, you save the return value in a local double named `result`. Therefore, you pass that variable to `assertEquals` to compare with the expected value (60). If the actual value is not equal to the expected value, JUnit throws an unchecked exception, which causes the test to fail.

Most often, the `delta` parameter can be 0, and you can safely ignore it. This parameter comes into play with calculations that are not always precise, including many floating-point calculations. `delta` provides a range factor: if the actual value is within the range `expected - delta` and `expected + delta`, the test will pass. You may find this useful when you're performing mathematical computations with rounding or truncating errors or when you're asserting a condition about the modification date of a file, because the precision of these dates depends on the OS.

The remarkable thing about the JUnit `CalculatorTest` class in listing 1.6 is that the code is easier to write than the first `CalculatorTest` program in listing 1.2 or 1.3. In addition, you can run the test automatically through the JUnit framework.

When you run the test from the command line (figure 1.3), you see the amount of time it takes and the number of tests that passed. There are many other ways to run tests, from IDEs and from different build tools. This simple example gives you a taste of the power of JUnit and unit testing.

You may modify the `Calculator` class so that it has a bug – for example, instead of adding the numbers, to subtract them. Then, you can run the test and watch what the result looks like when a test fails.

In chapter 2, we will take a closer look at the JUnit framework classes (annotations and assertion mechanisms) and capabilities (nested and tagged tests, as well as repeated, parameterized, and dynamic tests). We'll show how they work together to make unit testing efficient and effective. You will learn how to use the JUnit 5 features in practice and differences between the old-style JUnit 4 and JUnit 5.

1.5 Summary

This chapter has covered the following:

- Why every developer should perform some type of test to see if code actually works. Developers who use automatic unit tests can repeat these tests on demand to ensure that *new code works and does not break existing tests*.
- Writing simple unit tests, which are not difficult to create without JUnit.
- As tests are added and become more complex, writing and maintaining tests becomes more difficult.
- Introduction to JUnit as a unit testing framework that makes it easier to create, run, and revise unit tests.
- Stepping through a simple JUnit test.

2

Exploring core JUnit

This chapter covers

- Understanding the JUnit life cycle
- Working with the core JUnit classes, methods, and annotations
- Demonstrating the JUnit mechanisms

Mistakes are the portals of discovery.

—James Joyce

In chapter 1, we decided that we need a reliable, repeatable way to test programs. Our solution is to write or reuse a framework to drive the test code that exercises our program API. As our program grows, with new classes and new methods added to the existing classes, we need to grow our test code as well. Experience has taught us that classes sometimes interact in unexpected ways, so we need to make sure we can run all of our tests at any time, no matter what code changes have taken place. But how do we run multiple test classes? How do we find out which tests passed and which ones failed?

In this chapter, we look at how JUnit provides the functionality to answer those questions. The chapter begins with an overview of the core JUnit concepts: the test class, methods, and annotations. Then we take a detailed look at the many testing mechanisms of JUnit 5 and the JUnit life cycle.

This chapter is written in the practical spirit of the Manning “in Action” series, looking mainly at the usage of the new core features. For comprehensive documentation of each class, method, and annotation, please visit the JUnit 5 user guide (<https://junit.org/junit5/docs/current/user-guide/>) or the JUnit 5 Javadoc (<https://junit.org/junit5/docs/current/api/>).

The chapter introduces Tested Data Systems Inc., an example company that uses the testing mechanisms. Tested Data Systems is an outsourcing company that runs several Java projects for a few customers. These projects use different frameworks and different build tools, but they have something in common: they need to be tested to ensure the high quality of the code. Some older projects are running their tests with JUnit 4; newer ones have already started with JUnit 5. The engineers have decided to acquire in-depth knowledge of JUnit 5 and to transmit it to the projects that need to move from JUnit 4 to JUnit 5.

2.1 Core annotations

The `CalculatorTest` program from chapter 1, shown in the following listing, defines a test class with a single test method `testAdd`.

Listing 2.1 CalculatorTest test case

```
import static org.junit.jupiter.api.Assertions.assertEquals;
import org.junit.jupiter.api.Test;

public class CalculatorTest {

    @Test
    public void testAdd() {
        Calculator calculator = new Calculator();
        double result = calculator.add(10, 50);
        assertEquals(60, result, 0);
    }
}
```

These are the most important concepts:

- A *test class* may be a top-level class, static member class, or an inner class annotated as `@Nested` that contains one or more test methods. Test classes cannot be abstract and must have a single constructor. The constructor must have no arguments or arguments that can be dynamically resolved at runtime through dependency injection. (We discuss the details of dependency injection in section 2.6.) A test class is allowed to be package-private as a minimum requirement for visibility. It is no longer required that test classes be public, as was the case up to JUnit 4.x. In our example, because we do not define any other constructors, we do not need to define the no-arguments constructor; the Java compiler will supply a no-args constructor.
- A *test method* is an instance method that is annotated with `@Test`, `@RepeatedTest`, `@ParameterizedTest`, `@TestFactory`, or `@TestTemplate`.
- A *life cycle method* is a method that is annotated with `@BeforeAll`, `@AfterAll`, `@BeforeEach`, or `@AfterEach`.

Test methods must not be abstract and must not return a value (the return type should be `void`).

All the examples to follow are to be found in the source code files accompanying the book.

To use the imported classes, methods, and annotations needed for the test in listing 2.1, we'll need to declare their dependencies. Most projects use a build tool to manage them. (We have chosen to use Maven, as discussed in chapter 1. Chapter 10 covers running JUnit tests from Maven.)

You need to carry out only basic tasks in Maven: configure your project through the pom.xml file, execute the mvn clean install command, and understand the command's effects. The next listing shows the minimal JUnit 5 dependencies to be used in the pom.xml Maven configuration file.

Listing 2.2 pom.xml JUnit 5 dependencies

```
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-api</artifactId>      #1
    <version>5.6.0</version>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-engine</artifactId>      #2
    <version>5.6.0</version>
    <scope>test</scope>
</dependency>
```

This shows that the minimal needed dependencies are junit-jupiter-api #1 and junit-jupiter-engine #2.

JUnit creates a new instance of the test class before invoking each @Test method to ensure the independence of test methods and prevent unintentional side effects in the test code. Also, it is a universally accepted fact that the tests must produce the same result independently of the order of their execution. Because each test method runs on a new test class instance, you cannot reuse instance variables values across test methods. One test instance is created for the execution of each test method, which is the default behavior in JUnit 5 and all previous versions.

If you annotate your test class with @TestInstance(Lifecycle.PER_CLASS), JUnit 5 will execute all test methods on the same test instance. A new test instance will be created for each test class when using this annotation.

Listing 2.3 shows the use of the JUnit 5 life cycle methods in the lifecycle.SUTTest class. One of the projects at Tested Data Systems is testing a system that will start up, receive regular and additional work, and close itself. The life cycle methods ensure that the system is initializing and then shutting down before and after each effective test. The test methods check whether the system receives regular and additional work.

Listing 2.3 Using JUnit 5 life cycle methods

```
class SUTTest {
    private static ResourceForAllTests resourceForAllTests;
    private SUT systemUnderTest;
```

```

@BeforeAll #1
static void setUpClass() {
    resourceForAllTests =
        new ResourceForAllTests("Our resource for all tests");
}

@AfterAll #2
static void tearDownClass() {
    resourceForAllTests.close();
}

@BeforeEach #3
void setUp() {
    systemUnderTest = new SUT("Our system under test");
}

@AfterEach #4
void tearDown() {
    systemUnderTest.close();
}

@Test #5
void testRegularWork() {
    boolean canReceiveRegularWork =
        systemUnderTest.canReceiveRegularWork();

    assertTrue(canReceiveRegularWork);
}

@Test #5
void testAdditionalWork() {
    boolean canReceiveAdditionalWork =
        systemUnderTest.canReceiveAdditionalWork();

    assertFalse(canReceiveAdditionalWork);
}
}

```

Following the life cycle of the test execution, you see that

- The method annotated with `@BeforeAll #1` is executed once: before all tests. This method needs to be static unless the whole test class is annotated with `@TestInstance(Lifecycle.PER_CLASS)`.
- The method annotated with `@BeforeEach #3` is executed before each test. In our case, it will be executed twice.
- The two methods annotated with `@Test #5` are executed independently.
- The method annotated with `@AfterEach #4` is executed after each test. In our case, it will be executed twice.
- The method annotated with `@AfterAll #2` is executed once: after all tests. This method needs to be static unless the whole test class is annotated with `@TestInstance(Lifecycle.PER_CLASS)`.
- In order to run this test class, you can execute from the command line: `mvn -Dtest=SUTTest.java clean install`

2.1.1 The @DisplayName annotation

The `@DisplayName` annotation can be used over classes and test methods. It helps the engineers at Tested Data Systems declare their own display name for an annotated test class or test method. Typically, this annotation is used for test reporting in IDEs and build tools. The string argument of the `@DisplayName` annotation may contain spaces, special characters, and even emojis.

The following listing demonstrates the use of the `@DisplayName` annotation through the class `displayname.DisplayNameTest`. The name that's displayed is usually a full phrase that provides significant information about the purpose of the test.

Listing 2.4 `@DisplayName` annotation

```
@DisplayName("Test class showing the @DisplayName annotation.")      #1
class DisplayNameTest {
    private SUT systemUnderTest = new SUT();

    @Test
    @DisplayName("Our system under test says hello.")                  #2
    void testHello() {
        assertEquals("Hello", systemUnderTest.hello());
    }

    @Test
    @DisplayName("👋")                                                 #3
    void testTalking() {
        assertEquals("How are you?", systemUnderTest.talk());
    }

    @Test
    void testBye() {
        assertEquals("Bye", systemUnderTest.bye());
    }
}
```

This example does the following:

- Shows the display name applied to the entire class #1
- Applies a normal text display name #2
- Uses a display name that includes an emoji #3

A test that does not have an associated display name simply shows the method name. From IntelliJ, you can run a test by right-clicking on it and then executing the *Run* command. The results of these tests in the IntelliJ IDE are shown in figure 2.1.

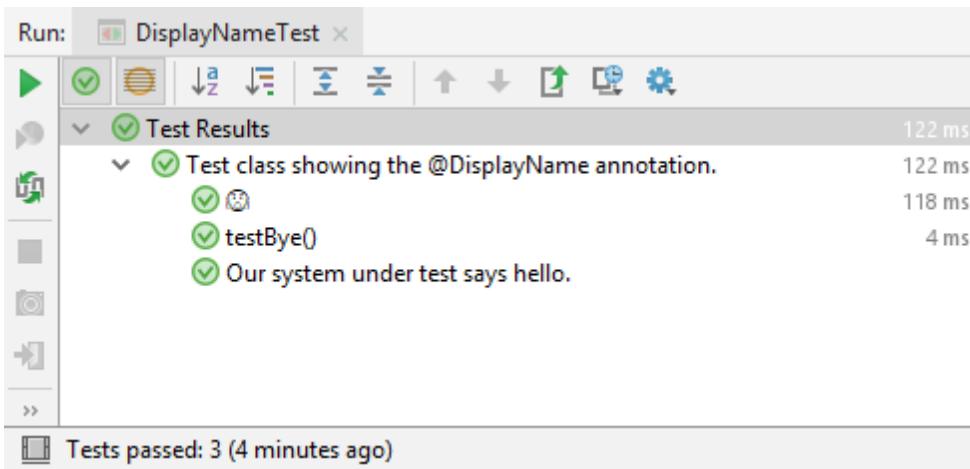


Figure 2.1 Running DisplayNameTest in IntelliJ

2.1.2 The @Disabled annotation

The `@Disabled` annotation can be used over classes and test methods. It signals that the annotated test class or test method is disabled and should not be executed. The programmers at Tested Data Systems use it to give their reasons for disabling a test so that the rest of the team knows exactly why that was done. If this annotation is applied to a class, it disables all the methods of the test. Also, the disabled tests and the reasons for their being disabled are displayed differently on each programmer's console when the programmer runs them from the IDE.

The use of the annotation is demonstrated by the classes `disabled.DisabledClassTest` and `disabled.DisabledMethodsTest`. Listings 2.5 and 2.6 show the code for these classes.

Listing 2.5 @Disabled annotation used on a test class

```
@Disabled("Feature is still under construction.") #1
class DisabledClassTest {
    private SUT systemUnderTest= new SUT("Our system under test");

    @Test
    void testRegularWork() {
        boolean canReceiveRegularWork = systemUnderTest.canReceiveRegularWork();

        assertTrue(canReceiveRegularWork);
    }

    @Test
    void testAdditionalWork() {
        boolean canReceiveAdditionalWork =
            systemUnderTest.canReceiveAdditionalWork();
    }
}
```

```
        assertFalse(canReceiveAdditionalWork);
    }

}
```

The whole testing class is disabled, and the reason is provided #1. This technique is recommended so that your colleagues (or even you) immediately understand why the test is not enabled.

Listing 2.6 @Disabled annotation used on methods

```
class DisabledMethodsTest {
    private SUT systemUnderTest= new SUT("Our system under test");

    @Test
    @Disabled
    void testRegularWork() {                                         #1
        boolean canReceiveRegularWork =
            systemUnderTest.canReceiveRegularWork ();

        assertTrue(canReceiveRegularWork);
    }

    @Test
    @Disabled("Feature still under construction.")                 #2
    void testAdditionalWork() {
        boolean canReceiveAdditionalWork =
            systemUnderTest.canReceiveAdditionalWork ();

        assertFalse(canReceiveAdditionalWork);
    }
}
```

You see that

- The code provides two tests, both disabled.
- One of the tests is disabled without a reason given #1.
- The other test is disabled with a reason that other programmers will understand #2—the recommended approach.

2.2 Nested tests

An *inner class* is a class that is a member of another class. It can access any private instance variable of the outer class, as it is effectively part of that outer class. The typical use case is when two classes are tightly coupled, and it's logical to provide direct access from the inner class to all instance variables of the outer class. For example, we may test a flight that has two types of passengers trying to board. The behavior of the flight will be described in the outer test class, while the behavior of each type of passenger will be described in its own nested class. Each passenger is able to interact with the flight. The nested tests will support following the business logic and lead to writing clearer code, as you will be able to easier follow the imbricated tests.

Following this tight-coupling idea, nested tests give the test writer more capabilities to express the relationship among several groups of tests. Inner classes may be package-private.

The Tested Data Systems company works with customers. Each customer has a gender, a first name, a last name, sometimes a middle name, and the date when they became a customer (if known). Some parameters may not be present, so the engineers are using the builder pattern to create and test a customer.

The following listing demonstrates the use of the @Nested annotation on the class NestedTestsTest. The customer being tested is John Michael Smith, and the date when he became a customer is known.

Listing 2.7 Nested tests

```
public class NestedTestsTest { #1
    private static final String FIRST_NAME = "John"; #2
    private static final String LAST_NAME = "Smith"; #2

    @Nested #3
    class BuilderTest { #3
        private String MIDDLE_NAME = "Michael";

        @Test #4
        void customerBuilder() throws ParseException { #4
            SimpleDateFormat simpleDateFormat =
                new SimpleDateFormat("MM-dd-yyyy");
            Date customerDate = simpleDateFormat.parse("04-21-2019");

            Customer customer = new Customer.Builder( #5
                Gender.MALE, FIRST_NAME, LAST_NAME) #5
                .withMiddleName(MIDDLE_NAME) #5
                .withBecomeCustomer(customerDate) #5
                .build(); #5

            assertAll(() -> { #6
                assertEquals(Gender.MALE, customer.getGender()); #6
                assertEquals(FIRST_NAME, customer.getFirstName()); #6
                assertEquals(LAST_NAME, customer.getLastName()); #6
                assertEquals(MIDDLE_NAME, customer.getMiddleName()); #6
                assertEquals(customerDate, #6
                    customer.getBecomeCustomer()); #6
            });
        }
    }
}
```

The main test is NestedTestsTest #1, which is tightly coupled with the nested test BuilderTest #3. First, NestedTestsTest defines the first name and the last name of a customer that will be used for all nested tests #2. The nested test, BuilderTest, verifies the construction of a Customer object #4 with the help of the builder pattern #5. The equality of the fields is verified at the end of the customerBuilder test #6.

The source code file has another nested class, CustomerHashCodeTest, containing two more tests. You can follow along with it.

2.3 Tagged tests

If you are familiar with JUnit 4, *tagged tests* are replacements for JUnit 4 categories. You can use the `@Tag` annotation over classes and test methods. Later, you can use tags to filter test discovery and execution.

Listing 2.8 presents the `CustomerTest` tagged class, which tests the correct creation of Tested Data Systems customers. Listing 2.9 presents the `CustomerRepositoryTest` tagged class, which tests the existence and nonexistence of customers inside a repository. One use case may be to group your tests into a few categories, based on the business logic and the things you are effectively testing. (Each test category has its own tag.) Then you may decide to run only some tests or alternate among categories, depending on your current needs.

Listing 2.8 CustomerTest tagged class

```
@Tag("individual") #1
public class CustomerTest {
    private String CUSTOMER_NAME = "John Smith";

    @Test
    void testCustomer() {
        Customer customer = new Customer(CUSTOMER_NAME);

        assertEquals("John Smith", customer.getName());
    }
}
```

The `@Tag` annotation is added to the whole `CustomerTest` class #1.

Listing 2.9 CustomersRepositoryTest tagged class

```
@Tag("repository") #1
public class CustomersRepositoryTest {
    private String CUSTOMER_NAME = "John Smith";
    private CustomersRepository repository = new CustomersRepository();

    @Test
    void testNonExistence() {
        boolean exists = repository.contains("John Smith");

        assertFalse(exists);
    }

    @Test
    void testCustomerPersistence() {
        repository.persist(new Customer(CUSTOMER_NAME));

        assertTrue(repository.contains("John Smith"));
    }
}
```

Similarly, the `@Tag` annotation is added to the whole `CustomerRepositoryTest` class #1. Here is the Maven configuration file for these tests.

Listing 2.10 pom.xml configuration file

```
<plugin>
    <artifactId>maven-surefire-plugin</artifactId>
    <version>2.22.2</version>
    <!--
        <configuration>
            <groups>individual</groups>
            <excludedGroups>repository</excludedGroups>
        </configuration>
    -->
</plugin>
```

To activate the tags, you have a few alternatives. One is to work at the level of the pom.xml configuration file. In this example, it's enough to uncomment the configuration node of the Surefire plugin #1 and run `mvn clean install`.

Another alternative in the IntelliJ IDEA IDE is to activate the tags by creating a configuration by choosing Run > Run... > Edit Configurations > Tags (JUnit 5) as the test kind (figure 2.2). This is fine when you would like to quickly make some changes about which tests to run locally. However, it is strongly recommended that you make the changes at the level of the pom.xml file—otherwise, any automated build of the project will fail.

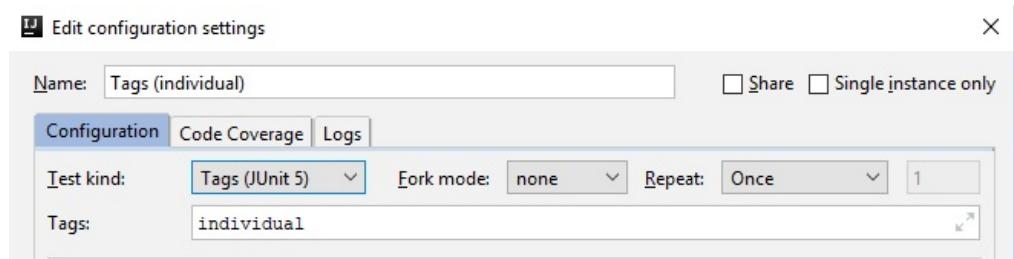


Figure 2.2 Configuring the tagged tests from the IntelliJ IDEA IDE

2.4 Assertions

To perform test validation, you use the `assert` methods provided by the JUnit Assertions class. As you can see from the previous examples, we have statically imported these methods in our test class. Alternatively, you can import the JUnit Assertions class itself, depending on your taste for static imports. Table 2.1 lists some of the most popular `assert` methods.

Table 2.1 Sample JUnit 5 assert methods

assert method	What it is used for
<code>assertAll</code>	Overloaded method. It asserts that none of the supplied executables throw exceptions. An executable is an object of type <code>org.junit.jupiter.api.function.Executable</code> .

assertArrayEquals	Overloaded method. It asserts that the expected array and the actual array are equal.
assertEquals	Overloaded method. It asserts that the expected values and the actual values are equal.
assertX(..., String message)	Assertion that delivers the supplied message to the test framework if the assertion fails.
assertX(..., Supplier<String> messageSupplier)	Assertion that delivers the supplied message to the test framework if the assertion fails. The failure message is retrieved lazily from the supplied messageSupplier.

JUnit 5 provides a lot of overloaded assertion methods. It takes many assertion methods from JUnit 4 and adds a few that can use Java 8 lambdas. All JUnit Jupiter assertions belong to the `org.junit.jupiter.api.Assertions` class and are static methods. The `assertThat()` method that works with Hamcrest matchers has been removed. The recommended approach in such a case is to use the Hamcrest `MatcherAssert.assertThat()` overloaded methods, which are more flexible and in the spirit of the Java 8 capabilities.

DEFINITION *Hamcrest* is a framework that assists with the writing of software tests in JUnit. It supports the creation of customized assertion matchers (*Hamcrest* is an anagram of *matchers*), letting us define match rules declaratively. Later in this chapter, we discuss the capabilities of *Hamcrest*.

As stated previously, one of the projects at Tested Data Systems is testing a system that starts up, receives regular and additional work, and closes itself. After we run some operations, we need to verify more than a single condition. In this case, we'll also use the lambda expressions introduced in Java 8. Lambda expressions treat functionality as a method argument and code as data. We can pass around a lambda expression as if it were an object and execute it on demand.

This section presents a few examples provided by the `assertions` package. Listing 2.11 shows some of the overloaded `assertAll` methods. The `heading` parameter allows us to recognize the group of assertions within the `assertAll()` methods. The failure message of the `assertAll()` method can provide detailed information about every particular assertion within a group. Also, we're using the `@DisplayName` annotation to provide easy-to-understand information about what the test is looking for. Our purpose is the verification of the same system under test (SUT) class that we introduced earlier.

After the `heading` parameter from the `assertAll` method, we provide the rest of the arguments as a collection of executables—a shorter, more convenient way to assert that supplied executables do not throw exceptions.

Listing 2.11 `assertAll` method

```
class AssertAllTest {
    @Test
    @DisplayName("Verify that the system starts up, processes work, and closes")
```

```

"SUT should default to not being under current verification")
void testSystemNotVerified() {
    SUT systemUnderTest = new SUT("Our system under test");

    assertAll("By default,
              SUT is not under current verification",           #1
              () -> assertEquals("Our system under test",          #2
                                  systemUnderTest.getSystemName()),      #2
              () -> assertFalse(systemUnderTest.isVerified())       #3
    );
}

@Test
@DisplayName("SUT should be under current verification")
void testSystemUnderVerification() {
    SUT systemUnderTest = new SUT("Our system under test");

    systemUnderTest.verify();

    assertAll("SUT under current verification",            #4
              () -> assertEquals("Our system under test",      #5
                                  systemUnderTest.getSystemName()),      #5
              () -> assertTrue(systemUnderTest.isVerified())     #6
    );
}
}

```

The `assertAll` method will always check all the assertions that are provided to it, even if some of them fail—if any of the executables fail, the remaining ones will still be run. That is not true for the JUnit 4 approach: if you have a few assert methods one under the other, and one of them fails, that failure will stop the execution of the others.

In the first test, the `assertAll` method receives as a parameter the message to be displayed if one of the supplied executables throws an exception #1. Then the method receives one executable to be verified with `assertEquals` #2 and one to be verified with `assertFalse` #3. The assertion conditions are brief so that they can be read at a glance.

In the second test, the `assertAll` method receives as a parameter the message to be displayed if one of the supplied executables throws an exception #4. Then it receives one executable to be verified with `assertEquals` #5 and one to be verified with `assertTrue` #6. Just like in the first test, the assertion conditions are easy to read.

The next listing shows the use of some assertion methods with messages. Thanks to `Supplier<String>`, the instructions required to create a complex message aren't provided in the case of success. We can use lambda or method references to verify our SUT; they improve performance.

Listing 2.12 Some assertion methods with messages

```

...
@Test
@DisplayName("SUT should be under current verification")
void testSystemUnderVerification() {
    systemUnderTest.verify();
    assertTrue(systemUnderTest.isVerified(),                  #1
}

```

```

        () -> "System should be under verification");      #2
    }

@Test
@DisplayName("SUT should not be under current verification")
void testSystemNotUnderVerification() {
    assertFalse(systemUnderTest.isVerified(),           #3
                () -> "System should not be under verification.");  #4
}

@Test
@DisplayName("SUT should have no current job")
void testNoJob() {
    assertNull(systemUnderTest.getCurrentJob(),          #5
                () -> "There should be no current job");       #6
}
...

```

In this example:

- A condition is verified with the help of the `assertTrue` method #1. In case of failure, a message is lazily created #2.
- A condition is verified with the help of the `assertFalse` method #3. In case of failure, a message is lazily created #4.
- The existence of an object is verified with the help of the `assertNull` method #5. In case of failure, a message is lazily created #6.

The advantage of using lambda expressions as arguments for assertion methods is that all of them are lazily created, resulting in improved performance. If the condition at #1 is fulfilled, meaning the test succeeded, the invocation of the lambda expression at #2 does not take place, which would be impossible if the test were written in the old style.

There may be situations in which you expect a test to be executed within a given interval. In our example, it is natural for the user to expect that the system under test will run the given jobs quickly. JUnit 5 offers an elegant solution for this kind of use cases.

The following listing shows the use of some `assertTimeout` and `assertTimeoutPreemptively` methods, which replace the JUnit 4 Timeout Rule. *The methods need to check whether the SUT is performant enough, meaning it is executing its jobs within a given timeout.*

Listing 2.13 Some `assertTimeout` methods

```

class AssertTimeoutTest {
    private SUT systemUnderTest = new SUT("Our system under test");

    @Test
    @DisplayName("A job is executed within a timeout")
    void testTimeout() throws InterruptedException {
        systemUnderTest.addJob(new Job("Job 1"));
        assertTimeout(ofMillis(500), () -> systemUnderTest.run(200));    #1
    }

    @Test

```

```

    @DisplayName("A job is executed preemptively within a timeout")
    void testTimeoutPreemptively() throws InterruptedException {
        systemUnderTest.addJob(new Job("Job 1"));
        assertTimeoutPreemptively(ofMillis(500),
            () -> systemUnderTest.run(200));           #2
    }
}

```

`assertTimeout` waits until the executable finishes #1. The failure message looks something like this: execution exceeded timeout of 500 ms by 193 ms.

`assertTimeoutPreemptively` stops the executable when the time has expired #2). The failure message looks like this: execution timed out after 100 ms.

In some situations, you expect a test to be executed and to throw an exception, so you may force the rest to run under inappropriate conditions or to receive inappropriate input. In our example, it is natural that the SUT that tries to run without a job assigned to it will throw an exception. Again, JUnit 5 offers an elegant solution.

Listing 2.14 shows the use of some `assertThrows` methods, which replace the JUnit 4 `ExpectedException` Rule and the `expected` attribute of the `@Test` annotation. All assertions can be made against the returned instance of `Throwable`. This makes the tests more readable, as we are verifying that the SUT is throwing exceptions: a current job is expected but not found.

Listing 2.14 Some `assertThrows` methods

```

class AssertThrowsTest {
    private SUT systemUnderTest = new SUT("Our system under test");

    @Test
    @DisplayName("An exception is expected")
    void testExpectedException() {
        assertThrows(NoJobException.class, systemUnderTest::run);      #1
    }

    @Test
    @DisplayName("An exception is caught")
    void testCatchException() {
        Throwable throwable = assertThrows(NoJobException.class,
            () -> systemUnderTest.run(1000));                         #2
        assertEquals("No jobs on the execution list!",
            throwable.getMessage());                                  #3
    }
}

```

In this example:

- We verify that the `systemUnderTest` object's call of the `run` method throws a `NoJobException` #1.
- We verify that a call to `systemUnderTest.run(1000)` throws a `NoJobException`, and we keep a reference to the thrown exception in the `throwable` variable #2.
- We check the message kept in the `throwable` exception variable #3.

2.5 Assumptions

Sometimes tests fail due to an external environment configuration or a date or time zone issue that we cannot control. We can prevent our tests from being executed under inappropriate conditions.

Assumptions verify the fulfillment of preconditions that are essential for running the tests. You can use assumptions when it does not make sense to continue the execution of a given test method. In the test report, these tests are marked as aborted.

JUnit 5 comes with a set of assumption methods suitable for use with Java 8 lambdas. The JUnit 5 assumptions are static methods belonging to the org.junit.jupiter.api.Assumptions class. The message parameter is in the last position.

JUnit 4 users should be aware that not all previously existing assumptions are provided in JUnit 5. There is no `assumeThat()` method, which we may regard as confirmation that matchers are no longer part of JUnit. The new `assumingThat()` method executes an assertion only if the assumption is fulfilled.

Suppose we have a test that needs to run only in the Windows operating system and in the Java 8 version. These preconditions are turned into JUnit 5 assumptions. A test is executed only if the assumptions are true. The following listing shows the use of some assumption methods and verifies our SUT only under the environmental conditions we imposed: the operating system needs to be Windows, and the Java version needs to be 8. If these conditions (assumptions) are not fulfilled, the check is not made.

Listing 2.15 Some assumption methods

```
class AssumptionsTest {  
    private static String EXPECTED_JAVA_VERSION = "1.8";  
    private TestsEnvironment environment = new TestsEnvironment(  
        new JavaSpecification(  
            System.getProperty("java.vm.specification.version")),  
        new OperationSystem(  
            System.getProperty("os.name"),  
            System.getProperty("os.arch"))  
    );  
  
    private SUT systemUnderTest = new SUT();  
  
    @BeforeEach  
    void setUp() {  
        assumeTrue(environment.isWindows());  
    }  
  
    @Test  
    void testNoJobToRun() {  
        assumingThat(  
            () -> environment.getJavaVersion()  
                .equals(EXPECTED_JAVA_VERSION),  
            () -> assertFalse(systemUnderTest.hasJobToRun());  
    }  
}
```

```

    @Test
    void testJobToRun() {
        assumeTrue(environment.isAmd64Architecture()); #4
        systemUnderTest.run(new Job()); #5
        assertTrue(systemUnderTest.hasJobToRun()); #6
    }
}

```

In this example:

- The `@BeforeEach` annotated method is executed before each test. The test will not run unless the assumption that the current environment is Windows is true #1.
- The first test checks that the current Java version is the expected one #2. Only if this assumption is true does it verify that no job is currently being run by the SUT #3.)
- The second test checks the current environment architecture #4. Only if this architecture is the expected one does it run a new job on the SUT #5 and verify that the system has a job to run #6.

2.6 Dependency injection in JUnit 5

The previous JUnit versions did not permit test constructors or methods to have parameters. JUnit 5 allows test constructors and methods to have parameters, but they need to be resolved through dependency injection.

The `ParameterResolver` interface dynamically resolves parameters at runtime. A parameter of a constructor or method must be resolved at runtime by a registered `ParameterResolver`. You can inject as many parameters as you want, in any order.

JUnit 5 now has three built-in resolvers. You must explicitly enable other parameter resolvers by registering appropriate extensions via `@ExtendWith`. The parameter resolvers that are automatically registered are discussed in the following sections.

2.6.1 TestInfoParameterResolver

If a constructor or method parameter is of type `TestInfo`, `TestInfoParameterResolver` supplies an instance of this type. `TestInfo` is a class whose objects are used to inject information about the currently executed test or container into the `@Test`, `@BeforeEach`, `@AfterEach`, `@BeforeAll`, and `@AfterAll` methods. Then `TestInfo` gets information about the current test: the display name, test class or method, and associated tags. The display name can be the name of the test class or test method or a custom name provided with the help of `@DisplayName`. Here's how to use a `TestInfo` parameter as an argument of a constructor and annotated methods.

Listing 2.16 `TestInfo` parameters

```

class TestInfoTest {
    TestInfoTest(TestInfo testInfo) {
        assertEquals("TestInfoTest", testInfo.getDisplayName()); #1
    }
}

```

```

@BeforeEach
void setUp(TestInfo testInfo) {
    String displayName = testInfo.getDisplayName();
    assertTrue(displayName.equals("display name of the method") || #2
               displayName.equals(                                #2
                       "testGetNameOfTheMethod(TestInfo)"));
}

@Test
void testGetNameOfTheMethod(TestInfo testInfo) {
    assertEquals("testGetNameOfTheMethod(TestInfo)",
                 testInfo.getDisplayName());                      #3
}

@Test
@DisplayName("display name of the method")
void testGetNameOfTheMethodWithDisplayNameAnnotation(TestInfo testInfo) {
    assertEquals("display name of the method",
                 testInfo.getDisplayName());                      #4
}
}

```

In this example:

- A `TestInfo` parameter is injected into the constructor and into three methods. The constructor verifies that the display name is `TestInfoTest`: its own name #1. This behavior is the default behavior, which we can vary using `@DisplayName` annotations.
- The `@BeforeEach` annotated method is executed before each test. It has an injected `TestInfo` parameter, and it verifies that the displayed name is the expected one: the name of the method or the name specified by the `@DisplayName` annotation #2.
- Both tests have an injected `TestInfo` parameter. Each parameter verifies that the displayed name is the expected one: the name of the method in the first test #3 or the name specified by the `@DisplayName` annotation in the second test #4.
- The built-in `TestInfoParameterResolver` supplies an instance of `TestInfo` that corresponds to the current container or test as the value of the expected parameters of the constructor and of the methods.

2.6.2 TestReporterParameterResolver

If a constructor or method parameter is of type `TestReporter`, `TestReporterParameterResolver` supplies an instance of this type. `TestReporter` is a functional interface and therefore can be used as the assignment target for a lambda expression or method reference. `TestReporter` has a single `publishEntry` abstract method and several overloaded `publishEntry` default methods. Parameters of type `TestReporter` can be injected into methods of test classes annotated with `@BeforeEach`, `@AfterEach`, and `@Test`. `TestReporter` can also be used to provide additional information about the test that is run. Here's how to use a `TestReporter` parameter as an argument of `@Test` annotated methods.

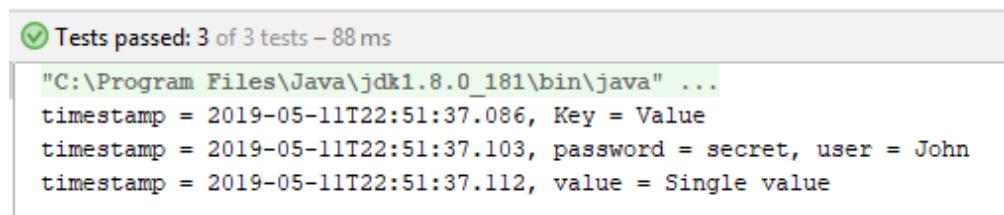
Listing 2.17 TestReporter parameters

```
class TestReporterTest {  
  
    @Test  
    void testReportSingleValue(TestReporter testReporter) {  
        testReporter.publishEntry("Single value");  
    }  
  
    @Test  
    void testReportKeyValuePair(TestReporter testReporter) {  
        testReporter.publishEntry("Key", "Value");  
    }  
  
    @Test  
    void testReportMultipleKeyValuePairs(TestReporter testReporter) {  
        Map<String, String> values = new HashMap<>();  
        values.put("user", "John");  
        values.put("password", "secret");  
  
        testReporter.publishEntry(values);  
    }  
}
```

In this example, a `TestReporter` parameter is injected into three methods:

- In the first method, it is used to publish a single value entry 1).
- In the second method, it is used to publish a key-value pair #2.
- In the third method, we construct a map #3, populate it with two key-value pairs #4, and then use it to publish the constructed map #5.
- The built-in `TestReporterParameterResolver` supplies the instance of `TestReporter` needed to publish the entries.

The result of the execution of this test is shown in figure 2.3.



```
"C:\Program Files\Java\jdk1.8.0_181\bin\java" ...  
timestamp = 2019-05-11T22:51:37.086, Key = Value  
timestamp = 2019-05-11T22:51:37.103, password = secret, user = John  
timestamp = 2019-05-11T22:51:37.112, value = Single value
```

Figure 2.3 The result of executing `TestReporterTest`

2.6.3 RepetitionInfoParameterResolver

If a parameter in a method annotated with `@RepeatedTest`, `@BeforeEach`, or `@AfterEach` is of type `RepetitionInfo`, `RepetitionInfoParameterResolver` supplies an instance of this type. Then `RepetitionInfo` gets information about the current repetition and the total

number of repetitions for a test annotated with `@RepeatedTest`. Repeated tests and examples are discussed in the following section.

2.7 Repeated tests

JUnit 5 allows us to repeat a test a specified number of times using the `@RepeatedTest` annotation, which has as a parameter the required number of repetitions. This feature can be particularly useful when conditions may change from one execution of a test to another. For example, some data that affects success may have changed between two executions of the same test, and an unexpected change in this data would be a bug that needs to be fixed.

A custom display name can be configured for each repetition using the `name` attribute of the `@RepeatedTest` annotation. The following placeholders are now supported:

- `{displayName}`—Display name of the method annotated with `@RepeatedTest`
- `{currentRepetition}`—Current repetition number
- `{totalRepetitions}`—Total number of repetitions

Listing 2.18 shows the use of repeated tests, display name placeholders, and `RepetitionInfo` parameters. The first repeated test verifies that the execution of the `add` method from the `Calculator` class is stable and always provides the same result. The second repeated test verifies that collections follow the appropriate behavior: a list receives a new element at each iteration, and a set does not get duplicate elements even if we try to insert such an element multiple times.

Listing 2.18 Repeated tests

```
public class RepeatedTestsTest {

    private static Set<Integer> integerSet = new HashSet<>();
    private static List<Integer> integerList = new ArrayList<>();

    @RepeatedTest(value = 5, name =
    "{displayName} - repetition {currentRepetition}/{totalRepetitions}") #1
    @DisplayName("Test add operation")
    void addNumber() {
        Calculator calculator = new Calculator();
        assertEquals(2, calculator.add(1, 1),
                    "1 + 1 should equal 2");
    }

    @RepeatedTest(value = 5, name = "the list contains
    {currentRepetition} elements(s), the set contains 1 element") #2
    void testAddingToCollections(TestReporter testReporter,
                                RepetitionInfo repetitionInfo) {
        integerSet.add(1);
        integerList.add(repetitionInfo.getCurrentRepetition());

        testReporter.publishEntry("Repetition number",
            String.valueOf(repetitionInfo.getCurrentRepetition())); #3
        assertEquals(1, integerSet.size()); #3
        assertEquals(repetitionInfo.getCurrentRepetition(),
                    repetitionInfo.getTotalRepetitions());
    }
}
```

```
        integerList.size());  
    }  
}
```

In this example:

- The first test is repeated five times. Each repetition shows the display name, the current repetition number, and the total number of repetitions #1.
- The second test is repeated five times. Each repetition shows the number of elements in the list (the current repetition number) and checks whether the set always has only one element #2.
- Each time the second test is repeated, the repetition number is displayed as it is injected into the `RepetitionInfo` parameter #3.

The results of executing these tests are shown in figures 2.4 and 2.5. Each invocation of a repeated test behaves like the execution of a regular `@Test` method with full support for life cycle callbacks and extensions. That is why the list and the set in the example are declared as static.

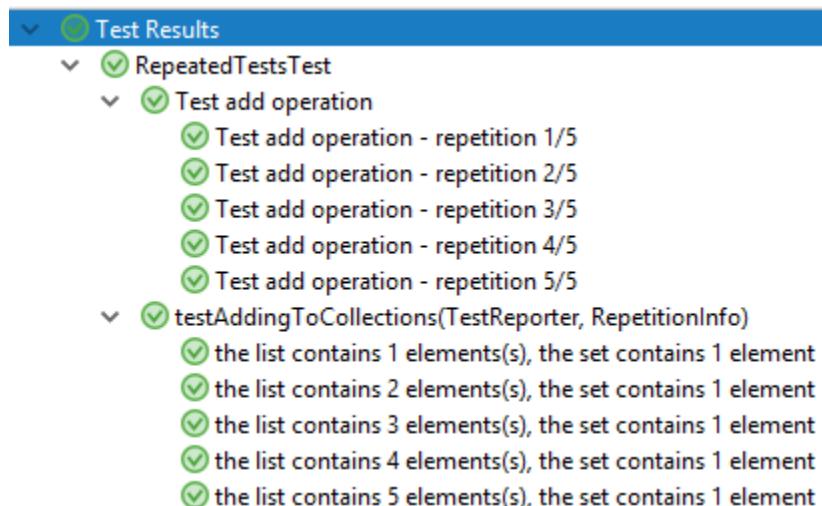


Figure 2.4 The names of the repeated tests at the time of execution

```
timestamp = 2019-05-12T17:26:34.783, Repetition number = 1
timestamp = 2019-05-12T17:26:34.798, Repetition number = 2
timestamp = 2019-05-12T17:26:34.803, Repetition number = 3
timestamp = 2019-05-12T17:26:34.813, Repetition number = 4
timestamp = 2019-05-12T17:26:34.817, Repetition number = 5
```

Figure 2.5 The messages shown on the console by the second repeated test

2.8 Parameterized tests

Parameterized tests allow a test to run multiple times with different arguments. The great benefit is that we can write a single test to be performed using arguments that check various input data. The methods are annotated with `@ParameterizedTest`. We must declare at least one source providing the arguments for each invocation. The arguments are then passed to the test method.

`@ValueSource` lets us specify a single array of literal values. At execution, this array provides a single argument for each invocation of the parameterized test. The following test checks the number of words in some phrases that are provided as parameters.

Listing 2.19 `@ValueSource` annotation

```
class ParameterizedWithValueSourceTest {
    private WordCounter wordCounter = new WordCounter();

    @ParameterizedTest
    @ValueSource(strings = {"Check three parameters",           #1
                           "JUnit in Action"})          #2
    void testWordsInSentence(String sentence) {                  #2
        assertEquals(3, wordCounter.countWords(sentence));
    }
}
```

In this example:

- We mark the test as being parameterized by using the corresponding annotation #1.
- We specify the values to be passed as an argument of the testing method #2. The testing method is executed twice: once for each argument provided by the `@ValueSource` annotation.

`@EnumSource` enables us to use `enum` instances. The annotation provides an optional `names` parameter that lets us specify which instances must be used or excluded. By default, all instances of an `enum` are used.

The following listing shows the use of the `@EnumSource` annotation to check the number of words in some phrases that are provided as `enum` instances.

Listing 2.20 `@EnumSource` annotation

```
class ParameterizedWithEnumSourceTest {
```

```

private WordCounter wordCounter = new WordCounter();

@ParameterizedTest
@EnumSource(Sentences.class)                                #1
void testWordsInSentence(Sentences sentence) {               #1
    assertEquals(3, wordCounter.countWords(sentence.value()));
}

@ParameterizedTest #2
@EnumSource(value=Sentences.class, #2
            names = { "JUNIT_IN_ACTION", "THREE_PARAMETERS" })   #2
void testSelectedWordsInSentence(Sentences sentence) {
    assertEquals(3, wordCounter.countWords(sentence.value()));
}

@ParameterizedTest #3
@EnumSource(value=Sentences.class, mode = EXCLUDE, names =      #3
            { "THREE_PARAMETERS" })                            #3
void testExcludedWordsInSentence(Sentences sentence) {
    assertEquals(3, wordCounter.countWords(sentence.value()));
}

enum Sentences {
    JUNIT_IN_ACTION("JUnit in Action"),
    SOME_PARAMETERS("Check some parameters"),
    THREE_PARAMETERS("Check three parameters");

    private final String sentence;

    Sentences(String sentence) {
        this.sentence = sentence;
    }

    public String value() {
        return sentence;
    }
}
}

```

This example has three tests, which work as follows:

- The first test is annotated as being parameterized. Then we specify the enum source as the entire `Sentences.class` #1. So this test is executed three times, once for each instance of the `Sentences` enum: `JUNIT_IN_ACTION`, `SOME_PARAMETERS`, and `THREE_PARAMETERS`.
- The second test is annotated as being parameterized. Then we specify the enum source as `Sentences.class`, but we restrict the instances to be passed to the test to `JUNIT_IN_ACTION` and `THREE_PARAMETERS` #2. So, this test will be executed twice.
- The third test is annotated as being parameterized. Then we specify the enum source as `Sentences.class`, but we exclude the `THREE_PARAMETERS` instance #3. So, this test is executed twice: for `JUNIT_IN_ACTION` and `SOME_PARAMETERS`.

We can use `@CsvSource` to express argument lists as comma-separated values (CSV), such as String literals. The following test uses the `@CsvSource` annotation to check the number of words in some phrases that are provided as parameters—this time, in CSV format.

Listing 2.21 `@CsvSource` annotation

```
class ParameterizedWithCsvSourceTest {  
    private WordCounter wordCounter = new WordCounter();  
  
    @ParameterizedTest  
    @CsvSource({"2, Unit testing", "3, JUnit in Action",  
               "4, Write solid Java code"})  
    void testWordsInSentence(int expected, String sentence) {  
        assertEquals(expected, wordCounter.countWords(sentence));  
    }  
}
```

This example has one parameterized test, which functions as follows:

- The test is parameterized as indicated by the appropriate annotation #1.
- The parameters passed to the test are from the parsed CSV strings listed in the `@CsvSource` annotation #2. So, this test is executed three times: once for each CSV line.
- Each CSV line is parsed. The first value is assigned to the `expected` parameter, and the second value is assigned to the `sentence` parameter.

`@CsvFileSource` allows us to use CSV files from the classpath. The parameterized test is executed once for each line of a CSV file. Listing 2.22 shows the use of the `@CsvFileSource` annotation, and listing 2.23 displays the contents of the `word_counter.csv` file on the classpath. The Maven build tool automatically adds the `src/test/resources` folder to the classpath. The test checks the number of words in some phrases that are provided as parameters—this time, in CSV format with a CSV file as resource input.

Listing 2.22 `@CsvFileSource` annotation

```
class ParameterizedWithCsvFileSourceTest {  
    private WordCounter wordCounter = new WordCounter();  
  
    @ParameterizedTest  
    @CsvFileSource(resources = "/word_counter.csv")  
    void testWordsInSentence(int expected, String sentence) {  
        assertEquals(expected, wordCounter.countWords(sentence));  
    }  
}
```

Listing 2.23 Contents of the `word_counter.csv` file

```
2, Unit testing  
3, JUnit in Action  
4, Write solid Java code
```

This example has one parameterized test that receives as parameters the lines indicated in the `@CsvFileSource` annotation #1. So, this test is executed three times: once for each CSV file line. The CSV file line is parsed, the first value is assigned to the `expected` parameter, and then the second value is assigned to the `sentence` parameter.

2.9 Dynamic tests

JUnit 5 introduces a dynamic new programming model that can generate tests at runtime. We write a factory method, and at runtime, it creates a series of tests to be executed. Such a factory method must be annotated with `@TestFactory`.

A `@TestFactory` method is not a regular test but a factory that generates tests. A method annotated as `@TestFactory` must return one of the following:

- A `DynamicNode` (an abstract class; `DynamicContainer` and `DynamicTest` are the instantiable concrete classes)
- An array of `DynamicNode` objects
- A Stream of `DynamicNode` objects
- A Collection of `DynamicNode` objects
- An Iterable of `DynamicNode` objects
- An Iterator of `DynamicNode` objects

As with the requirements for `@Test` annotated methods, `@TestFactory` annotated methods are allowed to be package-private as a minimum requirement for visibility, but they cannot be private or static. They may also declare parameters to be resolved by a `ParameterResolver`.

A `DynamicTest` is a test case generated at runtime, composed of a display name and an `Executable`. Because the `Executable` is a Java 8 functional interface, the implementation of a dynamic test can be provided as a lambda expression or as a method reference.

A dynamic test has a different life cycle than a standard test annotated with `@Test`. The methods annotated with `@BeforeEach` and `@AfterEach` are executed for the `@TestFactory` method but not for each dynamic test; other than these methods, there are no life cycle callbacks for individual dynamic tests. The behavior of `@BeforeAll` and `@AfterAll` remains the same; they are executed before all tests and at the end of all tests.

Listing 2.24 demonstrates dynamic tests. We want to check a predicate against a numerical value. To do so, we use a single factory to generate three tests to be created at runtime: one for a negative value, one for zero, and one for a positive value. We write one method but get three tests dynamically.

Listing 2.24 Dynamic tests

```
class DynamicTestsTest {  
  
    private PositiveNumberPredicate predicate = new PositiveNumberPredicate();  
  
    @BeforeAll  
    static void setUpClass() {  
        #1  
    }  
}
```

```

        System.out.println("@BeforeAll method");
    }

    @AfterAll
    static void tearDownClass() {                                #2
        System.out.println("@AfterAll method");                  #2
    }

    @BeforeEach
    void setUp() {                                         #3
        System.out.println("@BeforeEach method");            #3
    }

    @AfterEach
    void tearDown() {                                       #4
        System.out.println("@AfterEach method");             #4
    }

    @TestFactory
    Iterator<DynamicTest> positiveNumberPredicateTestCases() {      #5
        return asList(
            dynamicTest("negative number",                   #6
                () -> assertFalse(predicate.check(-1))),     #6
            dynamicTest("zero",                            #7
                () -> assertFalse(predicate.check(0))),       #7
            dynamicTest("positive number",                 #8
                () -> assertTrue(predicate.check(1)))        #8
        ).iterator();
    }
}

```

In this example:

- The methods annotated with `@BeforeAll` #1 and `@AfterAll` #2 are executed once as expected: at the beginning and at the end of the entire tests list, respectively.
- The methods annotated with `@BeforeEach` #3 and `@AfterEach` #4 are executed before and after the execution of the `@TestFactory` annotated method, respectively #5.
- This factory method generates three test methods labeled "negative number" #6, "zero" #7, and "positive number" #8.
- The effective behavior of each test is given by the `Executable` provided as the second parameter of the `dynamicTest` method.

The result of executing these tests is shown in figure 2.6.

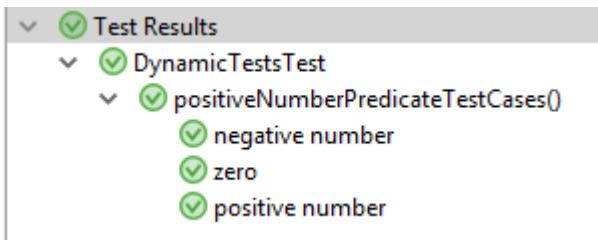


Figure 2.6 The result of executing dynamic tests

2.10 Using Hamcrest matchers

Statistics show that people easily become hooked on the unit-testing philosophy. When we get accustomed to writing tests and see how good it feels to be protected from possible mistakes, we wonder how it was possible to live without unit testing.

As we write more unit tests and assertions, we inevitably find that some assertions are big and hard to read. Our example company, Tested Data Systems, is working with customers whose data may be kept in lists. Engineers may populate a list with values like "Michael", "John", and "Edwin"; then they may search for customers like "Oliver", "Jack", and "Harry", as in the following listing. This test is intended to fail and show the description of the assertion failure.

Listing 2.25 Cumbrous JUnit assert method

```
[...]
public class HamcrestListTest {
    private List<String> values;

    @BeforeEach
    public void setUp () {                                         #1
        values = new ArrayList<>();
        values.add("Michael");
        values.add("John");
        values.add("Edwin");
    }

    @Test
    @DisplayName("List without Hamcrest")                         #2
    public void testWithoutHamcrest() {
        assertEquals(3, values.size());
        assertTrue(values.contains("Oliver"));                   #3
        || values.contains("Jack");
        || values.contains("Harry"));
    }
}
```

This example constructs a simple JUnit test like those described earlier in this chapter:

- A `@BeforeEach` fixture #1 initializes some data for the test. A single test method is used #2.

- This test method makes a long, hard-to-read assertion #3. (Maybe the assertion itself is not too hard to read, but what it does definitely is not obvious at first glance.)
- The goal is to simplify the assertion made in the test method.

To solve this problem, Tested Data Systems uses a library of matchers for building test expressions: Hamcrest. Hamcrest (<https://code.google.com/archive/p/hamcrest>) is a library that contains a lot of helpful *matcher* objects (also known as *constraints* or *predicates*) ported in several languages, such as Java, C++, Objective-C, Python, and PHP.

The Hamcrest library

Hamcrest is not a testing framework itself, but it helps us declaratively specify simple matching rules. These matching rules can be used in many situations, but they are particularly helpful for unit testing.

The next listing provides the same test method as listing 2.25, this time written with the Hamcrest library.

Listing 2.26 Using the Hamcrest library

```
[...]
import static org.hamcrest.CoreMatchers.anyOf;                      #1
import static org.hamcrest.CoreMatchers.equalTo;                      #1
import static org.hamcrest.MatcherAssert.assertThat;                  #1
import static org.hamcrest.Matchers.*;                                #1
[...]

@DisplayName("List with Hamcrest")
public void testListWithHamcrest() {
    assertThat(values, hasSize(3));
    assertThat(values, hasItem(anyOf(equalTo("Oliver"),
        equalTo("Jack"), equalTo("Harry"))));                         #2
}
[...]
```

This example adds a test method that imports the needed matchers and the `assertThat` method #1 and then constructs a test method. The test method uses one of the most powerful features of the matchers: they can nest #2. What Hamcrest gives us that standard assertions do not is a human-readable description of an assertion failure. Using assertion code with or without Hamcrest matchers is a personal preference.

The examples in the previous two listings construct a `List` with the customers "Michael", "John", and "Edwin" as elements. After that, the code asserts the presence of "Oliver", "Jack", or "Harry", so the tests will fail on purpose. The result from the execution without Hamcrest is shown in figure 2.7, and the result from the execution with Hamcrest is shown in figure 2.8. As the figures show, the test that uses Hamcrest provides more details.

```

org.opentest4j.AssertionFailedError:  

Expected :<true>  

Actual   :<false>  

<Click to see difference>  
  

<4 internal calls>  

at com.manning.junitbook.ch02.core.hamcrest.HamcrestListTest.testListWithoutHamcrest(HamcrestListTest.java:53) <19 internal calls>  

at java.util.ArrayList.forEach(ArrayList.java:1257) <9 internal calls>  

at java.util.ArrayList.forEach(ArrayList.java:1257) <21 internal calls>

```

Figure 2.7 The result of the test execution without Hamcrest

```

java.lang.AssertionError:  

Expected: a collection containing ("Oliver" or "Jack" or "Harry")  

but: mismatches were: [was "John", was "Michael", was "Edwin"]  
  

at org.hamcrest.MatcherAssert.assertThat(MatcherAssert.java:18)  

at org.hamcrest.MatcherAssert.assertThat(MatcherAssert.java:6)  

at com.manning.junitbook.ch02.hamcrest.HamcrestListTest.testListWithHamcrest(HamcrestListTest.java:60) <19 internal calls>  

at java.util.ArrayList.forEach(ArrayList.java:1257) <9 internal calls>  

at java.util.ArrayList.forEach(ArrayList.java:1257) <21 internal calls>

```

Figure 2.8 The result of the test execution with Hamcrest

To use Hamcrest in our projects, we need to add the required dependency to the pom.xml file.

Listing 2.27 pom.xml Hamcrest dependency

```

<dependency>  

  <groupId>org.hamcrest</groupId>  

  <artifactId>hamcrest-library</artifactId>  

  <version>2.1</version>  

  <scope>test</scope>  

</dependency>

```

To use Hamcrest in JUnit 4, you have to use the `assertThat` method from the `org.junit.Assert` class. But as explained earlier in this chapter, JUnit 5 removes the `assertThat()` method. The user guide justifies the decision this way:

[...] `org.junit.jupiter.api.Assertions` class does not provide an `assertThat()` method like the one found in JUnit 4's `org.junit.Assert` class, which accepts a Hamcrest Matcher. Instead, developers are encouraged to use the built-in support for matchers provided by third-party assertion libraries.

This text means that if we want to use Hamcrest matchers, we have to use the `assertThat()` methods of the `org.hamcrest.MatcherAssert` class. As the previous examples illustrated, the overloaded methods take two or three method parameters:

- An error message shown when the assertion fails (optional)
- The actual value or object
- A `Matcher` object for the expected value

To create the `Matcher` object, we need to use one of the static factory methods provided by the `org.hamcrest.Matchers` class, as shown in table 2.2.

Table 2.2 Sample of common Hamcrest static factory methods

Factory method	Logical
<code>anything</code>	Matches absolutely anything; useful when we want to make the assert statement more readable
<code>is</code>	Used only to improve the readability of statements
<code>allOf</code>	Checks whether all contained matchers match (like the <code>&&</code> operator)
<code>anyOf</code>	Checks whether any of the contained matchers match (like the <code> </code> operator)
<code>not</code>	Inverts the meaning of the contained matchers (like the <code>!</code> operator in Java)
<code>instanceOf</code>	Check whether objects are instances of one another
<code>sameInstance</code>	Tests object identity
<code>nullValue, notNullValue</code>	Tests for null or non-null values
<code>hasProperty</code>	Tests whether a Java Bean has a certain property
<code>hasEntry, hasKey, hasValue</code>	Tests whether a given Map has a given entry, key, or value
<code>hasItem, hasItems</code>	Tests a given collection for the presence of an item or items
<code>closeTo, greaterThan, greaterThanOrEqualTo, lessThan, lessThanOrEqualTo</code>	Tests whether given numbers are close to, greater than, greater than or equal, less than, or less than or equal to a given value
<code>equalToIgnoringCase</code>	Tests whether a given string equals another one, ignoring the case
<code>equalToIgnoringWhiteSpace</code>	Tests whether a given string equals another one, ignoring the white space
<code>containsString, endsWith, startsWith</code>	Tests whether a given string contains, starts with, or ends with a certain string

All of these methods are pretty straightforward to read and use. Also, remember that we can compose them into one another.

For each service provided to customers, Tested Data Systems charges a particular price. The following code tests the properties of a customer and some service prices using a few Hamcrest methods.

Listing 2.28 A few Hamcrest static factory methods

```
public class HamcrestMatchersTest {  
  
    private static String FIRST_NAME = "John";  
    private static String LAST_NAME = "Smith";  
    private static Customer customer = new Customer(FIRST_NAME, LAST_NAME);  
}
```

```

@Test
@DisplayName("Hamcrest is, anyOf, allOf")
public void testHamcrestIs() {
    int price1 = 1, price2 = 1, price3 = 2;

    assertThat(1, is(price1));                                #1
    assertThat(1, anyOf(is(price2), is(price3)));           #1
    assertThat(1, allOf(is(price1), is(price2))));          #1
}

@Test
@DisplayName("Null expected")
void testNull() {
    assertThat(null, nullValue());                           #2
}

@Test
@DisplayName("Object expected")
void testNotNull() {
    assertThat(customer, notNullValue());                   #3
}

@Test
@DisplayName("Check correct customer properties")
void checkCorrectCustomerProperties() {
    assertThat(customer, allOf(
        hasProperty("firstName", is(FIRST_NAME)),          #4
        hasProperty("lastName", is(LAST_NAME))              #4
    ));                                                       #4
}
}

```

This example shows

- The use of the `is`, `anyOf`, and `allOf` methods #1
- The use of the `nullValue` method #2
- The use of the `notNullValue` method #3
- The use of the `assertThat` method #1, #2, and #3, as described in table 2.2

We have also constructed a `Customer` object and check its properties with the help of the `hasProperty` method #4.

Last but not least, Hamcrest is extremely extensible. Writing matchers that check a certain condition is easy: we implement the `Matcher` interface and an appropriately named factory method.

Chapter 3 analyzes the architectures of JUnit 4 and JUnit 5 and explains the move to the new architectures.

2.11 Summary

This chapter has covered the following:

- The core JUnit 5 classes related to assertions and assumptions.

- Using JUnit 5 methods and annotations: the methods from the assertions and assumptions classes, and annotations like `@Test`, `@DisplayName`, and `@Disabled`
- The life cycle of a JUnit 5 test, and controlling it through the `@BeforeEach`, `@AfterEach`, `@BeforeAll`, and `@AfterAll` annotations
- Applying JUnit 5 capabilities to create nested tests and tagged tests (annotations: `@NestedTest`, `@Tag`)
- Implementing dependency injection with the help of test constructors and methods with parameters
- Applying dependency injection by using different parameter resolvers (`TestInfoParameterResolver`, `TestReporterParameterResolver`)
- Implementing repeated tests (annotation: `@RepeatedTest`) as another application of dependency injection
- Parameterized tests, a very flexible tool for testing that consumes different data sets and dynamic tests created at runtime (annotations: `@ParameterizedTest`, `@TestFactory`)
- Using Hamcrest matchers to simplify assertions

3

JUnit architecture

This chapter covers

- Demonstrating the concept and importance of software architecture
- Comparing the JUnit 4 and JUnit 5 architectures

Architecture is the stuff that's hard to change later. And there should be as little of that stuff as possible.

—Martin Fowler

So far, we have made and used a JUnit survey (chapter 1). We've also looked at JUnit core classes and methods, and how they interact with each other, as well as how to use the many features of JUnit 5 (chapter 2).

This chapter looks at the architecture of the two most recent JUnit versions. It discusses the architecture of JUnit 4 to show where JUnit 5 started, where the big changes between versions are, and which shortcomings had to be addressed.

3.1 The concept and importance of software architecture

Software architecture refers to the fundamental structures of a software system. Such a system must be created in an organized manner. A software system structure comprises software elements, relationships among those elements, and properties of both elements and relationships.

Software architecture is like the architecture of a building. The architecture of a software system represents the foundation on which everything else sits, represented by the bottom boxes in figure 3.1. But architectural elements are harder to move around and replace than

those in physical architecture because we have to move all the things on top of them to implement the changes.

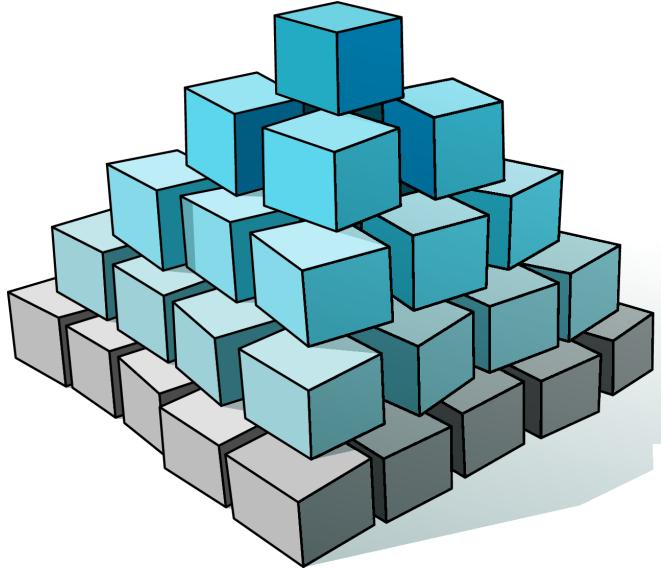


Figure 3.1 The architecture represents the foundation of the system. At the bottom level, the architecture is hard to move around and replace. The intermediary boxes represent the design; the top ones represent the idioms.

The corollary to Fowler's definition of architecture is that we should construct the architectural elements so that they're easier to replace.

The architecture of JUnit 5 emerged from the shortcomings of JUnit 4. To understand the great impact of the architecture on the entire system, read the stories in the next two sections.

3.1.1 Story 1 – the telephone directories books

Once upon a time, two companies published telephone directories. The companies' books were identical in shape, size, and cost.

Neither company could gain a significant advantage. Clerks were buying both products for \$1; they couldn't tell which book was better because the books contained similar information. Finally, company A hired a troubleshooter.

The troubleshooter thought for a while and found a solution: "Let's make the format of our own book smaller than our competitor's book while keeping the same information."

The fact is that when books are the same size, they are put on office desks, one near the other. When one book is large and flat, and the second is small but plump, however, the small one is most often put on top of the large one (figure 3.2).

At the end of the month, the customer will understand that he has used only the top one; the large one has never been opened. So why would he spend a dollar on the larger one?

This is one of the architectural changes from JUnit 4 to JUnit 5: smaller works better.

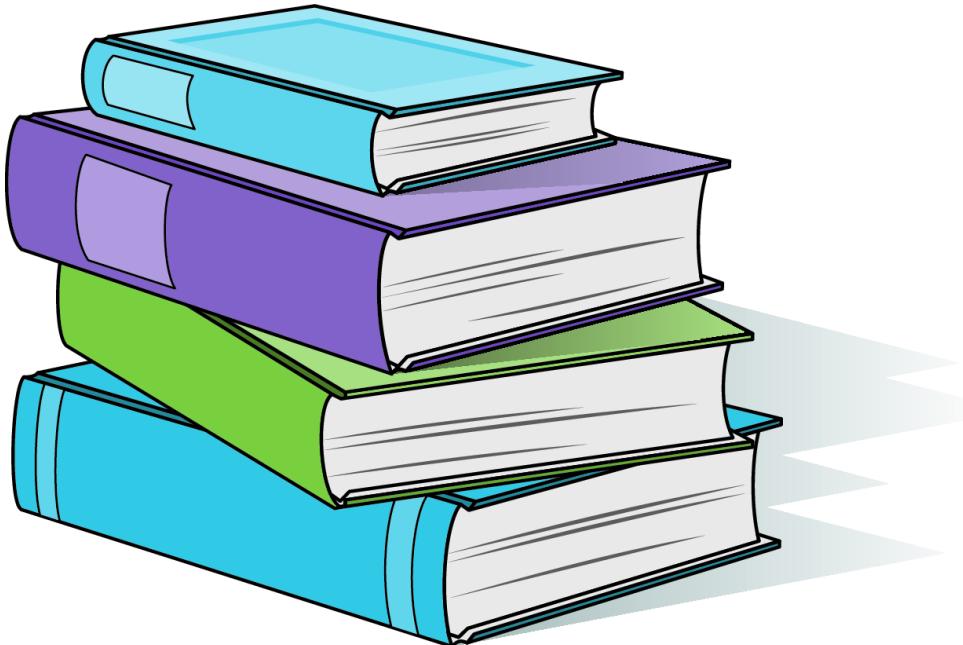


Figure 3.2 Changing the size of books is an architectural change that has a big impact. A small item is used and moved more easily and frequently than a big one.

3.1.2 Story 2 – the sneakers manufacturing company

A company has begun manufacturing sneakers in low-cost locations, so the net cost should be low. But losses from sneaker thefts have been unexpectedly huge. The company tried to get more guards, but hiring guards increases the final price. The company needed a way to decrease the number of stolen sneakers without additional costs.

The company's analysts arrived at this solution: produce the left and right shoes in different locations (figure 3.3). Consequently, sneaker thefts will decrease dramatically.

This scenario is another architectural change from JUnit 4 to JUnit 5: modularity improves the work. When we need some functionality, we can address to the module that is implementing it. We depend on and load only a specific module instead of the whole testing framework, which saves time and memory.



Figure 3.3 Separating the production locations for left and right sneakers represents a large architectural change.

3.2 JUnit 4 architecture

This book focuses on JUnit 5, but it also discusses JUnit 4 for some important reasons. One reason is that a lot of legacy code uses it. Also, migration from JUnit 4 to JUnit 5 does not happen instantly (if ever), and projects may work with a hybrid JUnit 4-and-JUnit 5 approach.

Moreover, JUnit 5 was designed to work with old JUnit 4 code in existing legacy projects through JUnit Vintage. (Chapter 4 clarifies the best times to postpone or cancel migration from JUnit 4.)

The shortcomings of JUnit 4 helped JUnit 5 emerge, however.

This section emphasizes some JUnit 4 characteristics that clearly show the need for the JUnit 5 approach: modularity, runners, and rules.

3.2.1 JUnit 4 modularity

JUnit 4, released in 2006, has a simple, monolithic architecture. All its functionality is concentrated inside a single JAR file (figure 3.4). If a programmer wants to use JUnit 4 in a project, all they need to do is add that JAR file on the classpath.



Figure 3.4 The monolithic architecture of JUnit 4: a single JAR file

3.2.2 JUnit 4 runners

A JUnit 4 *runner* is a class that extends the JUnit 4 abstract `Runner` class. A JUnit 4 runner is responsible for running JUnit tests. JUnit 4 remains a single JAR file, but it's generally necessary to extend the functionality of this file. In other words, developers have the opportunity to add custom features to the functionality, such as executing additional things before and after running a test.

Runners may extend a test behavior by using reflection. Reflection breaks encapsulation, of course, but this technique was the only way to provide extensibility in JUnit 4 and earlier

versions—one good reason for the JUnit 5 approach. We may need to keep existing JUnit 4 runners in our code for some time.

(Extensions are the JUnit 5 equivalent of JUnit 4 runners and are discussed in chapter 4.)

In practice, we can work with existing runners, such as the runners for the Spring framework or for mocking objects with Mockito (chapter 8). We consider as very useful to demonstrate the creation and usage of custom runners, as they reveal the principles of runners in general. We can extend the JUnit 4 abstract Runner class, override its methods and working with reflection. This is an approach that breaks the encapsulation but was the single possibility to add custom functionality to the JUnit 4 one. Revealing the shortcomings of working with custom runners in JUnit 4 opens the gate to understanding the capabilities and advantages of JUnit 5 extensions.

To demonstrate the use of JUnit 4 runners, we come back to our Calculator class (listing 3.1).

Listing 3.1 The test Calculator class

```
public class Calculator {  
    public double add(double number1, double number2) {  
        return number1 + number2;  
    }  
}
```

We would like to enrich the behavior of tests that are using this class by introducing additional action before the execution of the test suite. We'll create a custom runner and use it as an argument of the @RunWith annotation to add custom features to the original JUnit functionality. Listing 3.2 demonstrates how to build a CustomTestRunner.

Listing 3.2 The CustomTestRunner class

```
public class CustomTestRunner extends Runner {  
  
    private Class<?> testedClass; #A  
  
    public CustomTestRunner(Class<?> testedClass) {  
        this.testedClass = testedClass; #A  
    }  
  
    @Override  
    public Description getDescription() {  
        return Description #B  
            .createTestDescription(testedClass, #B  
                this.getClass().getSimpleName() + " description"); #B  
    }  
  
    @Override  
    public void run(RunNotifier notifier) {  
        System.out.println("Running tests with " + #C  
            this.getClass().getSimpleName() + ": " + testedClass);  
        try {  
            Object testObject = testedClass.newInstance(); #D  
            for (Method method : testedClass.getMethods()) { #D  
                if (method.isAnnotationPresent(Test.class)) { #D  
                    ...  
                }  
            }  
        } catch (Exception e) {  
            e.printStackTrace();  
        }  
    }  
}
```

```

        notifier.fireTestStarted(Description           #E
                                .createTestDescription(testedClass,
                                                       methodName())); #E
        method.invoke(testObject);                      #F
        notifier.fireTestFinished(Description          #G
                                .createTestDescription(testedClass,
                                                       methodName())); #G
    }
}
} catch (InstantiationException | IllegalAccessException |
         InvocationTargetException e) {
    throw new RuntimeException(e);
}
}
}

```

In this listing:

- We keep a reference to the tested class, which is initialized inside the constructor (#A).
- We override the abstract `getDescription` method, inherited from the abstract `Runner` class. This method contains information that is later exported and may be used by various tools (#B).
- We override the abstract `run` method, inherited from the abstract `Runner` class. Inside it, we create an instance of the tested class (#C).
- We browse all public methods from the tested class and we filter the `@Test` annotated ones (#D).
- We invoke `fireTestStarted` to tell listeners that an atomic test is about to start (#E).
- We reflectively invoke the original `@Test` annotated method (#F).
- We invoke `fireTestFinished` to tell listeners that an atomic test has finished (#G).

We'll use the `CustomTestRunner` class as an argument of the `@RunWith` annotation, to be applied to the `CalculatorTest` class (listing 3.3).

Listing 3.3 The CalculatorTest class

```

@RunWith(CustomTestRunner.class)
public class CalculatorTest
{
    @Test
    public void testAdd()
    {
        Calculator calculator = new Calculator();
        double result = calculator.add( 10, 50 );
        assertEquals( 60, result, 0 );
    }
}

```

Figure 3.5 shows the result of the execution of this test.

```
Tests passed: 1 of 1 test - 14 ms
"C:\Program Files\Java\jdk1.8.0_181\bin\java" ...
Running tests with CustomTestRunner: class com.manning.junitbook.runners.CalculatorTest

Process finished with exit code 0
```

Figure 3.5 The result of the execution of `CalculatorTest` with a custom runner

This example demonstrates how to add custom functionality to JUnit 4. Using the recognized terminology, we are extending the JUnit 4 functionality.

3.2.3 JUnit 4 rules

A JUnit 4 *rule* is a component that intercepts test method calls; it allows us to do something before a test method is run and something else after a test method has run. Rules are specific to JUnit 4.

To add behavior to the executed tests, we must use the `@Rule` annotation on `TestRule` fields. This technique increases the flexibility of tests by creating objects that can be used and configured in the test methods.

As with runners, we may need to keep existing ones into our code for some time, as migration to JUnit 5 mechanisms is not straightforward. The equivalent approach in JUnit 5 forces programmers to implement extensions (part 4).

We would like to add two more methods to the `Calculator` class, as shown in listing 3.4.

Listing 3.4 The Calculator class with additional functionality

```
public class Calculator {

    ...

    public double sqrt(double x) {                                #A
        if (x < 0) {
            throw new
                IllegalArgumentException("Cannot extract the square      #B
                                              root of a negative value"); #B
        }
        return Math.sqrt(x);                                     #A
    }

    public double divide(double x, double y) {                   #C
        if (y == 0) {
            throw new ArithmeticException("Cannot divide by zero"); #D
        }
        return x/y;                                            #C
    }
}
```

The new logic of the `Calculator` class does the following:

- Declares a method to calculate the square root of a number (#A). In case the number is negative, an exception containing a particular message is created and thrown (#B).
- Declares a method to divide two numbers (#C). In case the second number is zero, an exception containing a particular message is created and thrown (#D).

We would like to test the newly introduced methods and see whether the appropriate exceptions are thrown for particular inputs.

Listing 3.5 specifies which exception message is expected during the execution of the test code, using the new functionality of the `Calculator` class.

Listing 3.5 The RuleExceptionTester class

```
public class RuleExceptionTester {
    @Rule
    public ExpectedException expectedException = #A
        ExpectedException.none(); #A

    private Calculator calculator = new Calculator(); #B

    @Test
    public void expectIllegalArgumentException() {
        expectedException.expect(IllegalArgumentException.class); #C
        expectedException.expectMessage("Cannot extract the square root #D
            of a negative value"); #D
        calculator.sqrt(-1); #E
    }

    @Test
    public void expectArithmeticException() {
        expectedException.expect(ArithmeticException.class); #F
        expectedException.expectMessage("Cannot divide by zero"); #G
        calculator.divide(1, 0); #H
    }
}
```

In this example:

- We declare an `ExpectedException` field annotated with `@Rule`. The `@Rule` annotation must be applied to a public nonstatic field or a public nonstatic method (#A). The `ExpectedException.none()` factory method simply creates an unconfigured `ExpectedException`.
- We initialize an instance of the `Calculator` class whose functionality we are testing (#B).
- The `ExpectedException` is configured to keep the type of exception (#C) and message (#D) before it is thrown by invoking the `sqrt` method (#E).
- The `ExpectedException` is configured to keep the type of exception (#F) and message #G before it is thrown by invoking the `divide` method (#H).

In some situations, we need to work with temporary resources, such as creating files and folders to store only test-specific information. The `TemporaryFolder` Rule allows us to

create files and folders that should be deleted when the test method finishes (whether the test passes or fails).

Listing 3.6 shows a `TemporaryFolder` field annotated with `@Rule`, and we are testing the existence of these temporary resources.

Listing 3.6 The RuleTester class

```
public class RuleTester {  
    @Rule  
    public TemporaryFolder folder = new TemporaryFolder(); #A  
    private static File createdFolder; #B  
    private static File createdFile; #B  
  
    @Test  
    public void testTemporaryFolder() throws IOException {  
        createdFolder = folder.newFolder("createdFolder"); #C  
        createdFile = folder.newFile("createdFile.txt"); #C  
        assertTrue(createdFolder.exists()); #D  
        assertTrue(createdFile.exists()); #D  
    }  
  
    @AfterClass  
    public static void cleanUpAfterAllTestsRan() {  
        assertFalse(createdFolder.exists()); #E  
        assertFalse(createdFile.exists()); #E  
    }  
}
```

In this example:

- We declare a `TemporaryFolder` field annotated with `@Rule` and initialize it. The `@Rule` annotation must be applied to a public field or a public method (#A).
- We declare the static fields `createdFolder` and `createdFile` #B.
- We use the `TemporaryFolder` field to create a folder and a file (#C), which are located in the `Temp` folder of our user profile in the operating system.
- We check the existence of the temporary folder and of the temporary file (#D).
- At the end of the execution of the tests, we check that the temporary resources do not exist any longer (#E).

We have demonstrated that two existing JUnit 4 rules—`ExpectedException` and `TemporaryFolder`—are working.

Now we would like to write our own custom rule, which is useful for providing our own behavior before and after a test is run. We might like to start a process before the execution of a test and stop it after that or to connect to a database before the execution of a test and tear it down after that.

To write our own custom rule, we'll have to create a class that implements the `TestRule` interface. Consequently, we'll override the `apply(Statement, Description)` method, which must return an instance of `Statement`. Such an object represents the tests within the JUnit run time, and `Statement#evaluate()` runs them. The `Description` object describes

the individual test; we can use this object to read information about the test through reflection.

Listing 3.7 The CustomRule class

```
public class CustomRule implements TestRule { #A
    private Statement base; #B
    private Description description; #B

    @Override
    public Statement apply(Statement base, Description description) { #C
        this.base = base;
        this.description = description;
        return new CustomStatement(base, description); #C
    }

}
```

In this example:

- We declare a `CustomRule` class that implements the `TestRule` interface (#A).
- We keep references to a `Statement` field and to a `Description` field (#B), and we use them into the `apply` method that returns a `CustomStatement` (#C).

Listing 3.8 The CustomStatement class

```
public class CustomStatement extends Statement { #A
    private Statement base; #B
    private Description description; #B

    public CustomStatement(Statement base, Description description) { #C
        this.base = base;
        this.description = description;
    }

    @Override
    public void evaluate() throws Throwable { #D
        System.out.println(this.getClass().getSimpleName() + " " + #D
            description.getMethodName() + " has started"); #D
        try { #D
            base.evaluate(); #D
        } finally { #D
            System.out.println(this.getClass().getSimpleName() + " " + #D
                description.getMethodName() + " has finished"); #D
        }
    }
}
```

In this example:

- We declare our `CustomStatement` class that extends the `Statement` class (#A).
- We keep references to a `Statement` field and a `Description` field (#B), and we use them as arguments of the constructor (#C).
- We override the inherited `evaluate` method and call `base.evaluate()` inside it

(#D).

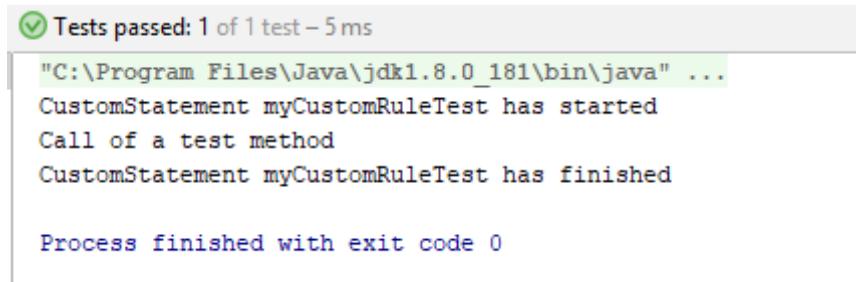
Listing 3.9 The CustomRuleTester class

```
public class CustomRuleTester {  
  
    @Rule  
    public CustomRule myRule = new CustomRule();  
    #A  
    #A  
  
    @Test  
    public void myCustomRuleTest() {  
        System.out.println("Call of a test method");  
    }  
    #B  
    #B  
}
```

In this example, we use the previously defined `CustomRule` as follows:

- We declare a public `CustomRule` field and annotate it with `@Rule` (#A).
- We create the `myCustomRuleTest` method and annotate it with `@Test` (#B).

Figure 3.6 shows the result of the execution of this test. The effective execution of the test is surrounded by additional messages provided to the `evaluate` method of the `CustomStatement` class.



The screenshot shows a terminal window with the following output:

```
Tests passed: 1 of 1 test - 5 ms  
"C:\Program Files\Java\jdk1.8.0_181\bin\java" ...  
CustomStatement myCustomRuleTest has started  
Call of a test method  
CustomStatement myCustomRuleTest has finished  
  
Process finished with exit code 0
```

Figure 3.6 The result of the execution of `CustomRuleTester`

As an alternative, we provide the `CustomRuleTester2` class, which keeps the `CustomRule` field private and exposes it through a public getter annotated with `@Rule`. This annotation works only on public and nonstatic fields and methods.

Listing 3.10 The CustomRuleTester2 class

```
public class CustomRuleTester2 {  
  
    private CustomRule myRule = new CustomRule();  
  
    @Rule  
    public CustomRule getMyRule() {  
        return myRule;  
    }  
}
```

```
@Test  
public void myCustomRuleTest() {  
    System.out.println("Call of a test method");  
}  
}
```

The result of the execution of this test is the same as shown in figure 3.6. The effective execution of the test is surrounded by the additional messages provided to the evaluate method of the CustomStatement class.

Writing our own rules is very useful when we need to add custom behavior to tests. Typical use cases include allocating a resource before a test is run and freeing it afterward; starting a process before the execution of a test and stopping it after that; and connecting to a database before the execution of a test and tearing it down after that.

So through the use of runners and rules, we can extend the monolithic architecture of JUnit 4.

We may still encounter a lot of JUnit 4 code that uses runners and rules. Besides, migration to the equivalent mechanisms of JUnit 5 (extensions) is not so straightforward. The equivalent approach in JUnit 5 forces the programmer to implement extensions, so we may keep them in our code for a while, even if we move our work to JUnit 5.

One more thing to look for in these examples of JUnit 4 runners and rules is the Maven pom.xml configuration file.

Listing 3.11 The pom.xml configuration file

```
<dependencies>  
    <dependency>  
        <groupId>org.junit.vintage</groupId>  
        <artifactId>junit-vintage-engine</artifactId>  
        <version>5.6.0</version>  
        <scope>provided</scope>  
    </dependency>  
</dependencies>
```

The only needed dependency is junit-vintage-engine, which belongs to JUnit 5. JUnit 5 Vintage (a component of the JUnit 5 architecture, discussed in section 3.3.4) ensures backward compatibility with previous versions of JUnit. Working with JUnit 5 Vintage, Maven transitively accesses the dependency to JUnit 4 as well. As it may be tedious and time-consuming to move the existing tests to JUnit 5, introducing this dependency may ensure the coexistence of JUnit 4 and JUnit 5 tests within the same project.

3.2.4 Shortcomings of the JUnit 4 architecture

Despite its apparent simplicity, this architecture generated a series of problems that grew as time passed. JUnit was used not only by programmers, but also by many software programs, such as IDEs (Eclipse, NetBeans, and IntelliJ) and build tools (ANT and Maven). Also, JUnit 4 was monolithic and was not designed to interact with these kinds of tools. But everyone wanted to work with such a popular, simple, and useful framework.

The API provided by JUnit 4 was not flexible enough. Consequently, the IDEs and tools that were using JUnit 4 were tightly coupled to the unit testing framework. The API (Application Programming Interface) was not designed to provide the classes and methods that were appropriate for the interaction with them. These tools needed to go into the JUnit classes and even use reflection to get the necessary information. If designers or JUnit decided to change the name of a private variable, this change could have affected the tools that were accessing it reflectively. Maintaining the interaction with JUnit 4 was hard work, so the popularity and simplicity of the framework became obstacles.

Consequently, because everyone was using the same single JAR and because all tools and IDEs were so tightly coupled to it, evolution possibilities for JUnit 4 were seriously reduced. A new API designed for such kind of tools and a new architecture resulted. As in the stories presented in section 3.1, a need for smaller and modular things arose. This need was met in JUnit 5.

3.3 JUnit 5 architecture

It was time for a new approach. It hasn't come instantly, it required time and analysis. The shortcomings of JUnit 4 have become a very good input for the needed improvements. Architects knew the problems, and they decided to go on the path of breaking the single JUnit 4 jar file into a few ones with reduced size.

3.3.1 JUnit 5 modularity

A new modular approach was needed to allow the evolution of the JUnit framework. The architecture had to allow JUnit to interact with different programmatic clients that used different tools and IDEs. The logical separation of concerns required

- An API to write tests, dedicated mainly to developers
- A mechanism for discovering and running the tests
- An API to allow easy interaction with IDEs and tools and to run the tests from them

As a consequence, the JUnit 5 architecture contained three modules (figure 3.7):

- *JUnit Platform*, which serves as a foundation for launching testing frameworks on the Java Virtual Machine (JVM). It also provides an API to launch tests from the console, IDEs, or build tools.
- *JUnit Jupiter*, the combination of the new programming and extension model for writing tests and extensions in JUnit 5. The name comes from the fifth planet of our solar system, which is also the largest.
- *JUnit Vintage*, a test engine for running JUnit 3- and JUnit 4-based tests on the platform, ensuring backward compatibility.



Figure 3.7 The modular architecture of JUnit 5

3.3.2 JUnit 5 platform

Going further with the modularity idea, this chapter takes a brief look at the artifacts contained in the JUnit 5 Platform (figure 3.8):

- `junit-platform-commons`, an internal common library of JUnit intended solely for use within the JUnit framework. Any use by external parties is not supported.
- `junit-platform-console`, which provides support for discovering and executing tests on the JUnit Platform from the console.
- `junit-platform-console-standalone`, an executable JAR with all dependencies included. This artifact is used by Console Launcher, a command-line Java application that lets us launch the JUnit Platform from the console. It can be used to run JUnit Vintage and JUnit Jupiter tests, for example, and to print test execution results to the console.
- `junit-platform-engine`, a public API for test engines.
- `junit-platform-launcher`, a public API for configuring and launching test plans; typically used by IDEs and build tools.
- `junit-platform-runner`, a runner for executing tests and test suites on the JUnit Platform in a JUnit 4 environment.
- `junit-platform-suite-api`, which contains the annotations for configuring test suites on the JUnit Platform.
- `junit-platform-surefire-provider`, which provides support for discovering and executing tests on the JUnit Platform by using Maven Surefire.
- `junit-platform-gradle-plugin`, which provides support for discovering and executing tests on the JUnit Platform by using Gradle.

3.3.3 JUnit 5 Jupiter

JUnit Jupiter is the combination of the new programming (annotations, classes, and methods) and extension model for writing tests and extensions in JUnit 5. The Jupiter subproject provides a `TestEngine` for running Jupiter-based tests on the platform. By contrast with the

existing runner and rule extension points in JUnit 4, the JUnit Jupiter extension model consists of a single coherent concept: the Extension API, discussed in chapter 4.

The artifacts contained in JUnit Jupiter are

- `junit-jupiter-api`, the JUnit Jupiter API for writing tests and extensions
- `junit-jupiter-engine`, the JUnit Jupiter test engine implementation, required only at run time
- `junit-jupiter-params`, which provides support for parameterized tests in JUnit Jupiter
- `junit-jupiter-migrationsupport`, which provides migration support from JUnit 4 to JUnit Jupiter and is required only for running selected JUnit 4 rules

3.3.4 JUnit 5 Vintage

JUnit Vintage provides a `TestEngine` for running JUnit 3- and JUnit 4-based tests on the platform. JUnit 5 Vintage contains only `junit-vintage-engine`, the engine implementation to execute tests written in JUnit 3 or 4. For this purpose, of course, we also need the JUnit 3 or 4 JARs.

This engine is very useful for interacting with the old tests through JUnit 5. We may need to work on our projects with JUnit 5 but still support many old tests. JUnit 5 Vintage is the solution in this situation

3.3.5 The big picture of the JUnit 5 architecture

To explain how the full architecture works, I'll say that the JUnit Platform provides the facilities for running different kinds of tests: JUnit 3, 4, and 5 tests, as well as third-party tests (figure 3.8).

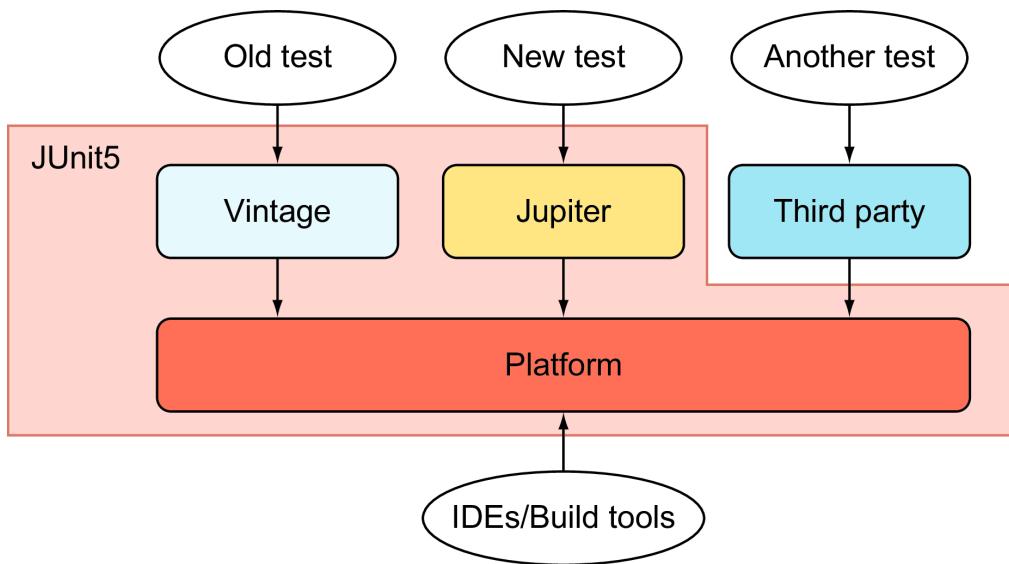


Figure 3.8 The big picture of the JUnit 5 architecture

Detailing at the level of the jar files (figure 3.9):

- The test APIs provide the facilities for different test engines: `junit-jupiter-api` for JUnit 5 tests, `junit-4.12` for legacy tests, and custom engines for third-party tests.
- The test engines mentioned earlier are created by extending the `junit-platform-engine` public API, which is part of the JUnit 5 Platform.
- The `junit-platform-launcher` public API provides the facilities to discover tests inside the JUnit 5 Platform for build tools such as Maven or Gradle or for IDEs.

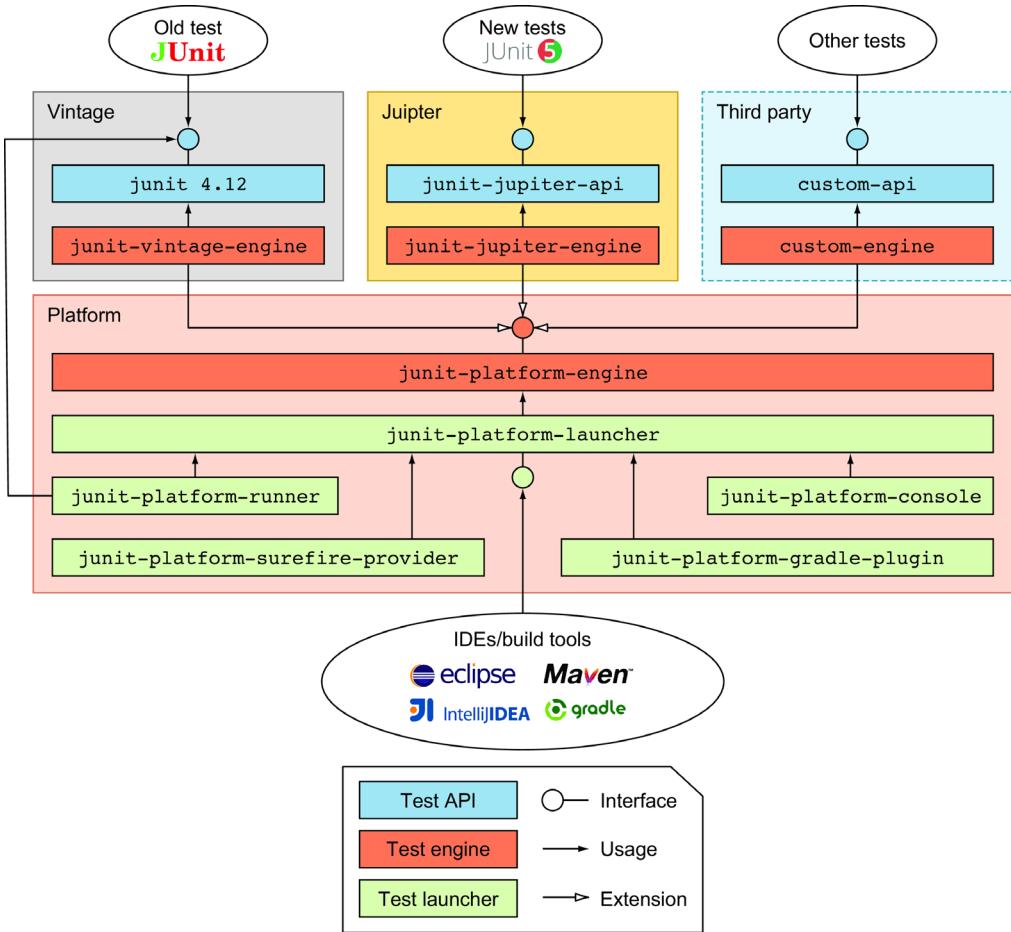


Figure 3.9 The detailed picture of the JUnit 5 architecture

Besides modular architecture, JUnit 5 provides the extensions mechanism, which is discussed in chapter 4.

The architecture of a system strongly determines its capabilities and its behavior. Consider the runners and rules analyzed earlier in this chapter. Their way of working is generated by the JUnit 4 architecture that represents their foundation. Understanding the architectures of both JUnit 4 and JUnit 5 helps us apply their capabilities in practice, write efficient tests, and analyze the alternative implementations, thereby quickening the pace at which we master unit testing.

In chapter 4, we analyze the process of migrating projects from JUnit 4 to JUnit 5 and the dependencies that are required.

3.4 Summary

This chapter has covered the following:

- The idea of software architecture, as the fundamental structure of a software system. A software system structure comprises software elements, relationships among them, and properties of both elements and relationships.
- The analysis of the JUnit 4 monolithic architecture. It is simple, but the real-life challenges, as the interaction with the IDEs and build tools have emphasized its shortcomings.
- The usage of JUnit 4 runners and JUnit 4 rules, which represent possibilities to extend the JUnit 4 monolithic architecture. They are and will still be in use, because there is a large amount of already written tests and because their migration to the JUnit 5 extensions is not straightforward.
- Contrasting the JUnit 4 architectures with the JUnit 5 one, which is modular and it contains the following components: JUnit 5 Platform, JUnit 5 Jupiter and JUnit 5 Vintage.
- Details concerning the big picture of the JUnit 5 architecture and the interaction between the components.
- The JUnit Platform, which serves as a foundation for launching testing frameworks on the JVM and provides an API to launch tests from either the console, IDEs, or build tools.
- JUnit Jupiter, which represents the combination of the new programming model and extension model for writing tests and extensions in JUnit 5.
- JUnit Vintage, the test engine for running JUnit 3 and JUnit 4 based tests on the platform, ensuring the necessary backward compatibility.

4

Migrating from JUnit 4 to JUnit 5

This chapter covers

- Implementing the migration from JUnit 4 to JUnit 5
- Working with a hybrid approach for mature projects
- Comparing the needed JUnit 4 and JUnit 5 dependencies
- Comparing the equivalent JUnit 4 and JUnit 5 annotations
- Comparing the JUnit 4 rules and JUnit 5 extensions

Nothing in this world can survive and remain useful without an update.

—Charles M. Tadros

So far, this book has introduced JUnit and its latest version, 5, and explained the big architectural step between JUnit 4 and JUnit 5. I discussed the core classes and methods and presented them in action so that you have a good understanding of how to build your tests in an efficient way. I emphasized the importance of software architecture in general and shown the big architectural change between JUnit 4 and JUnit 5.

This chapter demonstrates the effective step from JUnit 4 to JUnit 5 inside a project managed by our example company, Tested Data Systems Inc. The company keeps its customer information in a repository and addresses this repository to get the information. In addition, the company needs to track payments and other business rules.

JUnit 4 and JUnit 5 may work together within the same application. This fact is of particular benefit for implementing migration in phases and not necessarily at once.

4.1 The steps between JUnit 4 and JUnit 5

JUnit 5 is a new paradigm introducing a new architecture. Also, it introduces new packages, annotations, methods, and classes. Some JUnit 5 features are similar to JUnit 4 features; others are new, providing new capabilities. The JUnit Jupiter programming and extension model does not support JUnit 4 features such as rules and runners natively. Programmers may not need to update all existing tests, test extensions, and custom build test infrastructure to migrate their projects to JUnit Jupiter—at least, not instantly.

JUnit provides a migration path with the help of the JUnit Vintage test engine. This one offers the possibility that tests based on old JUnit versions are executed with the JUnit Platform infrastructure. All classes and annotations specific to JUnit Jupiter are located in the new `org.junit.jupiter` base package. All classes and annotations specific to JUnit 4 are located in the old `org.junit` base package. So, if both JUnit 4 and JUnit 5 Jupiter are in the classpath does not result in any conflict. Consequently, our projects may keep previously implemented JUnit 4 tests together with the JUnit Jupiter tests. JUnit 5 and JUnit 4 may coexist until we finalize our migration, whenever that may be, and this migration may be planned and executed slowly, according to the priority of the tasks and the challenges of various steps.

Before developing and running the JUnit tests, programmers must have the following programs:

- JUnit 4 requires Java 5 or later.
- JUnit 5 requires Java 8 or later.

Consequently, migrating from JUnit 4 to JUnit 5 may require an update of the Java version in use inside the project.

Table 4.1 summarizes the most important steps in migrating from JUnit 4 to JUnit 5.

Table 4.1 Migrating from JUnit 4 and JUnit 5

Main step	Comments
Replace the needed dependencies	JUnit 4 needs a single dependency. JUnit 5 requires more dependencies, related to the features that are used. JUnit 5 uses JUnit Vintage to work with old JUnit 4 tests.
Replace the annotations, and introduce the new ones	Some JUnit 5 annotations mirror the old JUnit 4 ones. Some new ones introduce new facilities and help developers write better tests.
Replace the testing classes and methods	JUnit 5 assertions and assumptions have been moved to different classes from different packages.
Replace the JUnit 4 rules and runners with the JUnit 5 extension model	This step generally requires more effort than the other steps in this table. Because JUnit 4 and JUnit 5 may

	coexist for a long period, however, the rules and runners may remain in the code or be replaced much later.
--	---

4.2 Needed dependencies

This section discusses the migration process from JUnit 4 to JUnit 5 in Tested Data Systems. The company has decided to migrate because it needs to create more testing code for its products and needs more flexibility and more clarity for writing these tests. JUnit 5 provides the capabilities of labeling the tests with display names, by its nested tests and dynamic tests. Before it can work effectively with these new capabilities of JUnit 5, Tested Data Systems needs to take the first step of creating JUnit 5 tests for its projects.

As we have explained, JUnit 4 has a monolithic architecture, so there is one single dependency in the Maven configuration that supports running JUnit 4 tests. This one is shown in listing 4.1.

Listing 4.1 The JUnit 4 Maven dependency

```
<dependencies>
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.12</version>
        <scope>test</scope>
    </dependency>
</dependencies>
```

One JUnit 5 dependency, JUnit Vintage, may replace the dependency from listing 4.1 in case of migration. The first things that Tested Data Systems needs to do in the migration process are at the level of the dependencies that are used.

The first dependency is `junit-vintage-engine` (listing 4.2). It belongs to JUnit 5 but ensures backward compatibility with previous versions of JUnit. Working with JUnit 5 Vintage, Maven transitively accesses the dependency to JUnit 4. Introducing this dependency is a first step in the migration 5 that Tested Data Systems decided to implement for its projects. JUnit 4 and JUnit 5 tests may coexist within the same project until the migration process is finalized.

Listing 4.2 The JUnit Vintage Maven dependency

```
<dependencies>
    <dependency>
        <groupId>org.junit.vintage</groupId>
        <artifactId>junit-vintage-engine</artifactId>
        <version>5.6.0</version>
        <scope>test</scope>
    </dependency>
</dependencies>
```

Running the JUnit 4 tests right now, we may notice that they are successfully executed (figure 4.1). Working with the JUnit 5 Vintage dependency instead of the old JUnit 4 one will not make any difference.

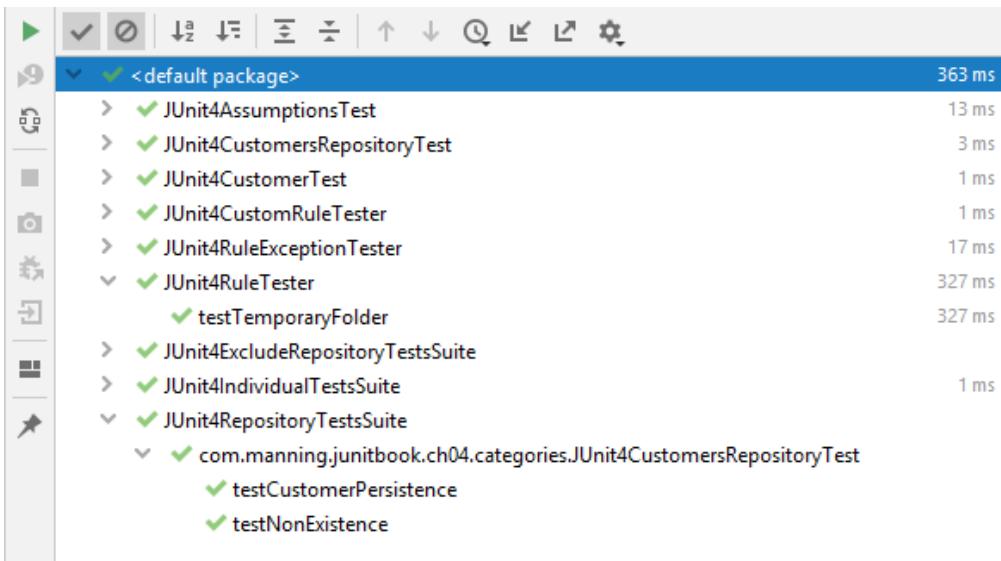


Figure 4.1 Running the JUnit 4 tests after replacing the old JUnit 4 dependency with JUnit 5 Vintage

After the company's programmers introduce the JUnit Vintage dependency, the migration path of Tested Data Systems's projects may continue with the introduction of JUnit 5 Jupiter annotations and features. The required dependencies are shown in listing 4.3.

Listing 4.3 The most useful JUnit Jupiter Maven dependencies

```
<dependencies>
    <dependency>
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter-api</artifactId>
        <version>5.6.0</version>
        <scope>test</scope>
    </dependency>
    <dependency>
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter-engine</artifactId>
        <version>5.6.0</version>
        <scope>test</scope>
    </dependency>
</dependencies>
```

To write tests using JUnit 5, you will always need the `junit-jupiter-api` and the `junit-jupiter-engine` dependencies. The first one represents the API for writing tests with JUnit Jupiter (including the annotations, classes, and methods to be migrated to). The second one represents the core JUnit Jupiter package for the execution test engine.

An additional dependency that we may need is `junit-jupiter-params` (for running parameterized tests). At the end of the migration process (when no more JUnit 4 tests are

left), we may remove the first junit-vintage-engine dependency, presented in listing 4.2.

4.3 Annotations, classes, and methods

After making the movement at the level of project dependencies, Tested Data Systems continues implementing the migration process. As explained earlier in this book, JUnit 5 provides similar features to JUnit 4 as well as a lot of new ones. Tested Data Systems hopes to benefit from the flexibility and clarity of these new features for the many tests that need to be developed for its projects. First, however, the company needs to migrate the equivalent features.

Tables 4.2, 4.3, and 4.4 summarize the equivalent annotations, classes, and methods between JUnit 4 and JUnit 5.

Table 4.2 Annotations

JUnit 4	JUnit 5
@BeforeClass, @AfterClass	@BeforeAll, @AfterAll
@Before, @After	@BeforeEach, @AfterEach
@Ignore	@Disable
@Category	@Tag

Table 4.3 Assertions

JUnit 4	JUnit 5
Assert class	<i>Assertions</i> class
Optional assertion message is the first parameter	Optional assertion message is the last parameter.
assertThat method	assertThat method removed. New methods are assertAll and assertThrows.

Table 4.4 Assumptions

JUnit 4	JUnit 5
Assume class	Assumptions class
assumeNotNull and assumeNoException	assumeNotNull and assumeNoException are removed.

Now, the engineers of Tested Data Systems decide to take the next step by applying the needed changes to the code from their projects.

This section uses tests similar to the Tested Data Systems examples, which are built with both JUnit 4 and JUnit 5, to clarify the changes that have been introduced in JUnit 5.

We start with a class that simulates a system under test (SUT). This one can be initialized, can receive usual but cannot receive additional work to execute, and may close itself. Listing 4.4 presents the SUT class.

Listing 4.4 The tested SUT class

```
public class SUT {  
    private String systemName;  
  
    public SUT(String systemName) {  
        this.systemName = systemName;  
        System.out.println(systemName + " from class " +  
                           getClass().getSimpleName() + " is initializing.");  
    }  
  
    public boolean canReceiveUsualWork() {  
        System.out.println(systemName + " from class " +  
                           getClass().getSimpleName() + " can receive usual work.");  
        return true;  
    }  
  
    public boolean canReceiveAdditionalWork() {  
        System.out.println(systemName + " from class " +  
                           getClass().getSimpleName() + " cannot receive additional work.");  
        return false;  
    }  
  
    public void close() {  
        System.out.println(systemName + " from class " +  
                           getClass().getSimpleName() + " is closing.");  
    }  
}
```

Listing 4.5 verifies the functionality of the SUT by using the JUnit 4 facilities, and listing 4.6 verifies the functionality of the SUT by using the JUnit 5 facilities. These examples also demonstrate the life cycle methods. As previously stated, the system may start up, receive usual and additional work, and close itself. The JUnit 4 and JUnit 5 life cycle and testing methods ensure that the system is initializing and then shutting down before and after each effective test. The test methods are checking whether the system may receive usual work and additional work.

Listing 4.5 The JUnit4SUTTest class

```
public class JUnit4SUTTest {  
  
    private static ResourceForAllTests resourceForAllTests;  
    private SUT systemUnderTest;  
  
    @BeforeClass  
    #A
```

```

public static void setUpClass() {
    resourceForAllTests =
        new ResourceForAllTests("Our resource for all tests");
}

@AfterClass                                     #B
public static void tearDownClass() {
    resourceForAllTests.close();
}

@Before                                         #C
public void setUp() {
    systemUnderTest = new SUT("Our system under test");
}

@After                                           #D
public void tearDown() {
    systemUnderTest.close();
}

@Test                                            #E
public void testUsualWork() {
    boolean canReceiveUsualWork = systemUnderTest.canReceiveUsualWork();

    assertTrue(canReceiveUsualWork);
}

@Test                                            #E
public void testAdditionalWork() {
    boolean canReceiveAdditionalWork =
        systemUnderTest.canReceiveAdditionalWork();
    assertFalse(canReceiveAdditionalWork);
}

@Test                                             #E
@Ignore                                         #F
public void mySecondTest() {
    assertEquals( "2 is not equal to 1", 2, 1 );
}
}

```

We previously replaced the JUnit 4 dependency with the JUnit Vintage one. The result of running the JUnit4SUTTest class is the same in both cases (figure 4.2), mySecondTest being marked with the `@Ignore` annotation. Now we can proceed to the effective migration of annotations, classes, and methods.

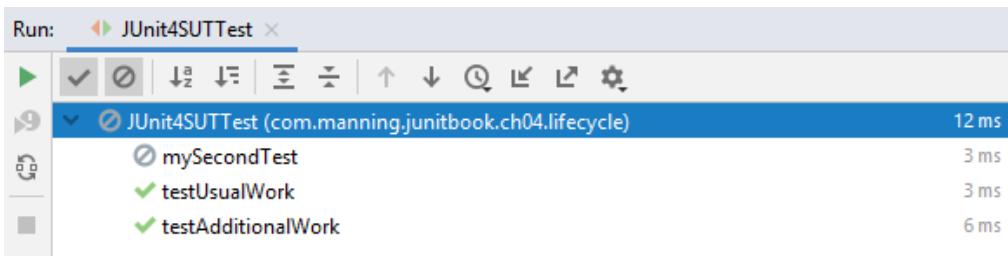


Figure 4.2 Running the JUnit4SUTTestSuite in IntelliJ, using both the JUnit 4 dependency and the JUnit Vintage one

Listing 4.6 The JUnit5SUTTest class

```
class JUnit5SUTTest {
    private static ResourceForAllTests resourceForAllTests;
    private SUT systemUnderTest;

    @BeforeAll                                         #A'
    static void setUpClass() {
        resourceForAllTests =
            new ResourceForAllTests("Our resource for all tests");
    }

    @AfterAll                                         #B'
    static void tearDownClass() {
        resourceForAllTests.close();
    }

    @BeforeEach                                         #C'
    void setUp() {
        systemUnderTest = new SUT("Our system under test");
    }

    @AfterEach                                         #D'
    void tearDown() {
        systemUnderTest.close();
    }

    @Test                                              #E'
    void testUsualWork() {
        boolean canReceiveUsualWork =
            systemUnderTest.canReceiveUsualWork();

        assertTrue(canReceiveUsualWork);
    }

    @Test                                              #E'
    void testAdditionalWork() {
        boolean canReceiveAdditionalWork =
            systemUnderTest.canReceiveAdditionalWork();

        assertFalse(canReceiveAdditionalWork);
    }
}
```

```

@Test
@Disabled
void mySecondTest() {
    assertEquals(2, 1, "2 is not equal to 1");
}

```

Comparing the JUnit 4 and JUnit 5 methods, we see that

- The methods annotated with `@BeforeClass` (#A in listing 4.5) and `@BeforeAll` (#A' in listing 4.6), respectively, are executed once, before all tests. These methods need to be static. In the JUnit 4 version, the method also needs to be public. In the JUnit 5 version, we can make the method nonstatic and annotate the whole test class with `@TestInstance(Life_cycle.PER_CLASS)`.
- The methods annotated with `@AfterClass` (#B in listing 4.5) and `@AfterAll` (#B' in listing 4.6), respectively, are executed once, after all tests. These methods need to be static. In the JUnit 4 version, the method also needs to be public. In the JUnit 5 version, we can make the method nonstatic and annotate the whole test class with `@TestInstance(Life_cycle.PER_CLASS)`.
- The methods annotated with `@Before` (#C in listing 4.5) and `@BeforeEach` (#C' in listing 4.6), respectively, are executed before each test. In the JUnit 4 version, the methods need to be public.
- The methods annotated with `@After` (#D in listing 4.5) and `@AfterEach` (#D' in listing 4.6), respectively, are executed after each test. In the JUnit 4 version, the methods need to be public.
- The methods annotated with `@Test` (#E in listing 4.5) and `@Test` (#E' in listing 4.6) are executed independently. In the JUnit 4 version, the methods need to be public. The two annotations belong to different packages: `org.junit.Test` and `org.junit.jupiter.api.Test`, respectively.
- To skip the execution of a test method, JUnit 4 uses the annotation `@Ignore` (#F in listing 4.5), whereas JUnit 5 uses the annotation `@Disabled` (#F' in listing 4.6).

The access level has been relaxed for the test methods, from public to package-private. These methods are accessed only from within the package to which the test class belongs to, so they didn't need to be made public.

The engineers of Tested Data Systems applied the equivalences presented in tables 4.2, 4.3, and 4.4 to the migration of their code.

Tested Data Systems needs to verify its customers' information, as well as their existence or nonexistence. It wants to classify the verification tests in two groups: the ones that work with individual customers and the ones that check inside a repository. The company has used categories (in JUnit 4) and needs to switch to tags (in JUnit 5).

Listing 4.7 shows the two interfaces that need to be declared for the definition of JUnit 4 categories. These interfaces will be used as arguments of the JUnit 4 `@Category` annotation.

Listing 4.7 The interfaces created to define categories with JUnit 4

```
public interface IndividualTests {  
}  
  
public interface RepositoryTests {  
}
```

Listing 4.8 defines a JUnit 4 test that contains a method annotated as `@Category(IndividualTests.class)`. This annotation assigns that test method as belonging to this category.

Listing 4.8 JUnit4CustomerTest class with one test method annotated with @Category

```
public class JUnit4CustomerTest {  
    private String CUSTOMER_NAME = "John Smith";  
  
    @Category(IndividualTests.class)  
    @Test  
    public void testCustomer() {  
        Customer customer = new Customer(CUSTOMER_NAME);  
  
        assertEquals("John Smith", customer.getName());  
    }  
}
```

Listing 4.9 defines a JUnit 4 test class annotated as `@Category(IndividualTests.class, RepositoryTests.class)`. This annotation assigns the two containing test methods as belonging to these two categories.

Listing 4.9 JUnit4CustomersRepositoryTest class annotated with @Category

```
@Category({IndividualTests.class, RepositoryTests.class})  
public class JUnit4CustomersRepositoryTest {  
    private String CUSTOMER_NAME = "John Smith";  
    private CustomersRepository repository = new CustomersRepository();  
  
    @Test  
    public void testNonExistence() {  
        boolean exists = repository.contains(CUSTOMER_NAME);  
  
        assertFalse(exists);  
    }  
  
    @Test  
    public void testCustomerPersistence() {  
        repository.persist(new Customer(CUSTOMER_NAME));  
  
        assertTrue(repository.contains("John Smith"));  
    }  
}
```

Listings 4.10, 4.11, and 4.12 describe three suites that look for particular test categories in the given classes.

Listing 4.10 The JUnit4IndividualTestsSuite class

```
@RunWith(Categories.class) #A
@Categories.IncludeCategory(IndividualTests.class) #B
@Suite.SuiteClasses({JUnit4CustomerTest.class, JUnit4CustomersRepositoryTest.class}) #C
public class JUnit4IndividualTestsSuite {
}
```

In this example, the `JUnit4IndividualTestsSuite`:

- Is annotated with `@RunWith(Categories.class)` (#A), informing JUnit that it has to execute the tests with this particular runner.
- Includes the category of tests annotated with `IndividualTests` (#B).
- Looks for these annotated tests in the `JUnit4CustomerTest` and `JUnit4CustomersRepositoryTest` classes (#C).

The result of running this suite is shown in figure 4.3. All tests from the `JUnit4CustomerTest` and `JUnit4CustomersRepositoryTest` classes will be executed, as all of them are annotated with `IndividualTests`.

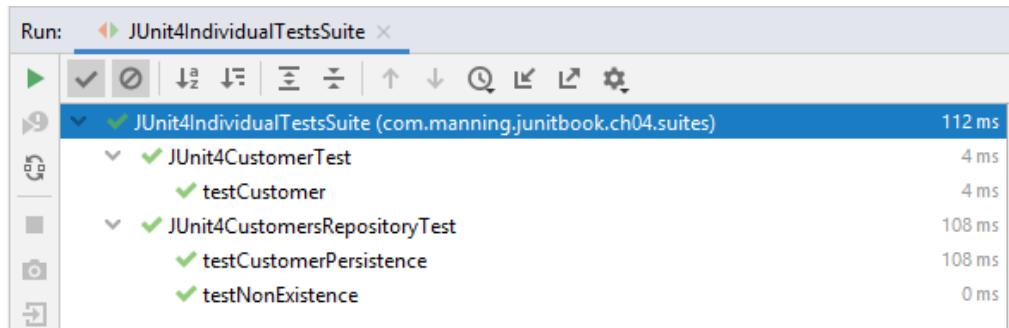


Figure 4.3 Running the `JUnit4IndividualTestsSuite` in IntelliJ

Listing 4.11 The JUnit4RepositoryTestsSuite class

```
@RunWith(Categories.class) #A
@Categories.IncludeCategory(RepositoryTests.class) #B
@Suite.SuiteClasses({JUnit4CustomerTest.class, JUnit4CustomersRepositoryTest.class}) #C
public class JUnit4RepositoryTestsSuite {
}
```

In this example, the `JUnit4RepositoryTestsSuite`

- Is annotated with `@RunWith(Categories.class)` (#A), informing JUnit that it has to execute the tests with this particular runner.

- Includes the category of tests annotated with `RepositoryTests` (#B).
- Looks for these annotated tests in the `JUnit4CustomerTest` and `JUnit4CustomersRepositoryTest` classes (#C).

The result of running this suite is shown in figure 4.4. Two tests from the `JUnit4CustomersRepositoryTest` class will be executed, as they are annotated with `RepositoryTests`.

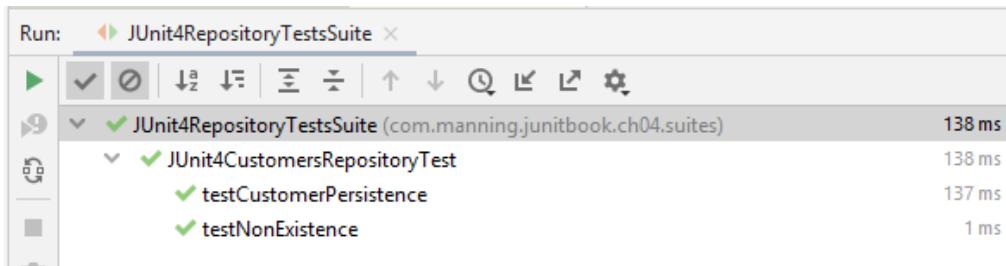


Figure 4.4 Running the `JUnit4RepositoryTestsSuite` in IntelliJ

Listing 4.12 The `JUnit4ExcludeRepositoryTestsSuite` class

```
@RunWith(Categories.class) #A
@Categories.ExcludeCategory(RepositoryTests.class) #B
@Suite.SuiteClasses({JUnit4CustomerTest.class, JUnit4CustomersRepositoryTest.class}) #C
public class JUnit4ExcludeRepositoryTestsSuite {
}
```

In this example, the `JUnit4ExcludeRepositoryTestsSuite`

- Is annotated with `@RunWith(Categories.class)` (#A), informing JUnit that it has to execute the tests with this particular runner.
- Excludes the category of tests annotated with `RepositoryTests` (#B).
- Looks for these annotated tests in the `JUnit4CustomerTest` and `JUnit4CustomersRepositoryTest` classes (#C).

The result of running this suite is shown in figure 4.5. One test from the `JUnit4CustomerTest` class will be executed, as it is not annotated with `RepositoryTests`.

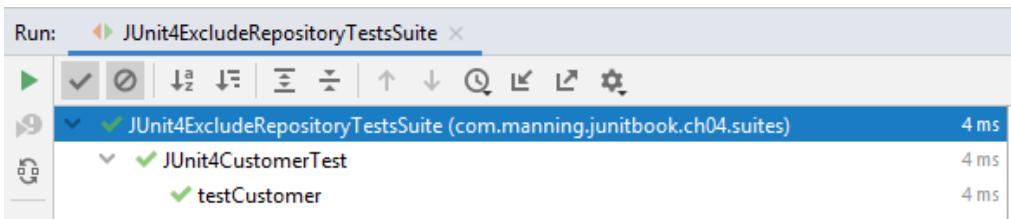


Figure 4.5 Running the JUnit4ExcludeRepositoryTestsSuite in IntelliJ

The JUnit 4 categories approach works, but it also has some shortcomings: it is a lot of code to write, and the team needs to define special test suites and special interfaces to be used solely as marker interfaces for the tests. For that reason, the engineers at Tested Data Systems decide to switch to the JUnit 5 tags approach.

Listing 4.13 introduces the `JUnit5CustomerTest` tagged class, and listing 4.14 shows the `CustomerRepositoryTest` tagged class.

Listing 4.13 The JUnit5CustomerTest tagged class

```
@Tag("individual") #A
public class JUnit5CustomerTest {
    private String CUSTOMER_NAME = "John Smith";

    @Test
    void testCustomer() {
        Customer customer = new Customer(CUSTOMER_NAME);

        assertEquals("John Smith", customer.getName());
    }
}
```

The `@Tag` annotation is added to the whole `JUnit5CustomerTest` class (#A).

Listing 4.14 The JUnit5CustomerRepositoryTest tagged class

```
@Tag("repository") #A
public class JUnit5CustomersRepositoryTest {
    private String CUSTOMER_NAME = "John Smith";
    private CustomersRepository repository = new CustomersRepository();

    @Test
    void testNonExistence() {
        boolean exists = repository.contains("John Smith");

        assertFalse(exists);
    }

    @Test
    void testCustomerPersistence() {
        repository.persist(new Customer(CUSTOMER_NAME));

        assertTrue(repository.contains("John Smith"));
    }
}
```

```
}
```

Similarly, the @Tag annotation is added to the whole JUnit5CustomerRepositoryTest class (#A).

To activate the JUnit 5 tags that replace the JUnit 4 categories, we have a few alternatives. For one, we can work at the level of the pom.xml configuration file. In listing 4.15, we can uncomment the configuration node of the Surefire plugin (#A) and run mvn clean install.

Listing 4.15 The pom.xml configuration file

```
<plugin>
  <artifactId>maven-surefire-plugin</artifactId>
  <version>2.22.2</version>
  <!-- #A
  <configuration>
    <groups>individual</groups> #A
    <excludedGroups>repository</excludedGroups> #A
  </configuration> #A
  --> #A
</plugin>
```

In order to activate the usage of the JUnit 5 tags that replace the JUnit 4 categories, you have a few alternatives. You can work at the level of the pom.xml configuration file. In listing 4.15, it will be enough to uncomment the configuration node of the surefire plugin #A and run mvn clean install.

Or, from the IntelliJ IDEA IDE, you can activate the usage of the tags by going to Run -> Edit Configurations and choose Tags (JUnit 5) as test kind (figure 4.6). However, the recommended way to go is to change the pom.xml, so that the tests can be correctly executed from the command line.

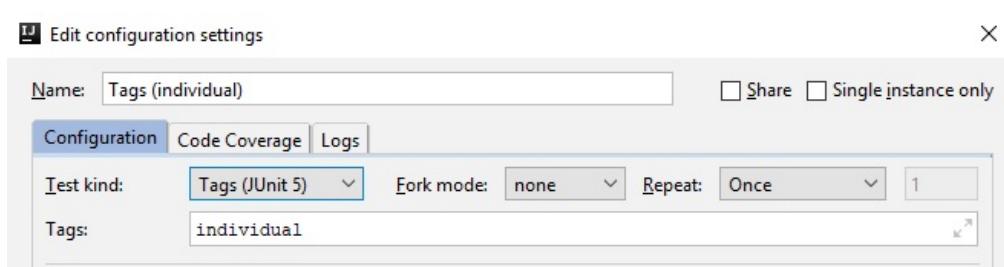


Figure 4.6 Configuring the tagged tests from the IntelliJ IDEA IDE

Notice that we no longer define special interfaces for this goal; we no longer create many tedious suites at the level of the code. We simply annotate the classes with our own tags and then choose what to run by making the selections through the Maven configuration file or the IDE. We have less code to write and fewer changes to make at the level of the code itself. The

decision of the engineers at Tested Data Systems to take advantage of the JUnit 5 facilities pays benefits.

To continue the comparison of JUnit 4 and JUnit 5, we look at the Hamcrest matchers functionality, introduced in chapter 2. In this chapter, we use our collections example to put the two versions face to face. We will populate a list with values from Tested Data Systems's internal information and then investigate whether its elements match some patterns, using JUnit 4 (listing 4.16) and JUnit 5 (listing 4.17).

Listing 4.16 The JUnit4HamcrestListTest class

```
import org.junit.Before;
import org.junit.Test;

public class JUnit4HamcrestListTest {

    private List<String> values;

    @Before
    public void setUp() {                                         #A
        values = new ArrayList<>();
        values.add("John");
        values.add("Michael");
        values.add("Edwin");
    }

    @Test
    public void testListWithHamcrest() {                           #B
        assertThat(values, hasSize(3));                          #C
        assertThat(values, hasItem(anyOf(equalTo("Oliver"), equalTo("Jack"),
            equalTo("Harry"))));                                #D
        assertThat("The list doesn't contain all the expected objects, in
            order", values, contains("Oliver", "Jack", "Harry")); #E
        assertThat("The list doesn't contain all the expected objects",
            values, containsInAnyOrder("Jack", "Harry", "Oliver")); #F
    }
}
```

Listing 4.17 The JUnit5HamcrestListTest class

```
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Test;

public class JUnit5HamcrestListTest {

    private List<String> values;

    @BeforeEach
    public void setUp() {                                         #A'
        values = new ArrayList<>();
        values.add("John");
        values.add("Michael");
        values.add("Edwin");
    }
}
```

```

@Test
@DisplayName("List with Hamcrest")
public void testListWithHamcrest() {
    assertThat(values, hasSize(3)); #B'
    assertThat(values, hasItem(anyOf(equalTo("Oliver"), equalTo("Jack"),
        equalTo("Harry")))); #C'
    assertThat("The list doesn't contain all the expected objects, in #E'
        order", values, contains("Oliver", "Jack", "Harry"))); #D'
    assertThat("The list doesn't contain all the expected objects", #F'
        values, containsInAnyOrder("Jack", "Harry", "Oliver"));
}
}

```

These examples are very similar (except for the `@Before`/`@BeforeEach` and `@DisplayName` annotations). The old imports are gradually replaced by the new ones, which is why the annotations `org.junit.Test` and `org.junit.jupiter.api.Test` belong to different packages.

What do these programs do with the internal information managed at Tested Data Systems?

- We initialize the list to work with. The code is the same, but the annotations are different: `@Before (#A)` and `@BeforeEach (#A')`.
- The test methods are annotated with `org.junit.Test (#B)` and `org.junit.jupiter.api.Test (#B')`, respectively.
- Verification (#C) uses the `org.junit.Assert.assertThat` method. JUnit 5 removes this method, so we use `org.hamcrest.MatcherAssert.assertThat (#C')`.
- We use the `anyOf` and `equalTo` methods from the `org.hamcrest.Matchers` class (#D) and the `anyOf` and `equalTo` methods from the `org.hamcrest.CoreMatchers` class (#D').
- We use the same `org.hamcrest.Matchers.contains` method (#E and #E').
- We use the same `org.hamcrest.Matchers.containsInAnyOrder` method (#F and #F').

4.3.1 Rules vs. the extension model

A JUnit4 rule is a component that allows introducing additional actions when a method is executed, by intercepting its call and do something before and after the execution of the method.

To compare the rules model of JUnit 4 and the extension model of JUnit 5, we'll revisit our extended `Calculator` class (listing 4.18). The developers at Tested Data Systems use this class to execute mathematical operations for verifying their SUTs. They are interested in testing the methods that may throw exceptions. One rule that the Tested Data Systems test code uses extensively is `ExpectedException`, which can easily be replaced by the JUnit 5 `assertThrows` method.

Listing 4.18 The extended Calculator class

```
public class Calculator {  
    ...  
    public double sqrt(double x) { #A  
        if (x < 0) {  
            throw new  
                IllegalArgumentException("Cannot extract the square #B  
                    root of a negative value"); #B  
        }  
        return Math.sqrt(x); #A  
    }  
  
    public double divide(double x, double y) { #C  
        if (y == 0) {  
            throw new ArithmeticException("Cannot divide by zero"); #D  
        }  
        return x/y; #C  
    }  
}
```

The logic that may throw exceptions into the `Calculator` class does the following:

- Declares a method to calculate the square root of a number (#A). In case the number is negative, an exception containing a particular message is created and thrown (#B).
- Declares a method to divide two numbers (#C). In case the second number is zero, an exception containing a particular message is created and thrown (#D).

Listing 4.19 provides an example that specifies which exception message is expected during the execution of the test code, using the new functionality of the `Calculator` class (refer to listing 4.18).

Listing 4.19 The JUnit4RuleExceptionTester class

```
public class JUnit4RuleExceptionTester {  
    @Rule  
    public ExpectedException expectedException = #A  
        ExpectedException.none(); #A  
  
    private Calculator calculator = new Calculator(); #B  
  
    @Test  
    public void expectIllegalArgumentException() {  
        expectedException.expect(IllegalArgumentException.class); #C  
        expectedException.expectMessage("Cannot extract the square root #D  
            of a negative value"); #D  
        calculator.sqrt(-1); #E  
    }  
  
    @Test  
    public void expectArithmeticException() {  
        expectedException.expect(ArithmeticException.class); #F  
        expectedException.expectMessage("Cannot divide by zero"); #G  
        calculator.divide(1, 0); #H  
    }  
}
```

```
    }  
}
```

In this example:

- We declare an `ExpectedException` field annotated with `@Rule`. The `@Rule` annotation must be applied on a public nonstatic field or a public nonstatic method (#A). The `ExpectedException.none()` factory method simply creates an unconfigured `ExpectedException`.
- We initialize an instance of the `Calculator` class whose functionality we are testing (#B).
- The `ExpectedException` is configured to keep the type of exception (#C) and the message (#D) before being thrown by invoking the `sqrt` method (#E).
- The `ExpectedException` is configured to keep the type of exception (#F) and the message (#G) before being thrown by invoking the `divide` method (#H).

Listing 4.20 shows the JUnit 5 approach.

Listing 4.20 The JUnit5ExceptionTester class

```
public class JUnit5ExceptionTester {  
    private Calculator calculator = new Calculator(); #A'  
  
    @Test  
    public void expectIllegalArgumentException() {  
        Throwable throwable = assertThrows( #B'  
            IllegalArgumentException.class, #B'  
            () -> calculator.sqrt(-1)); #B'  
        assertEquals("Cannot extract the square root of a #C'  
            negative value", throwable.getMessage()); #C'  
    }  
  
    @Test  
    public void expectArithmeticeException() {  
        Throwable throwable = assertThrows(ArithmeticeException.class, #D'  
            () -> calculator.divide(1, 0)); #D'  
        assertEquals("Cannot divide by zero", throwable.getMessage()); #E'  
    }  
}
```

In this example:

- We initialize an instance of the `Calculator` class whose functionality we are testing (#A').
- We assert that the execution of the supplied `calculator.sqrt(-1)` executable throws an `IllegalArgumentException` (#B') and we check the message from the exception (#C').
- We assert that the execution of the supplied `calculator.divide(1, 0)` executable throws an `ArithmeticeException` (#D') and we check the message from the

exception (#E').

A clear difference exists in code clarity and length between JUnit 4 and JUnit 5. The effective testing JUnit 5 code is 13 lines, whereas the effective JUnit 4 code is 20 lines. We do not need to initialize and manage any additional rule. The testing JUnit 5 methods contain one line each.

Another rule that Tested Data Systems would like to migrate is `TemporaryFolder`. The `TemporaryFolder` rule allows the creation of files and folders that should be deleted when the test method finishes (whether it passes or fails). As the tests of the Tested Data Systems projects work intensively with temporary resources, this step is also a must. The JUnit 4 rule has been replaced by the `@TempDir` annotation in JUnit 5. Listing 4.21 presents the JUnit 4 approach.

Listing 4.21 The JUnit4RuleTester class

```
public class JUnit4RuleTester {  
    @Rule  
    public TemporaryFolder folder = new TemporaryFolder(); #A  
  
    @Test  
    public void testTemporaryFolder() throws IOException {  
        File createdFolder = folder.newFolder("createdFolder"); #B  
        File createdFile = folder.newFile("createdFile.txt"); #B  
        assertTrue(createdFolder.exists()); #C  
        assertTrue(createdFile.exists()); #C  
    }  
}
```

In this example:

- We declare a `TemporaryFolder` field annotated with `@Rule` and initialize it. The `@Rule` annotation must be applied on a public field or a public method (#A).
- We use the `TemporaryFolder` field to create a folder and a file (#B), which located in the `Temp` folder of our user profile in the operating system.
- We check the existence of the temporary folder and of the temporary file (#C).

Listing 4.22 shows the new JUnit 5 approach.

Listing 4.22 The JUnit5TempDirTester class

```
public class JUnit5TempDirTester {  
    @TempDir  
    Path tempDir; #A,  
    #A,  
  
    private static Path createdFile; #B'  
  
    @Test  
    public void testTemporaryFolder() throws IOException {  
        assertTrue(Files.isDirectory(tempDir)); #C,  
        Path createdFile = Files.createFile( #D,  
            tempDir.resolve("createdFile.txt") #D,  
        ); #D,
```

```

        assertTrue(createdFile.toFile().exists());                      #D'
    }

@AfterAll
public static void afterAll() {
    assertFalse(createdFile.toFile().exists());                         #E'
}
}

```

In this example:

- We declare a `@TempDir` annotated field (#A').
- We declare the `createdFile` variable (#B').
- We check the creation of this temporary directory before the execution of the test (#C').
- We create a file within this directory and check its existence (#D').
- After the execution of the tests, we check that the temporary resource has been removed (#E'). The temporary folder will be removed after the execution of the `afterAll` method ends.

The advantage of the JUnit 5 extension approach is that we do not have to create the folder by ourselves through a constructor; the folder is created automatically when we annotate a field with `@TempDir`.

4.3.2 Custom rules

Tested Data Systems has defined some custom rules for its tests. Custom rules are particularly useful when some tests need similar additional actions before and after execution.

In JUnit 4, the Tested Data Systems engineers needed their additional actions to be executed before and after the execution of a test. Consequently, they created their own classes that implement the `TestRule` interface. To do this, they had to override the `apply(Statement, Description)` method, which returns an instance of `Statement`. Such an object represents the tests within the JUnit run time, and `Statement#evaluate()` will run them. The `Description` object describes the individual test. This object can be used to read information about the test through reflection.

Listing 4.23 The CustomRule class

```

public class CustomRule implements TestRule {                                #A
    private Statement base;                                                 #B
    private Description description;                                         #B

    @Override
    public Statement apply(Statement base, Description description) {      #C
        this.base = base;                                                    #C
        this.description = description;                                       #C
        return new CustomStatement(base, description);                        #C
    }
}

```

```
}
```

In this example:

- We declare our `CustomRule` class that implements the `TestRule` interface (#A).
- We keep references to a `Statement` field and to a `Description` field (#B), and we use them in the `apply` method that returns a `CustomStatement` (#C).

Listing 4.24 The CustomStatement class

```
public class CustomStatement extends Statement { #A
    private Statement base; #B
    private Description description; #B

    public CustomStatement(Statement base, Description description) { #C
        this.base = base;
        this.description = description;
    }

    @Override #D
    public void evaluate() throws Throwable {
        System.out.println(this.getClass().getSimpleName() + " " + #D
            description.getMethodName() + " has started"); #D
        try { #D
            base.evaluate(); #D
        } finally { #D
            System.out.println(this.getClass().getSimpleName() + " " + #D
                description.getMethodName() + " has finished"); #D
        }
    }
}
```

In this example:

- We declare our `CustomStatement` class that extends the `Statement` class (#A).
- We keep references to a `Statement` field and to a `Description` field (#B), and we use them as arguments of the constructor (#C).
- We override the inherited `evaluate` method and call `base.evaluate()` inside it (#D).

Listing 4.25 The JUnit4CustomRuleTester class

```
public class JUnit4CustomRuleTester {

    @Rule #A
    public CustomRule myRule = new CustomRule(); #A

    @Test #B
    public void myCustomRuleTest() { #B
        System.out.println("Call of a test method");
    }
}
```

In this example, we use the previously defined `CustomRule` by doing the following:

- We declare a public `CustomRule` field and annotate it with `@Rule (#A)`.
- We create the `myCustomRuleTest` method and annotate it with `@Test (#B)`.

The result of the execution of this test is shown in figure 4.7. As the engineers from Tested Data Systems required, the effective execution of the test is surrounded by the additional messages provided to the `evaluate` method of the `CustomStatement` class.

The screenshot shows a terminal window with the following output:

```
Tests passed: 1 of 1 test – 5 ms
"C:\Program Files\Java\jdk1.8.0_181\bin\java" ...
CustomStatement myCustomRuleTest has started
Call of a test method
CustomStatement myCustomRuleTest has finished

Process finished with exit code 0
```

Figure 4.7 The result of the execution of `JUnit4CustomRuleTester`

The engineers from Tested Data Systems would like to migrate their own rules as well. JUnit 5 allows similar effects, as in the case of the JUnit 4 rules, by introducing custom extensions, that will extend the behavior of test classes and methods. The code will be shorter and will rely on the declarative annotations style. First, the engineers define the `CustomExtension` class, which will be used as an argument of the `@ExtendWith` annotation on the tested class.

Listing 4.26 The CustomExtension class

```
public class CustomExtension implements AfterEachCallback,
BeforeEachCallback {
    @Override
    public void beforeEach(ExtensionContext extensionContext)
        throws Exception {
        System.out.println(this.getClass().getSimpleName() + " " +
            extensionContext.getDisplayName() + " has started");
    }

    @Override
    public void afterEach(ExtensionContext extensionContext)
        throws Exception {
        System.out.println(this.getClass().getSimpleName() + " " +
            extensionContext.getDisplayName() + " has finished");
    }
}
```

In this example:

- We are declaring `CustomExtension` as implementing the `AfterEachCallback` and

BeforeEachCallback interfaces (#A').

- We are overriding the `beforeEach` method, to be executed before each test method from the testing class that will be extended with `CustomExtension` (#B').
- We are overriding the `afterEach` method, to be executed after each test method from the testing class that will be extended with `CustomExtension` (#C').

Listing 4.27 The JUnit5CustomExtensionTester class

```
@ExtendWith(CustomExtension.class) #A'
public class JUnit5CustomExtensionTester {

    @Test #B'
    public void myCustomRuleTest() { #B,
        System.out.println("Call of a test method"); #B'
    } #B'
}
```

In this example:

- We extend `JUnit5CustomExtensionTester` with the `CustomExtension` class (#A').
- We create the `myCustomRuleTest` method and annotate it with `@Test` (#B).

The result of the execution of this test is shown in figure 4.8. As the test class is extended with the `CustomExtension` class, the previously defined `beforeEach` and `afterEach` methods will be executed before and after each test method, respectively.

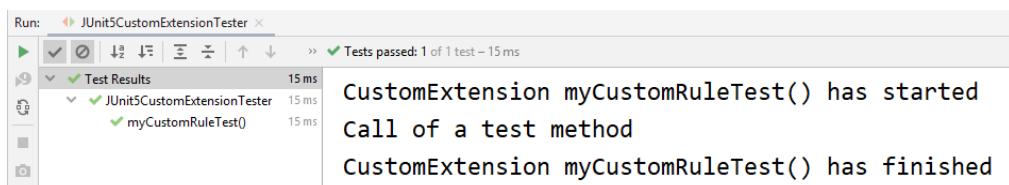


Figure 4.8 The result of the execution of `JUnit5CustomExtensionTester`.

Notice the clear difference in code clarity and length between JUnit 4 and JUnit 5. The JUnit 4 approach needs to work with three classes; the JUnit 5 approach needs to work with only two. The code to be executed before and after each test method is isolated into a dedicated method with a clear name. On the side of the testing class, we only need to annotate it with `@ExtendWith`.

The JUnit 5 extension model may also be used to replace the runners from JUnit 4 gradually. For the extensions that have already been created, the migration process is simple:

- To migrate the Mockito tests, we need to replace in the tested class the annotation `@RunWith(MockitoJUnitRunner.class)` with the annotation `@ExtendWith(MockitoExtension.class)`.

- To migrate the Spring tests, we need to replace in the tested class the annotation `@RunWith(SpringJUnit4ClassRunner.class)` with the annotation `@ExtendWith(SpringExtension.class)`.

When this chapter was written, there was no extension for the Arquillian tests.

The runners will be discussed in more detail in later chapters.

In chapter 5, we find out more about software testing principles and test types, the need to start from unit tests, and ways to advance to different testing levels.

4.4 Summary

This chapter has covered the following:

- Applying the steps needed by the migration from JUnit 4 to JUnit 5 and summarizing them into a guiding table: replace the dependencies; replace the annotations; replace the testing classes and methods; replace the JUnit 4 rules and the runners with the JUnit 5 extension model.
- Using the needed dependencies in both cases, making the step from one single dependency in JUnit 4 to the multiple dependencies in JUnit 5.
- Comparing and using the equivalent annotations, classes and methods between JUnit 4 and JUnit 5, that drive similar effects: definition of tests, controlling the life cycle of a test, checking the results.
- Implementing through examples the effective changes in code when making the step from JUnit 4 to JUnit 5.
- Demonstrating through examples the effective changes needed when moving from the JUnit 4 rules to the JUnit 5 extensions.

5

Software testing principles

This chapter covers

- Examining the need for unit tests
- Differentiating between types of software tests
- Comparing black-box and white-box testing

A crash is when your competitor's program dies. When your program dies, it is an "idiosyncrasy." Frequently, crashes are followed with a message like "ID 02." "ID" is an abbreviation for idiosyncrasy and the number that follows indicates how many more months of testing the product should have had.

—Guy Kawasaki

Earlier chapters in this book took a very pragmatic approach to design and deploy unit tests. I took a deep look at all the JUnit 5 capabilities and at the architecture of both JUnit 4 and JUnit 5 and demonstrated how to migrate between the two versions. This chapter steps back to look at the various types of software tests and the roles they play in the application life cycle.

Why would you need to know all this information? Unit testing is not just something you do without plans and preparations. To become a top-level developer, you need to contrast unit tests with functional and other types of tests. *Functional testing* simply means that you evaluate the compliance of a system or component with the requirements. When you understand why unit tests are necessary, you need to know how far to take your tests. Testing in and of itself is not the goal.

5.1 The need for unit tests

The main goal of unit testing is to verify that your application works as expected and to catch bugs early. Although functional testing helps you accomplish the same goal, unit tests are extremely powerful and versatile and offer much more than simply verifying that the application works. Unit tests

- Allow greater test coverage than functional tests
- Increase team productivity
- Detect regressions and limit the need for debugging.
- Give us the confidence to refactor, and, in general, make changes
- Improve implementation
- Document expected behavior
- Enable code coverage and other metrics

5.1.1 Allowing greater test-coverage

Unit tests are the first type of tests any application should have. If you had to choose between writing unit tests and functional tests, you should choose to write the second kind. In my experience, functional tests cover about 70 percent of the application code. If you want to go further and provide more test coverage, you need to write unit tests.

Unit tests can easily simulate error conditions, which is extremely difficult for functional tests to do (and impossible in some instances). Unit tests provide much more than just testing, as explained in the following sections.

5.1.2 Increasing team productivity

Imagine that you are on a team working on a large application. Unit tests allow you to deliver quality code (tested code) without waiting for all the other components to be ready. On the other hand, functional tests are more coarse-grained and need the full application, or a good part of it, to be ready before you can test it.

5.1.3 Detecting regressions and limiting debugging

A passing unit test suite confirms that your code works and gives you the confidence to modify your existing code, either for refactoring or for adding and modifying new features. As a developer, you can have = no better feeling than knowing someone is watching your back and will warn you if you break something.

A suite of unit tests reduces the need to debug an application to find out why something is failing. Whereas a functional test tells you that a bug exists somewhere in the implementation of a use case, a unit test tells you that a specific method is failing for a specific reason. You no longer need to spend hours trying to find the problem.

5.1.4 Refactoring with confidence

Without unit tests, it is difficult to justify refactoring, because there is always a relatively high chance that you will break something. Why would you risk spending hours of debugging time (and putting delivery at risk) only to improve the implementation or change a method name? As shown in figure 5.1, unit tests provide the safety net that gives you the confidence to refactor.

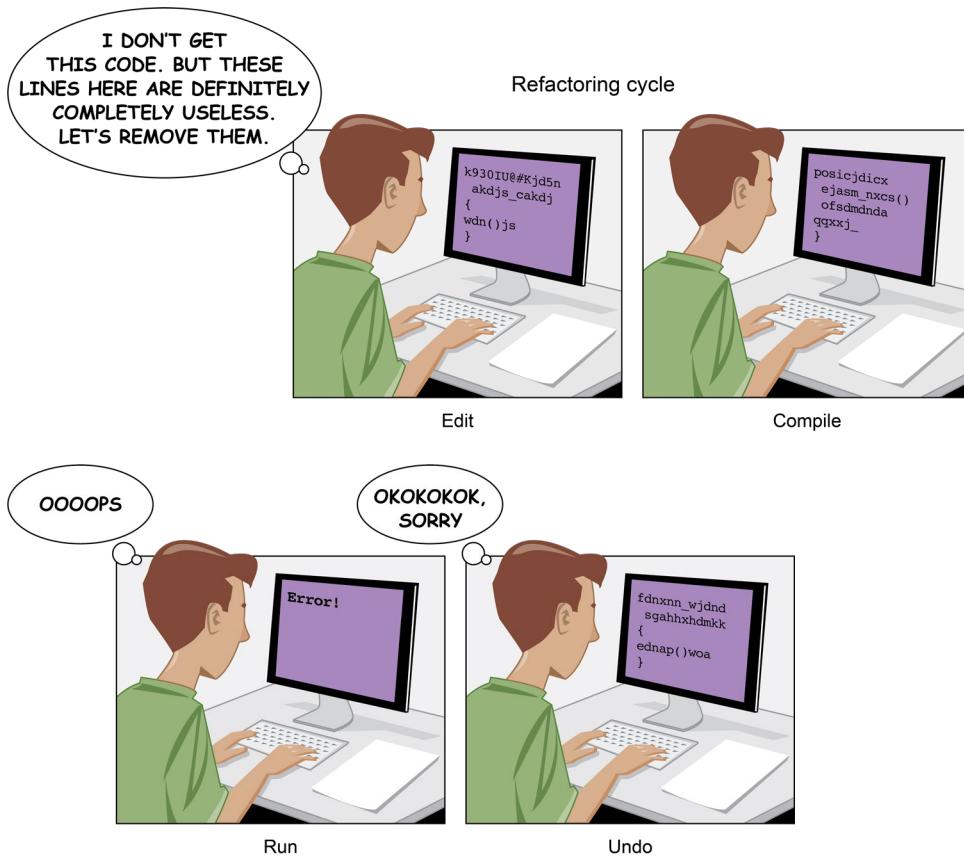


Figure 5.1 Unit tests provide the safety net that gives you the confidence to refactor

JUNIT best practice: Refactor

Throughout the history of computer science, many great teachers have advocated iterative development. Niklaus Wirth, for example, who gave us the now-ancient languages Algol and Pascal, championed techniques such as stepwise refinement.

For a time, these techniques seemed to be difficult to apply to larger, layered applications. Small changes can reverberate throughout a system. Project managers looked to up-front planning as a way to minimize change, but productivity remained low.

The rise of the xUnit framework fueled the popularity of agile methodologies, which (once again) advocate iterative development. Agile methodologists favor writing code in vertical slices to produce a working use case, as opposed to writing code in horizontal slices to provide services layer by layer.

When you design and write code for a single use case or functional chain, your design may be adequate for this feature, but it may not be adequate for the next feature. To retain a design across features, agile methodologies encourage refactoring to adapt the code base as needed.

How do you ensure that *refactoring* (improving the design of existing code) does not break the existing code? The answer is that unit tests tell you when and where code breaks. In short, unit tests give you the confidence to refactor.

The agile methodologies try to lower project risks by enabling you to cope with change. They allow and embrace change by standardizing quick iterations and applying principles such as *YAGNI* (You Ain't Gonna Need It) and *The Simplest Thing That Could Possibly Work*. The foundation upon which all these principles rest, however, is a solid bed of unit tests.

5.1.5 Improving implementation

Unit tests are first-rate clients of the code they test. They force the API under test to be flexible and to be unit-testable in isolation. Sometimes, you have to refactor your code under test to make it unit-testable. You may eventually use the Test Driven Development (TDD) approach, which by its own conception generates code that can be unit-tested. We'll experiment with TDD in detail later in the book (chapter 20).

It is important to monitor your unit tests as you create and modify them. If a unit test is too long and unwieldy, the code under test usually has a design smell, and you should refactor it. You may also be testing too many features in one test method. If a test cannot verify a feature in isolation, the code probably is not flexible enough, and you should refactor it. Modifying code to test it is normal.

5.1.6 Documenting expected behavior

Imagine that you need to learn a new API. On one hand, you have a 300-page document describing the API, and on the other hand, you have some examples of how to use the API. Which would you choose?

The power of examples is well known. Unit tests are examples that show how to use the API. As such, they make excellent developer documentation. Because unit tests match the production code, they *must* be up to date, unlike other forms of documentation.

To examine how tests effective documenting the expected behavior, go back to the `Calculator` class that was introduced in previous chapters. Listing 5.1 illustrates how unit tests help provide documentation. The `expectException()` method shows that an `IllegalArgumentException` is thrown when you try to extract the square root from a negative number. The `expectException()` method shows that an `ArithmaticException` is thrown when you try to divide to zero.

Listing 5.1 Unit tests as automatic documentation

```
public class JUnit5ExceptionTester {  
    private Calculator calculator = new Calculator(); #A  
  
    @Test  
    public void expectIllegalArgumentException() {  
        assertThrows(IllegalArgumentException.class, #B  
                    () -> calculator.sqrt(-1)); #B  
    }  
  
    @Test  
    public void expectArithmaticException() {  
        assertThrows(ArithmaticException.class, #C  
                    () -> calculator.divide(1, 0)); #C  
    }  
}
```

In this example:

- We initialize an instance of the `Calculator` class whose functionality we are testing (#A).
- We assert that the execution of the supplied `calculator.sqrt(-1)` executable throws an `IllegalArgumentException` (#B).
- We assert that the execution of the supplied `calculator.divide(1, 0)` executable throws an `ArithmaticException` (#C).

The execution of the tests clearly shows the use cases. You may follow these use cases and examine what a particular execution will trigger, so the tests are an effective part of the project documentation.

5.1.7 Enabling code coverage and other metrics

Unit tests tell you, at the push of a button, whether everything still works. Furthermore, unit tests enable you to gather code coverage metrics (see chapter 6) that show, statement by statement, what code execution the tests triggered and what code the tests did not touch. You can also use tools to track the progress of passing versus failing tests from one build to the next. Further, you can monitor performance and cause a test to fail if its performance has degraded from a previous build.

5.2 Test types

Figure 5.2 outlines four categories of software tests. There are other ways of categorizing software tests, but these categories are the most useful for the purposes of this book. Please note that this section examines *software tests in general*, not just the automated unit tests covered elsewhere in the book.

In figure 5.2, the outermost tests are broadest in scope. The innermost tests are narrowest in scope. As you move from the inner boxes to the outer boxes, the software tests get more functional, requiring that more of the application be present.



Figure 5.2 The four types of tests, from the innermost (narrowest one) to the outermost (broadest one)

Earlier, I mentioned that each unit test focuses on a distinct unit of work. What about testing different units of work combined into a workflow? Will the result of the workflow do what you expect?

Different kinds of tests answer these questions. I categorize them in these varieties:

- Unit tests
- Integration tests
- System tests
- Acceptance tests

The following sections look at each test type starting with the innermost (the narrowest in scope) and working out to the broadest in scope.

5.2.1 Unit testing

Unit testing is a software testing method in which individual units of source code (methods or classes) are tested to determine whether they are fit for use. Unit testing increases developer confidence in changing the code because from the beginning, it serves as a safety net. If we have good unit tests and run them every time we change the code, we will be certain that our changes are not affecting the existing functionality.

I'm focusing here on testing the classes and methods in isolation. Suppose that the engineers of our example company Tested Data Systems Inc. want to manage the customers of their company. They will first test the creation of the customer. Then, they will test the operations of searching for a customer, of adding and removing him from the company database. Being the narrowest in scope, the unit tests may require only some isolated classes to be present.

5.2.2 Integration software testing

Individual unit tests are essential quality controls, but what happens when different units of work are combined into a workflow? When you have the tests for a class up and running, the next step is hooking up the class with other methods and services. Examining the interaction among components, possibly running in their target environment, is the job of integration testing. Table 5.1 differentiates the various cases under which components interact.

Table 5.1 Testing how object, services, and subsystems interact

Interaction	Test description
Objects	The test instantiates objects and calls methods on these objects. You may use this test type when you would like to see how objects belonging to different classes cooperate to solve the problem.
Services	The test runs while a container hosts the application, which may connect to a database or attach to any other external resource or device. You may use this test type when you are developing an application that is deployed into a software container.
Subsystems	A layered application may have a frontend to handle the presentation and a backend to execute the business logic. Tests can verify that a request passes through the frontend and returns an appropriate response from the backend. You may use this test type in the case of an application with an architecture made of a presentation layer (web interface, for example) and a business service layer executing the logic.

Just as more traffic collisions occur at intersections, the points where objects interact are major contributors to bugs. Ideally, you should define integration tests before you write the application code. Being able to code to the test strongly increases a programmer's ability to write well-behaved objects. The engineers of Tested Data Systems will use integration testing to check whether the objects representing customers and offers cooperate well, such as when a customer is assigned only once to an offer. When the offer expires, the customer is automatically removed from the offer, if he hasn't accepted it; if we have added an offer on the customer side, the customer is added on the offer side.

5.2.3 System software testing

System testing of software is testing conducted on a complete, integrated system to evaluate the system's compliance with its specified requirements. The objective is to detect inconsistencies among units that are already integrated.

Test doubles or mock objects can simulate the behavior of complex, real objects and therefore are useful when a real object (such as some depended-on component) is impractical to incorporate into a test or when testing is impossible—at least for the moment—because the dependent component is not yet available. Mock objects may appear at the level of a unit test, but their role is to replace a part of a system that is not available or impractical to incorporate into a test.

Test doubles are simulated objects that mimic the behavior of real objects in a controlled way. They are created to test the behavior of some other objects that use them.

The engineers at Tested Data Systems will use test doubles when they are communicating with an external service; with an internal one that is not yet available or is slow, hard to configure, or difficult to access; or one that is not yet fully available. A test double is handy, in that it stops your tests from waiting for the availability of that service.

The customers and offers that the Tested Data Systems engineers are managing, for example, need to be persisted to a database. This database is hard to set up and to configure; the engineers must install the software and then create and populate the tables. These processes require time and people. The team will use a mock database for their programs.

5.2.4 Acceptance software testing

It is important that an application performs well, but the application must also meet the customer's needs. Acceptance tests are our final level of testing. The customer or a proxy usually conduct acceptance tests to ensure that the application meets whatever goals the customer or stakeholder defined.

Acceptance tests are supersets of all other tests. They try to answer essential questions, such as whether the application addresses the business goals and is doing the right thing.

Acceptance tests may be expressed by the *Given*, *When*, and *Then* keywords. When you use these keywords, you are practically following a scenario: the interaction of the user with the system. The verification in the *Then* step may look like a unit test, but it checks the end of the scenario and answers the question “Are we addressing the business goals?”

The team at Tested Data Systems may implement some acceptance tests such as this:

- Given that there is an economy offer,
 - When we have a usual customer,
 - We can add him to and remove him from the offer.
-
- Given that there is an economy offer,
 - When we have a VIP customer,
 - We can add him to the offer but cannot remove him from the offer.

Being the broadest in scope, acceptance tests are functional and require a larger part of the application to be present.

5.3 Black-box vs. white-box testing

Before I close this chapter, I'll focus on one other category of software tests: black-box and white-box testing. This category is intuitive and easy to grasp, but developers often forget about it.

5.3.1 Black-box testing

A *black-box test* has no knowledge of the internal state or behavior of the system. The test relies solely on the external system interface to verify its correctness.

As the name of this methodology suggests, you treat the system as a black box. Imagine it with buttons and LEDs. You do not know what is inside or how the system operates; all you know is that when the correct input is provided, the system produces the desired output. All you need to know to test the system properly is the system's functional specification. The early stages of a project typically produce this kind of specification, which means that we can start testing early. Anyone can take part in testing the system, such as a QA engineer, a developer, or even a customer.

The simplest form of black-box testing tries to mimic actions in the user interface manually. A more sophisticated approach is to use a tool for this task, such as HTTPUnit, HTMLUnit, or Selenium. We will apply some of these tools in another part of the book.

At Tested Data Systems, black-box testing is used for applications that provide a web interface. The testing tool knows only that it has to interact with the frontend (selections, pushbuttons, and so on) and to verify the results (the result of the action, the content of the destination page, and so on).

5.3.2 White-box testing

At the other end of the spectrum is *white-box testing*, sometimes called *glass-box testing*. In this type of testing, we use a detailed knowledge of the implementation to create tests and drive the testing process. In addition to understanding component implementation, we need to know how this testing process interacts with other components. For these reasons, the implementers are the best candidates to create white-box tests.

White-box testing can be implemented at an earlier stage. There is no need to wait for the GUI to be available. You may cover many execution paths. In the previous scenarios (section 5.2.4),

- Given that there is an economy offer,
 - When we have a usual customer,
 - We can add him to and remove him from the offer.
-
- Given that there is an economy offer,
 - When we have a VIP customer,
 - We can add him to the offer but cannot remove him from the offer.

These scenarios are good examples of white-box testing. They require the knowledge of the application internals (at least of the API) and cover different execution paths, of which the external user is not aware. Each step corresponds to writing pieces of code that work with the existing API. Developers can apply the code from the early stages without needing a GUI.

Which of the two approaches should you use? There is no absolute answer, so I suggest that you use both approaches. In some situations, you need user-centric tests (no need for details), and in others, you need to test the implementation details of the system. The next section presents the pros and cons of both approaches.

USERCENTRIC APPROACH

Black-box testing first addresses user needs. We know that there is tremendous value in customer feedback, and one of our goals in extreme programming is to release early and release often. We are unlikely to get useful feedback, however, if we just tell the customer "Here it is. Let me know what you think." It is far better to get a customer involved by providing a manual test script to run through. By making the customer think about the application, he can also clarify what the system should do. Interacting with the constructed GUI and comparing the results that are obtained with the expected ones is an example of testing that customers may run by themselves.

TESTING DIFFICULTIES

Black-box tests are more difficult to write and run¹ because they usually deal with a graphical frontend, whether it's a web browser or desktop application. Another issue is that a valid result on the screen does not always mean that the application is correct. White-box tests usually are usually easier to write and run than black-box tests, but the developers must implement them.

Tables 5.2 and 5.3 contrast black-box and white-box testing.

Table 5.2 The pros of black-box and white-box tests

Black-box tests	White-box tests
Tests are usercentric and expose specifications to discrepancies .	Testing can be implemented from the early stages of the project.
The tester may be a nontechnical person.	There is no need for an existing GUI.
Tests can be conducted independently of the developers.	Testing is controlled by the developer and can cover many execution paths.

¹ Black-box testing is getting easier with tools such as Selenium and HtmlUnit, which I examine in chapter 15.

Table 5.3 The cons of black-box and white-box tests

Black-box tests	White-box tests
A limited number of inputs may be tested.	Testing can be implemented only by skilled people with programming knowledge.
Many program paths may be left uncovered.	Tests will need to be rewritten if the implementation changes.
Tests may become redundant; the lack of details may mean covering the same execution paths.	Tests are tightly coupled to the implementation.

TEST COVERAGE

White-box testing provides better test coverage than black-box testing. On the other hand, black-box tests can bring more value than white-box tests. I focus on test coverage in chapter 6.

Although these test distinctions may seem to be academic, recall that divide and conquer does not have to apply only to writing production software; it can also apply to testing. I encourage you to use these different types of tests to provide the best code coverage possible, which will give you the confidence to refactor and evolve your applications.

Chapter 6, which starts part 2 of this book, look at test quality. In it, I present best practices such as measuring test coverage and writing testable code, and I introduce Test Driven Development, Behavior Driven Development, and mutation testing.

5.4 Summary

This chapter has covered the following:

- Examining the need for unit tests. They offer greater test coverage than functional tests; increase team productivity; detect regressions; give the confidence to refactor; improve implementation; document expected behavior; enable code coverage.
- Comparing different types of software testing. From the narrowest to the broadest scope, they are: unit testing, integration testing, system testing, and acceptance testing.
- Contrasting the black box testing and white box testing, examining the pros and cons for the usage of each of them. Black box testing has no knowledge of the internal state or behavior of the system and relies solely on the external system interface to verify its correctness; white box testing is controlled by the developer, can cover many execution paths and provides better test coverage.

6

Test quality

This chapter covers

- Measuring test coverage
- Writing testable code
- Investigating Test Driven Development
- Investigating Behavior Driven Development
- Introducing mutation testing
- Testing in the development cycle

I don't think anybody tests enough of anything.

—James Gosling

In the previous chapters, I introduced testing software, began to explore testing with JUnit, and presented different test methodologies.

Now that you are writing test cases, it is time to measure how good these tests are by using a test coverage tool to report what code is executed by the tests and what code is not. I also discuss how to write code that is easy to test and finish by taking a first look at Test Driven Development (TDD).

6.1 Measuring test coverage

Writing unit tests gives you the confidence to change and refactor an application. As you make changes, you run tests, which give you immediate feedback on new features under test and whether changes break the existing tests. The issue is that these changes may still break untested functionality.

To resolve this issue, you need to know precisely what code runs when you or the build invoke tests. Ideally, your tests should cover 100 percent of your application code. This section examines the test coverage in detail.

Test coverage can ensure some of the quality of your programming, but it is also a controversial metric. High code coverage does not tell you anything about the quality of the tests. One good programmer should be able to see beyond the pure percentage obtained by running tests.

6.1.1 Introduction to test coverage

Using black-box testing, we can create tests that cover the public API of an application. Because we are using documentation—not knowledge of the implementation—as our guide, we do not create tests that use special parameter values to exercise special conditions in the code, for example.

Many metrics can be used to calculate test coverage. Some of the most basic is the percentage of program methods and the percentage of program lines called during execution of the test suite.

One metric of test coverage is tracking which methods the tests call. The result does not tell you whether the tests are complete, but it does tell you whether you have a test for a method. Figure 6.1 shows the partial test coverage typically achieved by black-box testing alone.

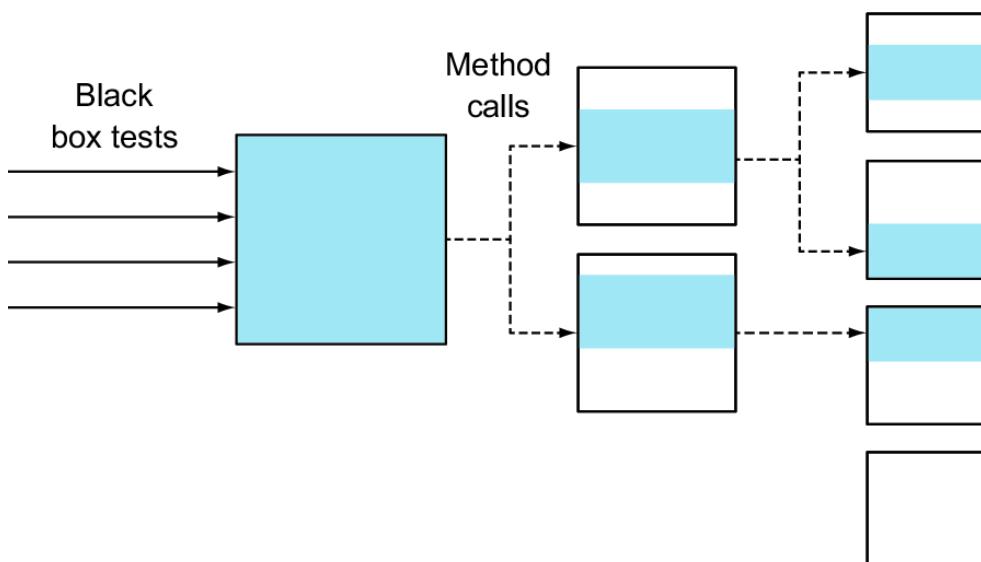


Figure 6.1 Partial test coverage with black-box tests. The boxes represent components or modules. The blue areas represent the portions of the system effectively covered by tests, and the white areas represent the untested portions.

You *can* write a unit test with intimate knowledge of a method's implementation. If a method contains a conditional branch, you can write two unit tests, one for each branch. Because you need to see into the method to create such a test, this type of test falls in the category of white-box testing. Figure 6.2 shows 100 percent test coverage with white-box testing.

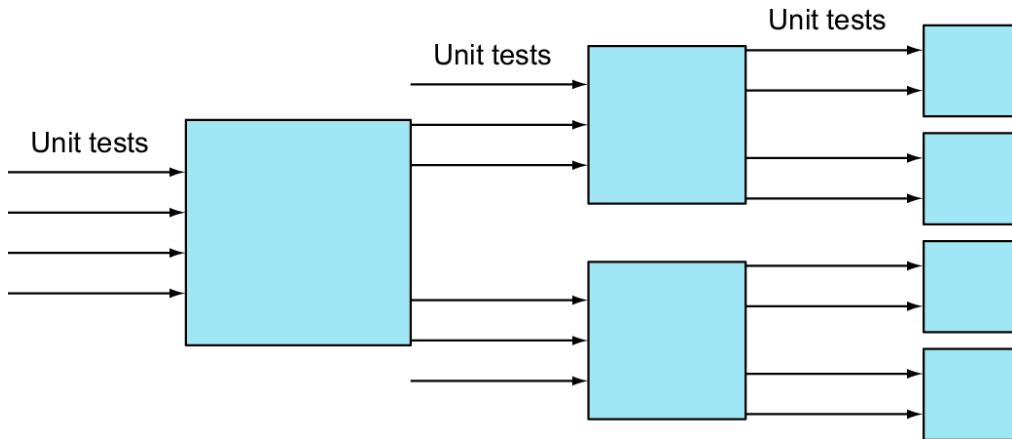


Figure 6.2 Complete test coverage with white-box tests. Each box represents a component or a module. The boxes are fully blue, as they are fully covered by tests,

You can achieve higher test coverage by using white-box unit tests, because you have access to more methods and because you can control the inputs to each method and the behavior of secondary objects (using stubs or mock objects, as you see in later chapters). Because you can write white-box unit tests against protected, package-private, and public methods, you get more code coverage.

If you haven't been able to get high code coverage with black-box testing, you need more tests (you may have uncovered ways to use the application), or you may have superfluous code that does not contribute to the business goals. In either case, an analysis may be required to reveal the real causes.

A program with high test coverage, measured as a percentage, has more of its source code executed during testing, which suggests that it has a lower chance of containing undetected software bugs compared with a program with low test coverage.

The metric is built by a tool that runs the test suite and analyzes the code that is effectively executed as a result.

6.1.2 Code coverage measuring tools

This section demonstrates the use of code coverage measuring tools with the help of the source code provided for this chapter. We'll use the `com.manning.junitbook.ch06` package, which gets back to the `Calculator` and `CalculatorTest` classes.

A few code coverage tools are well integrated with JUnit. A very convenient way to generate the code coverage would be to work directly from IntelliJ IDEA. The IDE provides the possibility of executing the tests with coverage, as shown in figure 6.3.

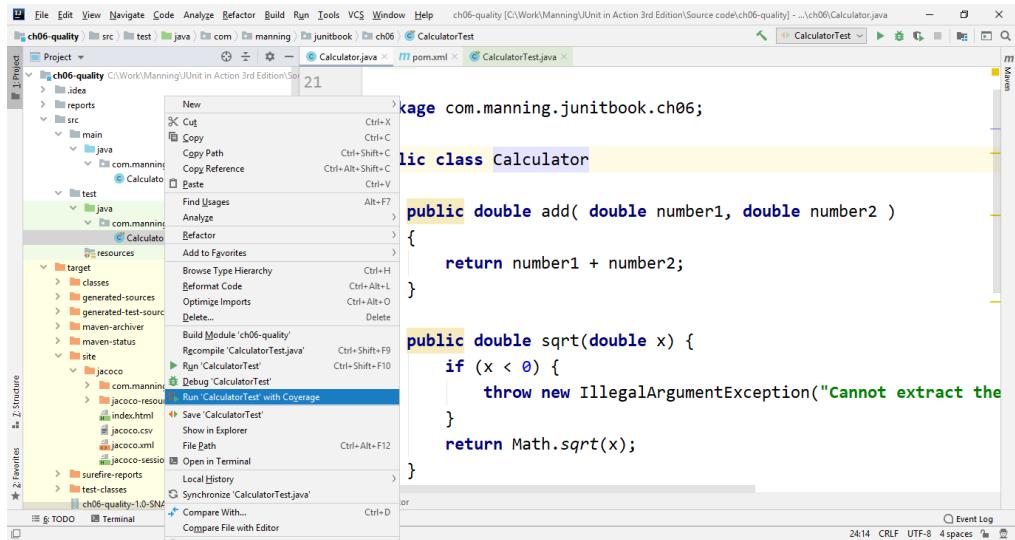


Figure 6.3 Running tests with coverage in IntelliJ

Running the code with coverage, you get a first report (figure 6.4). You can click the Generate Coverage Report button, which allows you to create reports in HTML format.

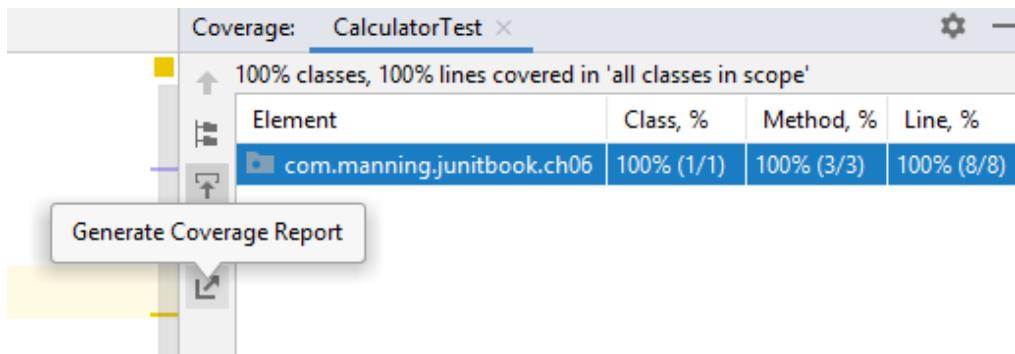


Figure 6.4 The result of running the tests with coverage

The HTML reports get at the level of the `com.manning.junitbook.ch06` package (figure 6.5), individual `Calculator` class (figure 6.6), or individual lines of code (figure 6.7).

[all classes]	
Overall Coverage Summary	
Package	Class, %
all classes	100% (1/ 1)
Coverage Breakdown	
Package	Class, %
com.manning.junitbook.ch06	100% (1/ 1)

Figure 6.5 The code coverage report at the level of a package

[all classes] [com.manning.junitbook.ch06]	
Coverage Summary for Package: com.manning.junitbook.ch06	
Package	Class, %
com.manning.junitbook.ch06	100% (1/ 1)
Class	
Calculator	Class, %
	100% (1/ 1)

Figure 6.6 The code coverage at the level of individual classes

```

21
22 package com.manning.junitbook.ch06;
23
24 public class Calculator {
25     public double add( double number1, double number2 ) {
26         return number1 + number2;
27     }
28
29     public double sqrt(double x) {
30         if (x < 0) {
31             throw new IllegalArgumentException("Cannot extract the square root of a negative value");
32         }
33         return Math.sqrt(x);
34     }
35
36     public double divide(double x, double y) {
37         if (y == 0) {
38             throw new ArithmeticException("Cannot divide by zero");
39         }
40         return x/y;
41     }
42 }
43

```

Figure 6.7 The code coverage report at the level of individual lines

The recommended way to work is to execute the tests from the command line with code coverage, it works fine in conjunction with the Continuous Integration/Continuous Development pipeline. You may use the JaCoCo (Java Code Coverage) tool. JaCoCo is an open-source toolkit that is updated frequently and has very good integration with Maven. To make Maven and JaCoCo work together, you need to insert the JaCoCo plugin information into the pom.xml file (figure 6.8).

```
<plugin>
    <groupId>org.jacoco</groupId>
    <artifactId>jacoco-maven-plugin</artifactId>
    <version>0.7.9</version>
    <executions>
        <execution>
            <goals>
                <goal>prepare-agent</goal>
            </goals>
        </execution>
        <execution>
            <id>report</id>
            <phase>test</phase>
            <goals>
                <goal>report</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

Figure 6.8 The JaCoCo plugin configuration in the Maven pom.xml file

You need to go to the command prompt and run `mvn test` (figure 6.9). This command also generates—for the `com.manning.junitbook` package and for the `Calculator` class—the code coverage report, which you can access from the project folder, `target\site\jacoco` (figure 6.10). I've chosen to show an example that does not have 100 percent code coverage. You can also do this by removing some of the existing tests from the `CalculatorTest` class in the accompanying `ch06-quality` folder.

```
C:\Windows\system32\cmd.exe
C:\Work\Manning\JUnit in Action 3rd Edition\Source code\ch06-quality>mvn test
```

Figure 6.9 Running `mvn test` from the command prompt

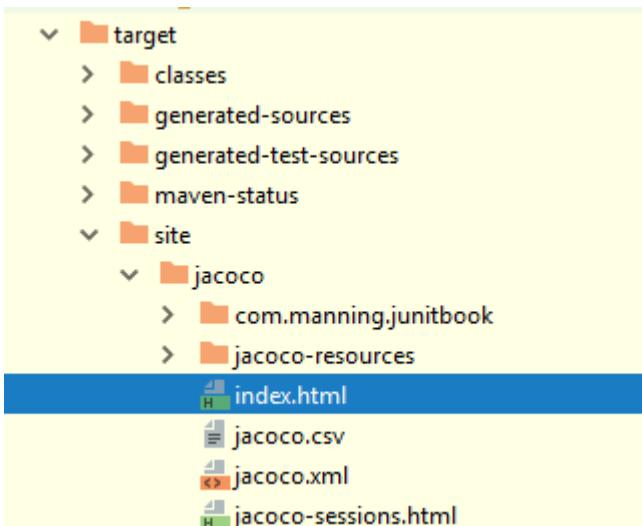


Figure 6.10 The code coverage report, accessible in the project folder target\site\jacoco

From here, you can access the reports at the level of the com.manning.junitbook package (figure 6.11), Calculator class (figure 6.12), methods (figure 6.13), and individual lines (figure 6.14).

ch06-quality

Element	Missed Instructions	Cov.	Missed Branches	Cov.
com.manning.junitbook.ch06	84%	75%		
Total	5 of 32	84%	1 of 4	75%

Figure 6.11 JaCoCo report at the level of packages

com.manning.junitbook.ch06

Element	Missed Instructions	Cov.	Missed Branches	Cov.
 Calculator	 84%		 75%	
Total	5 of 32	84%	1 of 4	75%

Figure 6.12 JaCoCo report at the level of classes

Calculator

Element	Missed Instructions	Cov.	Missed Branches	Cov.	I
 sqrt(double)	 58%		 50%		
 divide(double, double)	 100%		 100%		
 add(double, double)	 100%				n/a
 Calculator()	 100%				n/a
Total	5 of 32	84%	1 of 4	75%	

Figure 6.13 JaCoCo report at the level of methods

```

21.
22. package com.manning.junitbook.ch06;
23.
24. public class Calculator {
25.     public double add( double number1, double number2 ) {
26.         return number1 + number2;
27.     }
28.
29.
30.     public double sqrt(double x) {
31.         if (x < 0) {
32.             throw new IllegalArgumentException("Cannot extract the square root of a negative value");
33.         }
34.         return Math.sqrt(x);
35.     }
36.
37.
38.     public double divide(double x, double y) {
39.         if (y == 0) {
40.             throw new ArithmeticException("Cannot divide by zero");
41.         }
42.         return x/y;
43.     }
44.
45. }

```

Figure 6.14 JaCoCo report at the level of individual lines: a covered line is green, an uncovered line is red, a partially covered condition is yellow

6.2 Writing testable code

This chapter is dedicated to best practices in software testing. Now you’re ready to get to the next level: writing code that is easy to test. Sometimes, writing a single test case is easy; sometimes, it is not. Everything depends on the complexity of the application. A best practice avoids complexity as much as possible; code should be readable and testable.

In this section, I discuss some best practices that can improve your design and code. Remember that it is always easier to write easily testable code than it is to refactor existing code to make it easily testable.

6.2.1 Understand that public APIs are contracts

One principle in providing backward-compatible software states “Never change the signature of a public method.” An application code review shows that most calls are made from external applications, which play the role of clients of your API. If you change the signature of a public method, you need to change every call site in the application and unit tests. Even with refactoring wizards in tools such as IntelliJ or Eclipse, you must always perform this task with care. During initial development, especially if working TDD (Test Driven Development), it is a great time to refactor the API as needed, because you do not yet have any public users. Things will change afterward!

In the open-source world, and for any API made public by a commercial product, life can get even more complicated. Many people use your code, and you should be very careful about the changes you make to stay backward-compatible.

Public methods become the articulation points of an application between components, open-source projects, and commercial products that usually do not know of one another’s existence.

Imagine a public method that takes a distance as a `double` parameter and uses a black-box test to verify a computation. At some point, the meaning of the parameter changes from miles to kilometers. Your code still compiles, but the run time breaks. Without a unit test to fail and tell you what is wrong, you may spend a lot of time debugging and talking to angry customers.

Mismatched units are the reason for which NASA Mars Climate Observer was lost in 1999; one part of the system calculated the amount of thrust needed to adjust course and reduce speed, another took the info and used it to determine how long to fire thrusters. The problem was that one system used imperial measurements (foot-pounds), while the other part used metric measurements (Newton) and there was no test to catch the mistake...

This example illustrates that you must test all public methods. For nonpublic methods, you need to go to a deeper level and use white-box tests.

6.2.2 Reduce dependencies

Remember that unit tests verify your code in isolation. Your unit tests should instantiate the class you want to test, use it, and assert its correctness. Your test cases should be simple. What happens when your class instantiates a new set of objects, directly or indirectly? Now your class depends on these classes. To write testable code, you should reduce dependencies as much as possible. If your classes depend on many other classes that need to be instantiated and set up with some state, your tests will be very complicated; you may need to use a complicated mock-objects solution (see chapter 8).

A solution that reduces dependencies separates methods that instantiate new objects (factories) from methods that provide your application logic. At *Tested Data Systems Inc.*, one of the projects under development is in the automotive field. The project has to manage vehicles and drivers, and there are classes that shape them. Consider listing 6.1.

Listing 6.1 Reduce dependencies

```
class Vehicle {  
  
    Driver d = new Driver();  
    boolean hasDriver = true;  
  
    private void setHasDriver(boolean hasDriver) {  
        this.hasDriver = hasDriver;  
    }  
}
```

Every time the `Vehicle` class is instantiated with an object, the `Driver` class is also instantiated with an object. The concepts are mixed. The solution is to have the `Driver` interface passed to the `Vehicle` class, as shown in listing 6.2.

Listing 6.2 Pass the Driver to the Vehicle

```
class Vehicle {  
  
    Driver d;
```

```

boolean hasDriver = true;

Vehicle(Driver d) {
    this.d = d;
}

private void setHasDriver(boolean hasDriver) {
    this.hasDriver = hasDriver;
}
}

```

This code allows you to produce a mock Driver object (see chapter 8) and pass it to the Vehicle class on instantiation. This process is *dependency injection*—a technique in which a dependency is supplied to another object. You can mock any other type of Driver implementation. The requirements for the engineers at Tested Data Systems *may introduce specific classes such as* JuniorDriver and SeniorDriver, and pass those classes to the Vehicle class. Through dependency injection, the code supplies the Driver object to the Vehicle object by invoking the Vehicle(Driver d) constructor.

6.2.3 Create simple constructors

By striving for better test coverage, you add more test cases. In each of these test cases, you

- Instantiate the class to test.
- Set the class to a particular state.
- Assert the final state of the class.

By doing work in the constructor (other than populating instance variables), you mix the first and the second point in listing 6.3. It is a bad practice not only from the design point of view (you do the same work every time you instantiate your class), but also because you get your class in a predefined state. This code is hard to maintain and test. Setting the class to a particular state should be a separate operation, as shown in listing 6.4.

Listing 6.3 Setting the class to a particular state inside the constructor

```

class Car {
    private int maxSpeed;

    Car() {
        this.maxSpeed = 180;
    }
}

```

Listing 6.4 Setting the class to a particular state by using a setter

```

class Car {
    private int maxSpeed;

    public void setMaxSpeed(int maxSpeed) {
        this.maxSpeed = maxSpeed;
    }
}

```

6.2.4 Follow the Law of Demeter (Principle of Least Knowledge)

The Law of Demeter, or *Principle of Least Knowledge*, states that one class should know only as much as it needs to know.

The Law of Demeter may also be described this way:

Talk to your immediate friends.

or

Don't talk to strangers.

Consider the example violation in listing 6.5.

Listing 6.5 Law of Demeter violation

```
class Car {  
    private Driver driver;  
  
    Car(Context context) {  
        this.driver = context.getDriver();  
    }  
}
```

In this example, you pass to the `Car` constructor a `Context` object. This object shapes some environment attributes, such as whether the drive is long-distance, daytime, and/or nighttime. This code violates the Law of Demeter because the `Car` class needs to know that the `Context` object has a `getDriver` method. If you want to test this constructor, you need to get hold of a valid `Context` object before calling the constructor. If the `Context` object has a lot of variables and methods, you could be forced to use mock objects (see chapter 8) to simulate the context.

The proper solution is to apply the Law of Demeter and pass references to methods and constructors only when you need to do so. *In this example, you should pass the `Driver` to the `Car` constructor:*

```
Car(Driver driver) {  
    this.driver = driver;  
}
```

These concepts are key: require objects, do not search for objects, and ask only for objects that your application requires.

Misko Hevery's society analogy

You can live in a society in which everyone (every class) declares who their friends (collaborators) are. If I know that Joe knows Mary, but neither Mary nor Joe knows Tim, it is safe for me to assume that if I give some information to Joe, he may give it to Mary, but under no circumstances will Tim get hold of it. Now imagine that everyone (every class)

declares some of their friends (collaborators), but other friends are kept secret. Now you are left wondering how in the world Tim got hold of the information you gave to Joe. This is the analogy that Miško Hevery is providing on his blog.

Here is the interesting part: If you are the person who built the relationships (code), you know the true dependencies, but anyone who comes after you is baffled, because the friends that are declared are not the sole friends of objects, and information flows into some secret paths that are not clear to you. You live in a society full of liars.

6.2.5 Avoid hidden dependencies and global state

Be very careful with the global state, because the global state makes it possible for many clients to share the global object. This sharing can have unintended consequences if the global object is not coded for shared access or if clients expect exclusive access to the global object.

At Tested Data Systems, the internal organization has to install a database that is under the control of the database manager. Some reservations must be created for internal meetings and appointments. Consider the example in listing 6.6.

Listing 6.6 Global state in action

```
public void makeReservation() {
    Reservation reservation = new Reservation();
    reservation.makeReservation();
}

public class Reservation {
public void makeReservation() {
    manager.initDatabase(); //manager is a reference to a global
                           //DBManager, already initialized
    //require the global DBManager to do more action
}
}
```

The DBManager implies a global state. Without instantiating the database, you will not be able to make a reservation. Internally, the Reservation uses the DBManager to access the database. Unless it's documented, the Reservation class hides its dependency on the database manager from the programmer, because the API does not provide a clue.

Listing 6.7 provides a better implementation. The DBManager dependency is injected into the Reservation object by calling the constructor with an argument.

Listing 6.7 Avoiding global state

```
public void makeReservation() {
    DBManager manager = new DBManager();
    manager.initDatabase();
    Reservation reservation = new Reservation(manager);
    reservation.makeReservation();
}
```

In this example, the reservation object is constructed with a given database manager. Strictly speaking, the reservation object should be able to function only if it has been configured with a database manager.

Avoid the global state. When you provide access to a global object, you share not only that object, but also any object to which it refers.

6.2.6 Favor generic methods

Static methods, like factory methods, are very useful, but large groups of utility static methods can introduce issues of their own. Recall that unit testing is testing in isolation. To achieve isolation, you need some articulation points in your code, where you can easily substitute your code for the test code. These points use polymorphism. With *polymorphism* (the ability of one object to pass more than one IS-A tests), the method you are calling is not determined at compile time. You can easily use polymorphism to substitute application code for the test code to force certain code patterns to be tested.

The opposite situation occurs when you use nothing but static methods. Then you practice procedural programming, and all your method calls are determined at compile time. You no longer have articulation points that you can substitute.

Sometimes, the harm of static methods to your test is not great, especially when you chose some method that ends the execution, such as `Math.sqrt()`. On the other side, you can choose a method that lies in the heart of your application logic. In that case, every method that gets executed inside that static method becomes hard to test.

Static code and the inability to use polymorphism in your application affect your application and tests equally. No polymorphism means no code reuse for both your application and your tests. This situation can lead to code duplication in the application and tests, which you should try to avoid.

Consequently, static utility methods that operate on parameterized types should be generic. Consider this method, which returns the union of two sets:

```
public static Set union(Set s1, Set s2) {  
    Set result = new HashSet(s1);  
    result.addAll(s2);  
    return result;  
}
```

This method compiles, but with two warnings:

```
Union.java:5: warning: [unchecked] unchecked call to  
HashSet(Collection<? extends E>) as a member of raw type HashSet  
Set result = new HashSet(s1);  
^  
Union.java:6: warning: [unchecked] unchecked call to  
addAll(Collection<? extends E>) as a member of raw type Set  
result.addAll(s2);  
^
```

You would like to make the method typesafe and eliminate the warnings. Modify it to declare a type parameter representing the element type for the three sets (the two arguments and the return value), and use this type parameter throughout the method. The type parameter list will be `<E>`, and the return type will be `Set<E>`:

```
public static <E> Set<E> union(Set<E> s1, Set<E> s2) {  
    Set<E> result = new HashSet<>(s1);  
    result.addAll(s2);  
    return result;  
}
```

This method not only compiles without generating any warnings, but also provides type safety, ease of use, and ease of testing.

6.2.7 Favor composition over inheritance

Many people choose inheritance as a code-reuse mechanism. I think that composition can be easier to test. At run time, code cannot change an inheritance hierarchy, but you can compose objects differently. What you strive for is to make your code as flexible as possible at run time. This way, you can be sure that it is easy to switch from one state of your objects to another, which makes the code easily testable.

It is safe to use inheritance within a package, where the subclass and the superclass are under the control of the same programmers. Inheriting from ordinary concrete classes across package boundaries may be risky.

Inheritance is appropriate only when the subclass is a subtype of the superclass. When you have to have classes, A and B, we may ask ourselves if we can relate them. Class B should extend class A only if an IS-A relationship exists between the two classes.

You also encounter violations of this principle in the Java platform libraries: a stack is not a vector, so `Stack` should not extend `Vector`. Similarly, a property list is not a hash table, so `Properties` should not extend `Hashtable`. In both cases, the composition would be preferable.

6.2.8 Favor polymorphism over conditionals

As I mentioned previously, all you do in your tests is

- Instantiate the class to test.
- Set the class to a particular state.
- Assert the final state of the class.

Difficulties may arise at any of these points. Instantiating your class may be difficult if the class is too complex, for example.

One of the main ways to decrease complexity is to try to avoid long multiple-choice `switch` statements or `if` statements. Consider listing 6.8.

Listing 6.8 Example of bad design with conditionals

```
public class DocumentPrinter {  
    [...]  
    public void printDocument() {  
        switch (document.getDocumentType()) {  
            case Documents.WORD_DOCUMENT:  
                printWORDDocument();  
                break;
```

```

        case Documents.PDF_DOCUMENT:
            printPDFDocument();
            break;
        case Documents.TEXT_DOCUMENT:
            printTextDocument();
            break;
        default:
            printBinaryDocument();
            break;
    }
}
[...]
}

```

This implementation is awful for several reasons, and the code is hard to test and maintain. Every time you want to add a new document type, you need additional `case` clauses. If such a situation happens often in your code you will have to change it in every place that it occurs.

TIP Every time you see a long conditional statement, think of polymorphism. Polymorphism is a natural object-oriented way to avoid long conditionals by breaking a class into several smaller classes. Several smaller components are easier to test than one large complex component.

In this example, you can avoid the conditional by creating different document types, such as `WordDocument`, `PDFDocument`, and `TextDocument`, each of which implements a `printDocument()` method (listing 6.9). This solution decreases the complexity of the code and makes it easier to read. When the `printDocument(Document)` method of the `DocumentPrinter` class is called, it delegates to `printDocument()` from `Document`. The effective method to be executed is determined at run time through polymorphism, and the code is easier to understand and test because it is not cluttered with conditionals.

Listing 6.9 Replacing conditionals with polymorphism

```

public class DocumentPrinter {
    [...]
    public void printDocument(Document document) {
        document.printDocument();
    }
}

public abstract class Document {
    [...]
    public abstract void printDocument();
}

public class WordDocument {
    [...]
    public void printDocument() {
        printWORDDocument();
    }
}

public class PDFDocument {
    [...]
}

```

```
public void printDocument() {
    printPDFDocument();
}
}

public class TextDocument {
[...]
    public void printDocument() {
        printTextDocument();
    }
}
```

6.3 Test Driven Development

As you design an application, tests may help you improve the initial design. As you write more unit tests, positive reinforcement encourages you to write them earlier. As you design and implement, it becomes natural to wonder how you will test a class. Following this methodology, more developers are making the leap from test-friendly designs to Test Driven Development.

DEFINITION *Test Driven Development (TDD)*—Test Driven Development is a programming practice that instructs developers to write the tests first and then write the code that makes the software pass those tests. Then, the developer should examine the code and refactor it to clean up any mess or improve things. The goal of TDD is clean code that works.

6.3.1 Adapting the development cycle

When you develop code, you design an application programming interface (API) and then implement the behavior promised by the interface. When you unit-test code, you verify the promised behavior through an API. The test is a client of the API, just as your domain code is a client of the API.

The conventional development cycle goes something like this:

[code, test, (repeat)]

Developers who practice TDD make a seemingly slight but surprisingly effective adjustment:

[test, code, (repeat)]

The test drives the design and becomes the first client of the method, as opposed to software development that allows the addition of software that does not prove to meet requirements.

This approach has a few advantages:

- You write code that is driven by clear goals and can be sure that you address exactly what your application needs to do. Tests represent a means to design the code.
- You can introduce new functionality much faster. Tests drive you to implement the code that does what it is supposed to do.
- Tests prevent you from introducing bugs into existing code that is working well.
- Tests serve as documentation, so you can follow them and understand what problems

the code is supposed to solve.

6.3.2 Doing the TDD two-step

Earlier, I said that TDD tweaks the development cycle to go something like

[test, code, (repeat)]

The problem with this process is that it leaves out a key step. Development should go more like this:

[test, code, refactor, (repeat)].

Refactoring is the process of changing a software system in such a way that it improves the code's internal structure without altering its external behavior. To make sure that external behavior is not affected, you need to rely on tests.

The core tenets of TDD are

- Write a failing test before writing new code.
- Write the smallest piece of code that will make the new test pass.

Developers who follow this practice have found that test-backed, well-factored code is, by its very nature, easy *and safe* to change. TDD gives you the confidence to solve today's problems today and tomorrow's problems tomorrow. *Carpe diem!* Chapter 20 is dedicated to using TDD with JUnit 5.

JUNIT best practice: write failing tests first

If you take the TDD development pattern to heart, an interesting thing happens: before you can write any code, you must write a test that fails. Why does it fail? Because you have not written the code to make it succeed.

Faced with this situation, most of us begin by writing a simple implementation to let the test pass. When the test succeeds, you could stop and move on to the next problem. Being a professional, you would take a few minutes to refactor the implementation to remove redundancy, clarify intent, and optimize the investment in the new code. But as long as the test succeeds, technically, you're done.

If you always test first, you will never write a line of new code without a failing test.

6.4 Behavior Driven Development

Behavior Driven Development, which emerged during the mid-2000s, is a methodology for developing IT solutions that satisfy business requirements directly. Its philosophy is driven by business strategy, requirements, and goals, which are refined and transformed into an IT solution.

Whereas TDD helps you build good-quality software, BDD helps you build software that's worth building to solve the problems of the users.

Dan North is the originator of Behavior Driven Development, a philosophy of software development that encourages teams to deliver the software that matters by emphasizing interactions among stakeholders.

BDD helps you write software in a way that lets you discover and focus your efforts on what really matters. You find out what features will really benefit the organization and effective ways to implement them. You see beyond what the user asks for and build what the user actually needs.

A natural question that arises is what gives business value to the software. The answer is that working features are providing business value to the software. A feature is a tangible, deliverable piece of functionality that helps the business achieve its business goals. Also, software that can easily be maintained and enhanced is more valuable than software that is hard to change without breaking, so following a practice like BDD or TDD adds value.

To address the business goals, the business analyst works with the customer to decide what software features achieve these goals. These features are high-level requirements, such as "Provide a way for a customer to choose among a few alternative routes to the destination" and "Provide a way for a customer to choose the optimal route to the destination." These features will need to be broken into stories. The stories might be "Find the route between source and destination with the smallest number of flight changes" or "Find the quickest route between source and destination." Stories need to be described in terms of concrete examples, which become the acceptance criteria for the story.

Acceptance criteria may be expressed BDD-style in a way that can be automated later. The keywords are *given*, *when*, and *then*.

Following is an example of acceptance criteria:

Given the flights operated by company X,

When I want to find the quickest route from Bucharest to New York from May 15 to 20,

Then I will be provided the route Bucharest–Frankfurt–New York.

Chapter 21 discusses BDD with JUnit 5.

6.5 Mutation testing

If you strive for testing at different levels (from unit testing to acceptance testing) and obtain high code coverage (as close to 100 percent as possible), how sure can you be that your code is close to perfection?

The bad news is that perfection may not be enough! Even 100 percent code coverage does not mean that your code works perfectly. Are you surprised? You should not be, because your tests may not be good enough. The simplest way to let this happen is to omit assertions in the tests. This may be done for reasons like the output being too complex for the test to verify, so we'll display it, or log it, and let a person decide.

How can you check the quality of the tests and make that they do what they are supposed to do?

Mutation testing comes into play in this situation. *Mutation testing* (or *mutation analysis* or *program mutation*) is used to design new software tests and evaluate the quality of existing software tests.

The basic idea of mutation testing involves modifying a program in small ways. Each mutated version is called a *mutant*. The behavior of the original version differs from that of the mutant. Tests detect and reject mutants, which is called *killing the mutant*. Test suites are measured by the percentage of mutants that they kill. New tests can be designed to kill additional mutants.

Mutants are generated by well-defined mutation operators that change an existing operator for another one or invert some conditions. The goal is to support the creation of effective tests or to locate weaknesses in the test data used for the program or sections of the code that are seldom or never accessed during execution. Consequently, mutation testing is a form of white-box testing.

Given the following piece of code,

```
if(a) {  
    b = 1;  
} else {  
    b = 2;  
}
```

the mutation operation may reverse the condition so that the newly tested piece of code is

```
if(!a) {  
    b = 1;  
} else {  
    b = 2;  
}
```

A strong mutation test fulfills the following three conditions:

- The test reaches the `if` condition that has been mutated.
- The test continues on a different branch from the initial correct one.
- The changed value of `b` propagates to the output of the program and is checked by the test.
- The test will fail because the method returned the wrong value for `b`.

Well-written tests must bring to the failure of mutated tests to demonstrate that they initially covered the necessary logical conditions.

The best-known Java mutation testing framework is PITest (Parallel Isolated Test), which changes the Java bytecode to generate mutants. Mutation testing exceeds the scope of this book, but I considered to introduce its main ideas in the context of discussing test quality.

6.6 Testing in the development cycle

Testing occurs at different places and times during the development cycle. This section first introduces a development life cycle and then uses it as a basis for deciding what types of tests are executed when. Figure 6.15 shows a typical development cycle.

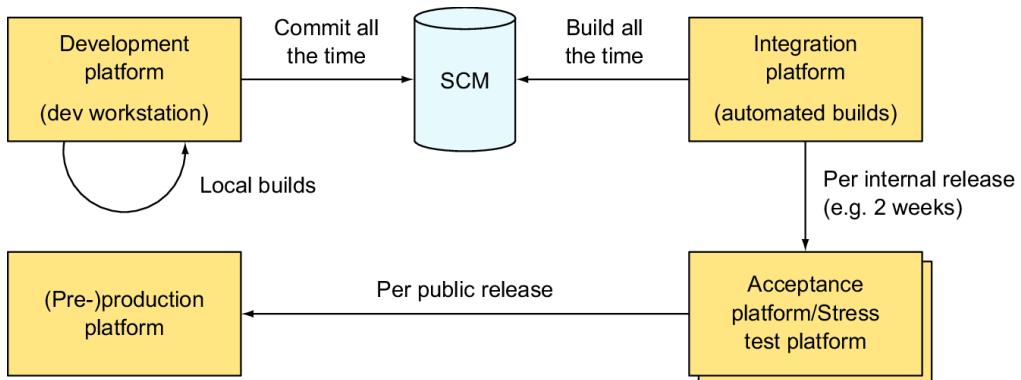


Figure 6.15 A typical application development life cycle, using the continuous integration principle

The life cycle is divided into the following platforms:

- *Development*—This platform is where coding happens on developers' workstations. One important rule is to commit (check in), up to several times per day to your source control management (SCM) system (Git, SVN, CVS, ClearCase, and so on) several times per day. After you commit, others can begin using your work. It is important to commit only something that works. To ensure this, you should first merge your changes with the current code in the repo. Then, you should run a local build with Maven or Gradle. Also, you can watch the results of an automated build based on the latest changes to the SCM repository.
- *Integration*—This platform builds the application from its components (which may have been developed by different teams) and ensures that the components work together. This step is extremely valuable because problems are often discovered here. The step is so valuable, in fact, that you want to automate it. After automation, the process is called *continuous integration* (see <http://www.martinfowler.com/articles/continuousIntgration.html>). You can achieve continuous integration by building the application automatically as part of the build process (see chapter 13).
- *Acceptance/stress test*—Depending on the resources available to your project, this platform can be one or two platforms. The stress test platform exercises the application under load and verifies that it scales correctly (with respect to size and response time). The acceptance platform is where the project's customers accept (sign off on) the

system. It is highly recommended for the system to be deployed on the acceptance platform as often as possible to get user feedback.

- *(Pre)production* —The preproduction platform is the last staging area before production. This platform is optional, and small or noncritical projects can do without it.

Next, I discuss how testing fits into the development cycle. Figure 6.16 highlights the types of tests you can perform on each platform.

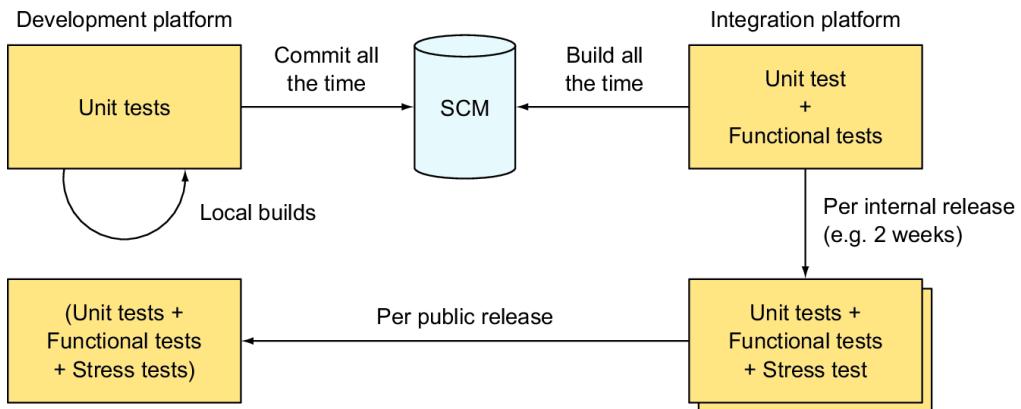


Figure 6.16 The different types of tests performed on each platform of the development cycle

- On the *development platform*, you execute *logic unit tests* (tests that can be executed in isolation from the environment). These tests execute very quickly, and you usually execute them from your IDE to verify that any change you have made in the code has not broken anything. These tests are also executed by your automated build before you commit the code to your SCM. You could also execute integration tests, but they often take much longer because they need some part of the environment to be set up (database, application server, and so on). In practice, you execute only a subset of all integration unit tests, including any new integration unit tests that you have written.
- The *integration platform* usually runs the build process automatically to package and deploy the application; then it executes unit and functional tests. *Functional testing* means evaluating the compliance of a system or component with the requirements. This black-box testing type usually describes what the system does. Usually, only a subset of all functional tests is run on the integration platform, because compared with the target production platform, integration is a simple platform that lacks elements. (It may be missing a connection to an external system being accessed, for example.) All tests are executed on the integration platform. Time is less important, and the whole build can take several hours with no effect on development.
- On the *acceptance platform/stress test platform*, you execute the same tests executed by the integration platform; in addition, you run stress tests (to verify the robustness

and performance of the software). The acceptance platform is extremely close to the production platform, and more functional tests can be executed here.

- A good habit is to run on the *(pre)production platform* the tests you ran on the acceptance platform. This technique acts as a sanity check, verifying that everything is set up correctly.

Human beings are strange creatures that tend to neglect details. In a perfect world, we would have all the four platforms to run our tests on. In the real world, however, most of the software companies try to skip some of the platforms listed here, or the concept of testing as a whole. As a developer who bought this book, you've already made the right decision: more tests and less debugging.

Now, again, it is up to you. Are you going to strive for perfection, stick to everything that you've learned so far, and let your code benefit from that knowledge?

JUNIT best practice: continuous regression testing

Most tests are written for the here and now. You write a new feature, and you write a new test. You see whether the feature plays well with others and whether the users like it. If everyone is happy, you can lock the feature and move to the next item on your list. Most software is written in a progressive fashion: You first add one feature and then another.

Most often, each new feature is built on a path paved by existing features. If an existing method can service a new feature, you reuse the method and save the cost of writing a new one. The process is never quite that easy, of course. Sometimes you need to change an existing method to make it work with a new feature. In this case, you need to confirm that the old features still work with the amended method.

A strong benefit of JUnit is that it makes test cases easy to automate. When a change is made to a method, you can run the test for that method. If that test passes, you can run the rest. If any test fails, you can change the code (or the tests) until all tests pass again.

Using old tests to guard against new changes is a form of regression testing. Any kind of test can be used as a regression test, but running unit tests after every change is your first, best line of defense.

The best way to ensure that regression testing takes place is to automate your test suites.

Chapter 7 discusses test granularity and introduces *stubs*: programs that simulate the behavior of software components that a module under test depends on.

6.7 Summary

This chapter has covered the following:

- Techniques in unit testing that take care of the quality of the tests.
- The concept of code coverage and tools that inspect it: you can do it from IntelliJ IDEA or with JaCoCo.
- Designing improvements of the code to make it easily testable.
- Investigating Test Driven Development (TDD) as a software development process focused on quality.
- Investigating Behavior Driven Development (BDD) as another software development

process focused on quality.

- Introducing mutation testing, a technique to evaluate the quality of existing software tests by modifying programs in small ways.
- Examining how the testing in the development cycle using the continuous integration principle can look like: starting with the coding (conventional development cycle) or starting with testing (Test Driven Development).

7

Coarse-grained testing with stubs

This chapter covers

- Testing with stubs
- Using an embedded server in place of a real webserver
- Implementing unit tests of an HTTP connection with stubs

And yet it moves.

—Galileo Galilei

As you develop your applications, you will find that the code you want to test depends on other classes, which themselves depend on other classes, which then depend on the environment (figure 7.1). You might be developing an application that uses Hibernate to access a database, a Java EE application (one that relies on a Java EE container for security, persistence, and other services), an application that accesses a file system, or an application that connects to some resource by using HTTP or another protocol.

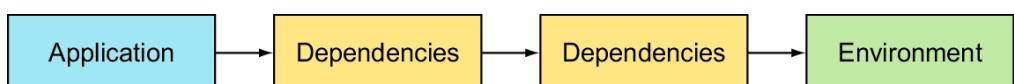


Figure 7.1 The application depends on other classes, which in turn depend on other classes, which then depend on the environment.

Starting in this chapter, we will look at using JUnit 5 to test an application that depends on external resources. We will include HTTP servers, database servers, and physical devices.

For applications that depend on a specific run-time environment, writing unit tests is a challenge. Your tests need to be stable, and when they run repeatedly, they need to yield the same results. You need a way to control the environment in which the tests run. One solution is to set up the real required environment as part of the tests and run the tests from within that environment. In some cases, this approach is practical and brings real benefits. (See chapter 9, which discusses in-container testing.) It works well only if you can set up the real environment on your development and build platforms, however, which is not always feasible.

If your application uses HTTP to connect to a web server provided by another company, for example, you usually will not have that server application available in your development environment. Therefore, you need a way to simulate that server so that you can still write and run tests for your code.

Alternatively, suppose that you are working with other developers on a project. What if you want to test your part of the application, but the other part is not ready? One solution is to simulate the missing part by replacing it with a fake that behaves in a similar way.

There are two strategies for providing these fake objects: stubbing and using mock objects.

When you write stubs, you provide a predetermined behavior right from the beginning. The stubbed code is written outside the test, and it will always have a fixed behavior, no matter how many times or where you will use the stub;— its methods will usually return hard-coded values. The pattern of testing with a stub is: *initialize stub > execute test > verify assertions*.

A mock object does not have a predetermined behavior. When running a test, you are setting the expectations on the mock before effectively using it. You may run different tests, and you may reinitialize a mock and set different expectations on it. The pattern of testing with a mock object is: *initialize mock > set expectations > execute test > verify assertions*.

This chapter is dedicated to stubbing; chapter 8 covers mock objects.

7.1Introducing stubs

Stubs are mechanisms for faking the behavior of real code or code that is not ready yet. In other words, stubs allow you to test a portion of a system when the other part is not available. Stubs usually do not change the code you are testing; instead, they adapt to provide seamless integration.

DEFINITION A *stub* is a piece of code that is inserted at run time in place of the real code to isolate the caller from the real implementation. The intent is to replace a complex behavior with a simpler one that allows independent testing of some part of the real code.

Here are some examples of when you might use stubs:

- You cannot modify an existing system because it is too complex and fragile.
- You are depending on an environment that you cannot control.
- You are replacing a full-blown external system such as a file system, a connection to a server, or a database.

- You are performing coarse-grained testing, such as integration testing between different subsystems.

Using stubs is not recommended in situations such as these:

- You need fine-grained tests to provide precise messages that underline the cause of the failure.
- You would like to test a small portion of the code in isolation.

In these situations, you should use mock objects (discussed in chapter 8).

Stubs usually provide very good confidence in the tested system. With stubs, you are not modifying the objects under test, and what you are testing is the same as what will execute in production. An automated build or a developer usually executes tests involving stubs in their running environment, providing additional confidence.

On the downside, stubs are usually hard to write, especially when the system to fake is complex. The stub needs to implement, in a simplified and short way, the same logic as the code it is replacing, which is difficult to get right for complex logic. Here are some cons of stubbing:

- Stubs are often complex to write and need debugging themselves.
- Stubs can be difficult to maintain because they are complex.
- A stub does not lend itself well to fine-grained unit testing.
- Each situation requires a different stubbing strategy.

In general, stubs are better adapted than mocks to replacing coarse-grained portions of code.

7.2 Stubbing an HTTP connection

To demonstrate what stubs can do, we'll explain how the engineers at Tested Data Systems Inc. build stubs for an application that opens an HTTP connection to a URL and reads its content. Figure 7.1 shows the sample application (limited to a `WebClient.getContent` method) opening an HTTP connection to a remote web resource. The remote web resource is a servlet, which generates an HTML response. The web resource in figure 7.2 is what we call the "real code" in the stub definition.

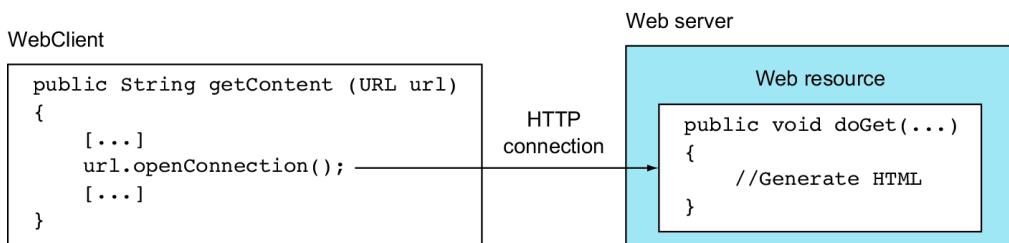


Figure 7.2 The sample application opens an HTTP connection to a remote web resource. The web resource is the real code in the stub definition.

The goal of the engineers at Tested Data Systems Inc. is to unit-test the `getContent` method by stubbing the remote web resource, as demonstrated in figure 7.3. The remote web resource is not yet available, and the engineers need to progress with their part of the work in the absence of that resource. They replace the servlet web resource with the stub—a simple HTML page that returns whatever they need for the `TestWebClient` test case. This approach allows them to test the `getContent` method independently of the implementation of the web resource (which in turn could call several other objects down the execution chain, possibly down to a database).

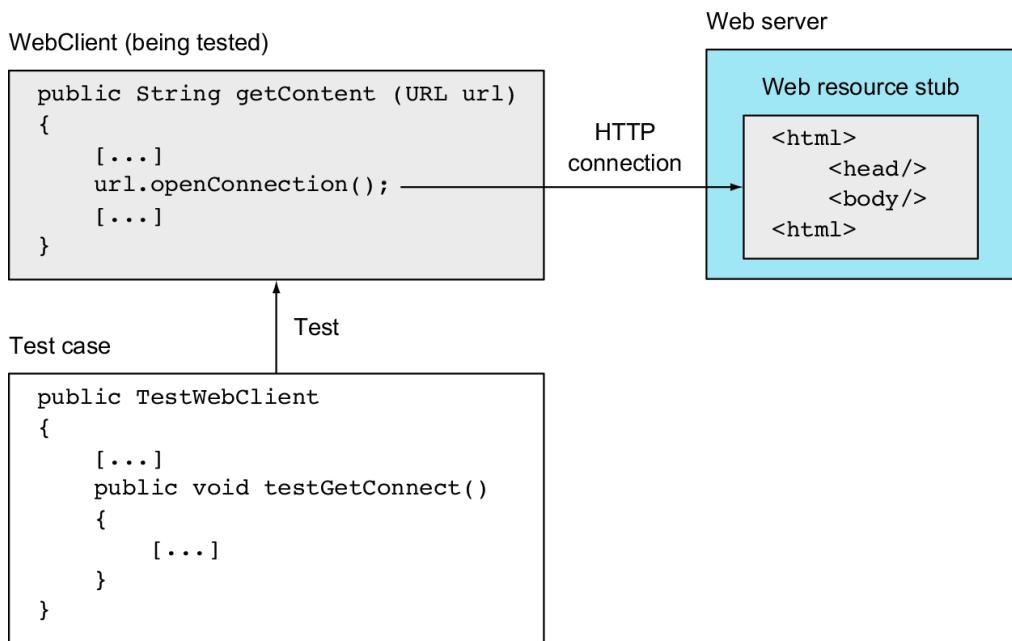


Figure 7.3 Adding a test case and replacing the real web resource with a stub

The important point to notice with stubbing is that `getContent` is not modified to accept the stub. The change is transparent to the application under test. For stubbing to be possible, the target code needs to have a well-defined interface and to allow different implementations (a stub, in this case) to be plugged in. In figure 7.1, the interface is actually the public abstract class `java.net.URLConnection`, which cleanly isolates the implementation of the page from its caller.

Following is an example of a stub in action, using the simple HTTP connection example. Listing 7.1 demonstrates a code snippet opening an HTTP connection to a given URL and reading the content at that URL. The method is one part of a bigger application to be unit-tested.

Listing 7.1 Sample method opening an HTTP connection

```
[...]
import java.net.URL;
import java.net.HttpURLConnection;
import java.io.InputStream;
import java.io.IOException;

public class WebClient {
    public String getContent(URL url) {
        StringBuffer content = new StringBuffer();
        try {
            HttpURLConnection connection = (HttpURLConnection)
                url.openConnection(); #A
            connection.setDoInput(true); #A
            InputStream is = connection.getInputStream(); #B
            byte[] buffer = new byte[2048]; #B
            int count; #B
            while (-1 != (count = is.read(buffer))) { #B
                content.append(new String(buffer, 0, count)); #B
            } #B
        } catch (IOException e) { #C
            throw new RuntimeException(e);
        }
        return content.toString();
    }
}
```

In this listing:

- We start by opening an HTTP connection using the `HttpURLConnection` class (#A).
- We read the stream content until there is nothing more to read (#B).
- If an error occurs, we pack it into a `RuntimeException` and rethrow it (#C).

7.2.1 Choosing a stubbing solution

The example application has two possible scenarios: the remote web server (see figure 7.2) could be located outside the development platform (such as on a partner site), or it could be part of the platform where the application is to be deployed. In both cases, you need to introduce a server into the development platform to be able to unit-test the `WebClient` class. One relatively easy solution would be to install an Apache test server and drop some test web pages in its document root. This stubbing solution is typical and widely used, but it has several drawbacks, as shown in table 7.1.

Table 7.1 Drawbacks of the chosen stubbing solution

Drawback	Explanation
Reliance on the environment	We need to be sure that the full environment is up and running before the test starts. If the webserver is down, and we execute the test, it will fail, and we will spend time determining why it is failing.

	<p>We will discover that the code is working fine and that the problem was only a setup issue generating a false failure.</p> <p>When we are unit testing, it is important to be able to control as much as possible of the environment in which the tests execute, such that test results are reproducible.</p>
Separated test logic	<p>The test logic is scattered in two locations: in the JUnit 5 test case and on the test web page.</p> <p>We need to keep both types of resources in sync for the tests to succeed.</p>
Difficult tests to automate	<p>Automating the execution of the tests is difficult because it involves deploying the web pages on the webserver, starting the webserver, and running the unit tests.</p>

Fortunately, an easier solution exists: using an embedded web server. Because the testing is in Java, the easiest solution is to use a Java web server that can be embedded in the test case. You can use the free, open-source Jetty server for this purpose. The developers at Tested Data Systems Inc. will use Jetty to set up their stubs. (For more information about Jetty, visit <https://www.eclipse.org/jetty/>.)

7.2.2 Using Jetty as an embedded server

We use Jetty because it is fast (important when running tests), it is lightweight, and the test cases can control it programmatically. Additionally, Jetty is a very good web, servlet, and JSP container that you can use in production.

Using Jetty allows us to eliminate the drawbacks outlined above: the JUnit 5 test case starts the server, we write the tests in Java in one location, and automating the test suite becomes a nonissue. Thanks to Jetty modularity, the real task facing the developers at Tested Data Systems Inc. is stubbing the Jetty handlers, not the whole server from the ground up.

To understand how Jetty works, implement and examine an example of setting up and controlling it from the tests. Listing 7.2 demonstrates how to start Jetty from Java and how to define a document root (/) from which to start serving files.

Listing 7.2 Starting Jetty in embedded mode: JettySample class

```
[...]
import org.mortbay.jetty.Server;
import org.mortbay.jetty.handler.ResourceHandler;
import org.mortbay.jetty.servlet.Context;

public class JettySample {
    public static void main(String[] args) throws Exception {
        Server server = new Server(8080); #A

        Context root = new Context(server, "/");
        root.setResourceBase("./pom.xml");
        root.setHandler(new ResourceHandler()); #B

        server.start(); #C
    }
}
```

```
}
```

In this listing:

- We start by creating the Jetty Server object (#A) and specifying in the constructor which port to listen to for HTTP requests (port 8080).
- Next, we create a Context object (#B) that processes the HTTP requests and passes them to various handlers. We map the context to the already-created server instance, and to the root (/) URL. The `setResourceBase` method sets the document root from which to serve resources. On the next line, we attach a `ResourceHandler` handler to the root to serve files from the file system.
- Finally, we start the server (#C).

If you start the program from listing 7.2 and navigate your browser to `http://localhost:8080`, you should be able to see the content of the `pom.xml` file (figure 7.4).

You get a similar effect by changing the code to set the base as `root.setResourceBase("."),` restarting the server, and then navigating to `http://localhost:8080/pom.xml`.

```
--<!--  
  
Licensed to the Apache Software Foundation (ASF) under one or more  
contributor license agreements. See the NOTICE file distributed with  
this work for additional information regarding copyright ownership.  
The ASF licenses this file to you under the Apache License, Version  
2.0 (the "License"); you may not use this file except in compliance  
with the License. You may obtain a copy of the License at  
  
http://www.apache.org/licenses/LICENSE-2.0 Unless required by  
applicable law or agreed to in writing, software distributed under the  
License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR  
CONDITIONS OF ANY KIND, either express or implied. See the License for  
the specific language governing permissions and limitations under the  
License.  
  
-->  
-<project xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">  
  <modelVersion>4.0.0</modelVersion>  
  <groupId>com.manning.junitbook</groupId>  
  <artifactId>ch07-stubs</artifactId>  
  <version>1.0-SNAPSHOT</version>  
  <name>ch07-stub</name>  
-<properties>  
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>  
  <maven.compiler.source>1.8</maven.compiler.source>
```

Figure 7.4 Testing the JettySample in a browser. It displays the content of the `pom.xml` file to demonstrate how the Jetty webserver works

Figure 7.4 demonstrates the results of running the code in listing 7.2 after opening a browser on <http://localhost:8080>.

Now that you have seen how to run Jetty as an embedded server, the next section shows you how to stub the server resources.

7.3 Stubbing the web server resources

This section focuses on the HTTP connection unit test. The developers at Tested Data Systems Inc. will write tests to verify they can call a valid URL and get its content. These tests are the first steps in checking the functionality of web applications that are interacting with external customers.

7.3.1 Setting up the first stub test

To verify that the `WebClient` works with a valid URL, we need to start the Jetty server before the test, which we can implement in a test case `setUp` method. We can also stop the server in a `tearDown` method. Listing 7.3 shows the code.

Listing 7.3 Skeleton of the first test to verify that the WebClient works with a valid URL

```
[...]
import java.net.URL;
import org.junit.jupiter.api.*;

public class TestWebClientSkeleton {

    @BeforeAll
    public static void setUp() {
        // Start Jetty and configure it to return "It works" when
        // the http://localhost:8080/testGetContentOk URL is
        // called.
    }

    @AfterAll
    public static void tearDown() {
        // Stop Jetty.
    }

    @Test
    public void testGetContentOk() throws MalformedURLException {
        WebClient client = new WebClient();
        String workingContent = client.getContent(new URL(
            "http://localhost:8080/testGetContentOk"));

        assertEquals ("It works", workingContent);
    }
}
```

To implement the `@BeforeAll` and `@AfterAll` methods, we have are two options. We can prepare a static page containing the text "It works", which we put in the document root (controlled by the call to `root.setResourceBase(String)` in listing 7.2). Alternatively, we can configure Jetty to use a custom handler that returns the string "It works" instead of

getting it from a file. This technique is much more powerful because it lets us unit-test the case when the remote HTTP server returns an error code to the WebClient client application.

CREATING A JETTY HANDLER

Listing 7.4 creates a Jetty Handler that returns the string "It works".

Listing 7.4 Create a Jetty Handler that returns "It works" when called

```
private static class TestGetContentOkHandler extends AbstractHandler { #A  
    @Override #A  
    public void handle(String target, HttpServletRequest request,  
                      HttpServletResponse response, int dispatch) throws IOException { #A  
  
        OutputStream out = response.getOutputStream(); #B  
        ByteArrayISO8859Writer writer = new ByteArrayISO8859Writer(); #B  
        writer.write("It works"); #C  
        writer.flush(); #C  
        response.setIntHeader(HttpHeaders.CONTENT_LENGTH, writer.size()); #D  
        writer.writeTo(out); #D  
        out.flush(); #D  
    } #D
```

In this listing:

- The class creates a handler (#A) by extending the Jetty `AbstractHandler` class and implementing a single method: `handle`.
- Jetty calls the `handle` method to forward an incoming request to our handler. After that, we use the Jetty `ByteArrayISO8859Writer` class (#B) to send back the string "It works", which we write in the HTTP response (#C).
- The last step is setting the response content length to the length of the string written to the output stream (required by Jetty) and then send the response (#D).
- Now that this handler is written, we can tell Jetty to use it by calling `context.setHandler(new TestGetContentOkHandler())`.

WRITING THE TEST CLASS

We are almost ready to run the test that is the basis of verifying the functionality of the web applications interacting with the external customers of Tested Data Systems Inc., as shown in listing 7.5.

Listing 7.5 Putting it all together

```
[...]  
import java.net.URL;  
[...]  
  
public class TestWebClient {  
  
    @BeforeAll  
    public static void setUp() throws Exception {  
        Server server = new Server(8080);  
    }
```

```

Context contentOkContext = new Context(server, "/testGetContentOk");
contentOkContext.setHandler(new TestGetContentOkHandler());

server.setStopAtShutDown(true);
server.start();

}

@AfterAll
public static void tearDown()
{
    // Empty
}

@Test
public void testGetContentOk() throws Exception {
    WebClient client = new WebClient();
    String workingContent = client.getContent(new URL(
        "http://localhost:8080/testGetContentOk"));
    assertEquals("It works", workingContent);
}

private static class TestGetContentOkHandler extends AbstractHandler {
    //Listing 7.4 here.
}

}

```

The test class has become quite simple. The `@BeforeAll` `setUp` method constructs the `Server` object the same way as in listing 7.2. Then we have our `@Test` method. We leave our `@AfterAll` method empty intentionally because we programmed the server to stop at shutdown; the server instance is explicitly stopped when the JVM is shut down.

If you run the test, you see the result in figure 7.5. The test passes.

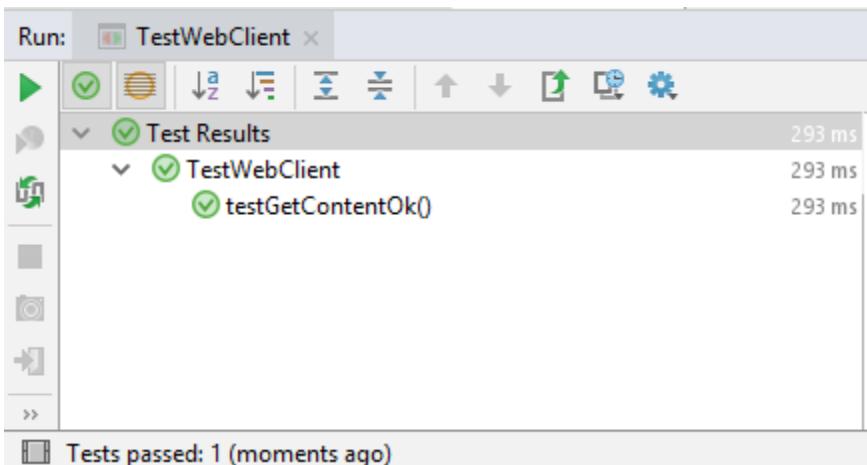


Figure 7.5 Result of the first working test using a Jetty stub. JUnit 5 starts the server before the first test, and the server shuts itself down after the last test.

7.3.2 Reviewing the first stub test

You have been able to fully unit-test the `getContent` method in isolation by stubbing the web resource. What have you really tested? What kind of test have you achieved? You have done something quite powerful: unit-tested the method and at the same time executed an integration test. In addition, you have tested not only the code logic but also the connection part that is outside the code (through the Java `HttpURLConnection` class).

The drawback of this approach is that it is complex. At Tested Data Systems Inc., it can take a Jetty novice half a day to learn enough about Jetty to set it up correctly. In some instances, the novice will have to debug stubs to get them to work properly. Keep in mind that the stub must stay simple, not become a full-fledged application that requires tests and maintenance. If you spend too much time debugging the stubs, a different solution may be called for.

In these examples, we need a web server—but another example and stub will be different and need a different setup. Experience helps, but different cases usually require different stubbing solutions.

The example tests the web application developed at Tested Data Systems Inc., which is nice because it allows us to unit-test the code and to perform some integration tests at the same time. This functionality, however, comes at the cost of complexity. More solutions that are lightweight focus on unit-testing the code without performing integration tests. The rationale is that although we need integration tests, they could run in a separate test suite or as part of functional tests.

The next section looks at another solution that still qualifies as stubbing and is simpler in the sense that it does not require stubbing a whole web server. This solution brings us one step closer to the mock-object strategy, which is described in chapter 8.

7.4 Stubbing the connection

So far, we have stubbed the web server resources. Next, we will stub the HTTP connection instead. Doing so will prevent us from effectively testing the connection, which is fine because that is not the real goal at this point. We want to test the code in isolation. Functional or integration tests will test the connection at a later stage.

When it comes to stubbing the connection without changing the code, we benefit from the Java `URL` and `HttpURLConnection` classes, which let us plugin custom protocol handlers to process any kind of communication protocol. We can have any call to the `HttpURLConnection` class redirected to our own class, which will return whatever we need for the test.

7.4.1 Producing a custom URL protocol handler

To implement a custom URL protocol handler, we need to call the `URL` method `setURLStreamHandlerFactory` and pass it a custom `URLStreamHandlerFactory`. The engineers at Tested Data Systems Inc. are using this approach to create their stub implementation of the URL stream handler. Whenever the `URL` `openConnection` method is called, the `URLStreamHandlerFactory` class is called to return a `URLStreamHandler`. Listing 7.6 shows the code that performs this action. The idea is to call the `URL` static method `setURLStreamHandlerFactory` in the JUnit 5 `setUp` method.

Listing 7.6 Providing custom stream handler classes for testing

```
[...]
import java.net.URL;
import java.net.URLStreamHandlerFactory;
import java.net.URLStreamHandler;
import java.net.URLConnection;
import java.io.IOException;

public class TestWebClient1 {

    @BeforeAll
    public static void setUp() {
        URL.setURLStreamHandlerFactory(new StubStreamHandlerFactory());    #A
    }

    private static class StubStreamHandlerFactory implements           #B
        URLStreamHandlerFactory {                                         #B
        #B
        public URLStreamHandler createURLStreamHandler(String protocol) { #B
            return new StubHttpURLStreamHandler();                         #B
        }
    }                                                               #B
    private static class StubHttpURLStreamHandler                   #C
}
```

```

        extends URLStreamHandler { #C
protected URLConnection openConnection(URL url) #C
    throws IOException { #C
        return new StubHttpURLConnection(url); #C
    } #C
} #C
@Test #C
public void testGetContentOk() throws MalformedURLException { #C
    WebClient client = new WebClient(); #C
    String workingContent = client.getContent( #C
        new URL("http://localhost")); #C
    assertEquals("It works", workingContent); #C
}
}

```

In this listing:

- We start by calling `setURLStreamHandlerFactory` (#A) with our first stub class, `StubStreamHandlerFactory`.
- We use several (inner) classes (#B and #C) to use the `StubHttpURLConnection` class.
- In `StubStreamHandlerFactory`, we override the `createURLStreamHandler` method (#B), in which we return a new instance of our second private stub class, `StubHttpURLStreamHandler`.
- In `StubHttpURLStreamHandler`, we override one method, `openConnection`, to open a connection to the given URL (#C).

Note that we have not written the `StubHttpURLConnection` class yet. That class is the topic of the next section.

7.4.2 Creating a JDK `HttpURLConnection` stub

The last step is creating a stub implementation of the `HttpURLConnection` class so that we can return any value we want for the test. Listing 7.7 shows a simple implementation that returns the string "It works" as a stream to the caller.

Listing 7.7 Stubbed `HttpURLConnection` class

```

[...]
import java.net.HttpURLConnection;
import java.net.ProtocolException;
import java.net.URL;
import java.io.InputStream;
import java.io.IOException;
import java.io.ByteArrayInputStream;

public class StubHttpURLConnection extends HttpURLConnection {
    private boolean isInput = true;
    protected StubHttpURLConnection(URL url) {
        super(url);
    }
    public InputStream getInputStream() throws IOException {
        if (!isInput) { #A
            ...
        }
    }
}

```

```

        throw new ProtocolException(
            "Cannot read from URLConnection"
            + " if doInput=false (call setDoInput(true))");
    }
    ByteArrayInputStream readStream = new ByteArrayInputStream(
        new String("It works").getBytes());
    return readStream;
}
public void connect() throws IOException {}
public void disconnect() {}
public boolean usingProxy() {
    return false;
}
}

```

In this listing:

- `HttpURLConnection` is an abstract public class that does not implement an interface, so we extend it and override the methods wanted by the stub.
- In this stub, we provide an implementation for the `getInputStream` method, as it is the only method used by our code under test.
- Should the code to be tested use more APIs from `HttpURLConnection`, we would need to stub these additional methods. This part is where the code would become more complex; we would need to reproduce the same behavior as the real `HttpURLConnection`.
- We test whether `setDoInput(false)` has been called in the code under test (#A). The `isInput` flag will tell if we use the URL connection for input. Then, a call to the `getInputStream` method returns a `ProtocolException` (the behavior of `HttpURLConnection`). Fortunately, in most cases, we need to stub only a few methods, not the whole API.

7.4.3 Running the test

We will test the `getContent` method by stubbing the connection to the remote resource, using the `TestWebClient1` test, which uses the `StubHttpURLConnection`. Figure 7.6 shows the result of the test.

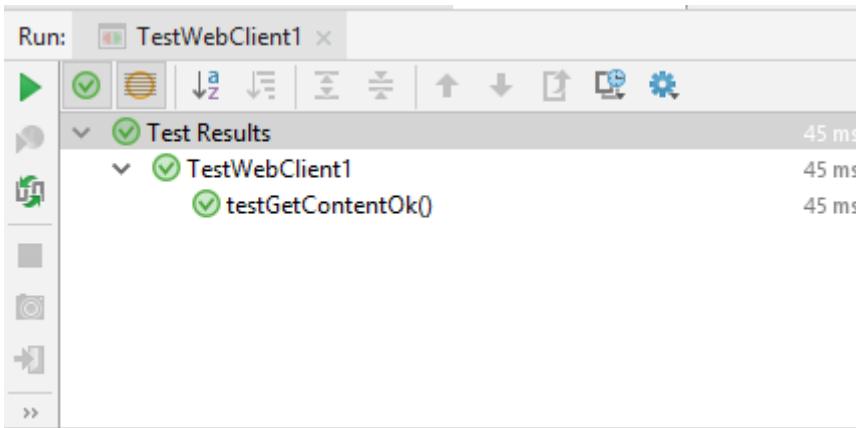


Figure 7.6 Result of executing `TestWebClient1` (which uses the `StubHttpURLConnection`).

As you can see, it is much easier to stub the connection than to stub the web resource. This approach does not provide the same level of testing (does not perform integration tests), but it enables the programmer to write a focused unit test for the `WebClient` logic more easily.

Chapter 8 demonstrates a technique called mock objects that allows fine-grained unit testing, which is completely generic, and (best of all) forces you to write good code. Although stubs are very useful in some cases, some people consider them to be vestiges of the past, when the consensus was that tests should be separate activities and should not modify existing code. The new mock-objects strategy not only allows modification of code but also favors it. Using mock objects is more than a unit-testing strategy: it is a completely new way of writing code.

7.5 Summary

This chapter has covered the following:

- Analyzing when to use stubs: when you cannot modify an existing complex or fragile system; when you are depending on an uncontrollable environment; to replace a full-blown external system; for coarse-grained testing.
- Analyzing when not to use stubs: when you need fine-grained tests to provide precise messages that underline the exact cause of a failure; when you would like to test a small portion of the code in isolation.
- Demonstrating how using a stub has helped us unit-test code accessing a remote web server using the Java `HttpURLConnection` API.
- In particular, implementing a stub for a remote web server by using the open-source Jetty server. The embeddable nature of Jetty lets us concentrate on stubbing only the Jetty HTTP request handler, instead of having to stub the whole container.
- Implementing a more lightweight solution by stubbing the Java `HttpURLConnection`

class.

8

Testing with mock objects

This chapter covers

- Introducing and demonstrating mock objects
- Executing different refactorings with the help of mock objects
- Practicing on an HTTP connection sample application
- Working with and comparing the EasyMock, JMock and Mockito frameworks

Programming today is a race between software engineers striving to build bigger and better idiot-proof programs, and the Universe trying to produce bigger and better idiots. So far, the Universe is winning.

- Rich Cook

Unit-testing each method in isolation from the other methods or the environment is certainly a nice goal. How do you perform this task? You saw in chapter 7 that the stubbing technique lets you unit-test portions of code by isolating them from the environment (such as by stubbing a web server, the file system, a database, and so on). What about fine-grained isolation, such as being able to isolate a method call to another class? Can you achieve this without deploying huge amounts of energy that would negate the benefits of having tests?

The answer is yes. The technique is called *mock objects*. Tim Mackinnon, Steve Freeman, and Philip Craig first presented the mock-objects concept at XP2000. The strategy allows you to unit-test at the finest possible level. You will develop method by method, after having provided unit tests for each method.

8.1 Introducing mock objects

Testing in isolation offers strong benefits, such as the ability to test code that has not yet been written (as long as you at least have an interface to work with). In addition, testing in isolation helps teams unit-test one part of the code without waiting for all the other parts.

The biggest advantage is the ability to write focused tests that test a single method, with no side effects resulting from other objects being called from the method under test. Small is beautiful. Writing small, focused tests is a tremendous help; small tests are easy to understand and do not break when other parts of the code are changed. Remember that one of the benefits of having a suite of unit tests is the courage it gives you to refactor mercilessly; the unit tests act as a safeguard against regression. If you have large tests, and your refactoring introduces a bug, several tests will fail. That result will tell you that there is a bug somewhere, but you will not know where. With fine-grained tests, potentially fewer tests are affected, and they provide precise messages that pinpoint the cause of the failure.

Mock objects (*mocks* for short) are perfectly suited for testing a portion of code logic in isolation from the rest of the code. Mocks replace the objects with which your methods under test collaborate, thus offering a layer of isolation. In that sense, they are similar to stubs. The similarity ends there, however, because mocks do not implement any logic: they are empty shells that provide methods that let the tests control the behavior of all the business methods of the faked class.

A stub is created with an already predetermined behavior, even a very simple one, and this behavior cannot be changed when running different tests.

A mock does not have a predetermined behavior. When running a test, you are setting the expectations on the mock before effectively using it. You may run different tests, and you may reinitialize a mock and set different expectations on it. The pattern for testing with a mock is *initialize mock > set expectations > execute test > verify assertions*.

We discuss when to use mock objects in section 8.6 at the end of this chapter, after we show them in action on some examples.

8.2 Unit testing with mock objects

In this section, we present an application and tests by using mock objects. Imagine a very simple use case in which you want to be able to make a bank transfer from one account to another (figure 8.1 and listings 8.1 and 8.2).

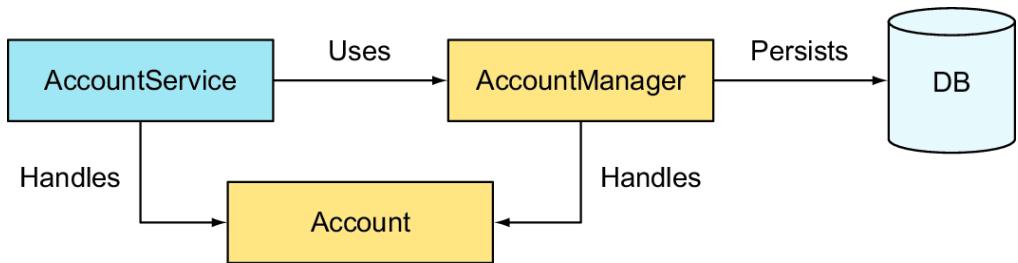


Figure 8.1 In this simple bank-account example, you use a mock object to test an account transfer method.

The `AccountService` class offers services related to `Account` objects and uses the `AccountManager` to persist data to the database (by using JDBC, for example). The service that is of interest for us materializes itself via the `AccountService.transfer` method, which makes the transfer. Without mocks, testing the `AccountService.transfer` behavior would imply setting up a database, presetting it with test data, deploying the code inside the container (J2EE application server, for example), and so forth. Although this process is required to ensure that the application works end to end, it is too much work when you want to unit-test only your code logic.

Tested Data Systems Inc. creates software projects for other companies. One of the projects under development requires managing accounts and money transfers. The engineers need to design some solutions for it. Listing 8.1 presents a very simple `Account` object with two properties: an account ID and a balance.

Listing 8.1 Account.java

```
[...]
public class Account {
    private String accountId;
    private long balance;

    public Account(String accountId, long initialBalance) {
        this.accountId = accountId;
        this.balance = initialBalance;
    }

    public void debit(long amount) {
        this.balance -= amount;
    }

    public void credit(long amount) {
        this.balance += amount;
    }

    public long getBalance() {
        return this.balance;
    }
}
```

As part of the solution, the `AccountManager` interface below manages the life cycle and persistence of `Account` objects (limited to finding accounts by ID and updating accounts).

```
public interface AccountManager {  
    Account findAccountForUser(String userId);  
    void updateAccount(Account account);  
}
```

Listing 8.2 shows the `transfer` method designed for transferring money between two accounts. It uses the previously defined `AccountManager` interface to find the debit and credit accounts by ID and to update them.

Listing 8.2 AccountService.java

```
[...]  
public class AccountService {  
    private AccountManager accountManager;  
  
    public void setAccountManager(AccountManager manager) {  
        this.accountManager = manager;  
    }  
  
    public void transfer(String senderId, String beneficiaryId, long amount) {  
        Account sender = accountManager.findAccountForUser(senderId);  
        Account beneficiary =  
            accountManager.findAccountForUser(beneficiaryId);  
  
        sender.debit(amount);  
        beneficiary.credit(amount);  
        this.accountManager.updateAccount(sender);  
        this.accountManager.updateAccount(beneficiary);  
    }  
}
```

We want to be able to unit-test the `AccountService.transfer` behavior. For that purpose, until the implementation of the `AccountManager` interface is ready, we use a mock implementation of the `AccountManager` interface (listing 8.3) because the `transfer` method is using this interface, and we need to test it in isolation.

Listing 8.3 MockAccountManager.java

```
[...]  
import java.util.HashMap;  
  
public class MockAccountManager implements AccountManager {  
    private Map<String, Account> accounts = new HashMap<String, Account>();  
  
    public void addAccount(String userId, Account account) {  
        this.accounts.put(userId, account);  
    }  
  
    public Account findAccountForUser(String userId) {  
        return this.accounts.get(userId);  
    }  
}
```

```
public void updateAccount(Account account) {  
    // do nothing  
}  
}
```

In this listing:

- The `addAccount` method uses an instance variable to hold the values to return (#A). Because we have several account objects that we want to be able to return, we store the `Account` objects to return in a `HashMap`. This step makes the mock generic and able to support different test cases. One test could set up the mock with one account, another test could set it up with two accounts or more, and so forth.
- We implement a method to retrieve the account from the `accounts` map (#B). We can retrieve only accounts that were added earlier.
- The `updateAccount` method updates an account but does not return any value (#C). Thus, we do nothing. When it is called by the `transfer` method, it does nothing, as though the account had been updated correctly.

JUNIT best practices: do not write business logic in mock objects

The single most important point to consider when writing a mock is that it should not have any business logic: a mock must be a dumb object that does only what the test tells it to do. In other words, it is purely driven by the tests. This characteristic is exactly the opposite of stubs, which contain all the logic (see chapter 7).

This point has two nice corollaries First, mock objects can be generated easily. Second, because mock objects are empty shells, they are too simple to break and do not need testing themselves.

Now you are ready to write a unit test for `AccountService.transfer`. Listing 8.4 shows a typical test that uses a mock.

Listing 8.4 Testing transfer with MockAccountManager

```
[...]  
public class TestAccountService {  
  
    @Test  
    public void testTransferOk() {  
        MockAccountManager mockAccountManager = new MockAccountManager(); #A  
        Account senderAccount = new Account("1", 200); #A  
        Account beneficiaryAccount = new Account("2", 100); #A  
        mockAccountManager.addAccount("1", senderAccount); #A  
        mockAccountManager.addAccount("2", beneficiaryAccount); #A  
        AccountService accountService = new AccountService(); #A  
        accountService.setAccountManager(mockAccountManager); #A  
        accountService.transfer("1", "2", 50); #B  
  
        assertEquals(150, senderAccount.getBalance()); #C  
        assertEquals(150, beneficiaryAccount.getBalance()); #C  
    }  
}
```

As usual, a test has three steps: setup (#A), execution (#B), and the verification of the result (#C). During the test setup, we create the `MockAccountManager` object and define what it should return when it's called for the two accounts we manipulate (the sender and beneficiary accounts). Practically, setting the expectations of the `mockAccountManager` object by adding two accounts to it transforms it into our own defined mock. As stated earlier, one of the characteristics of a mock is that when you run a test, you are setting the expectations on the mock before effectively using it.

You have succeeded in testing the `AccountService` code in isolation from the other domain object, `AccountManager`, which in this case did not exist but could have been implemented with JDBC in real life.

JUNIT best practices: test only what could possibly break

You may have noticed that we did not mock the `Account` class. The reason is that this data-access object class does not need to be mocked; it does not depend on the environment, and it is very simple. The other tests use the `Account` object, so they test it indirectly. If this class failed to operate correctly, the tests that rely on `Account` would fail and alert to the problem.

At this point in the chapter, you should have a reasonably good understanding of mocks. The next section demonstrates that writing unit tests with mocks leads to refactoring the code under test—and that this process is a good thing.

8.3 Refactoring with mock objects

Some people used to say that unit tests should be fully transparent to the code under test and that run-time code should not be changed to simplify testing. *This is wrong!* Unit tests are first-class users of the run-time code and deserve the same consideration as any other user. If the code is too inflexible for the tests to use, we should correct the code.

Consider the following piece of code created by one of the engineers at Test It. This engineer is taking care of the implementation of the `AccountManager` class that we previously mocked until it is fully available.

Listing 8.5 The DefaultAccountManager class

```
[...]
import java.util.PropertyResourceBundle;
import java.util.ResourceBundle;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
[...]
public class DefaultAccountManager implements AccountManager {
    private static final Log logger =                               #A
        LogFactory.getLog(AccountManager.class);                  #A

    public Account findAccountForUser(String userId) {
        logger.debug("Getting account for user [" + userId + "]);
```

```

    ResourceBundle bundle = #B
        PropertyResourceBundle.getBundle("technical");
    String sql = bundle.getString("FIND_ACCOUNT_FOR_USER");
    // Some code logic to load a user account using JDBC
    [...]
}
[...]
}

```

In this listing:

- We create a `Log` object (#A).
- We retrieve an SQL command (#B).

Does the code look fine to you? We can see two issues, both of which relate to code flexibility and the ability to resist change. The first problem is that it is not possible to decide to use a different `Log` object, as it is created inside the class. For testing, for example, you probably would like to use a `Log` that does nothing, but you cannot do so.

As a rule, a class like this one should be able to use whatever `Log` it is given.

The goal of this class is not to create loggers, but to perform some JDBC logic. The same goal applies to `PropertyResourceBundle`. It may sound OK right now, but what happens if you decide to use XML to store the configuration? Again, it should not be the goal of this class to decide what implementation to use.

An effective design strategy is to pass to an object any other object that is outside its immediate business logic. Ultimately, as you move up in the calling layers, the decision to use a given logger or configuration should be pushed to the top level. This strategy provides the best possible code flexibility and ability to cope with changes. And as we all know, change is the only constant.

Taking these issues into consideration, the Tested Data Systems engineer who created the code will need to refactor it.

8.3.1 Refactoring example

Refactoring all code so that domain objects are passed around can be time-consuming. You may not be ready to refactor the whole application just to be able to write a unit test. Fortunately, an easy refactoring technique lets you keep the same interface for the code but allows it to be passed domain objects that it should not create. As proof, the following listing shows what the refactored `DefaultAccountManager` class could look like (modifications in bold).

Listing 8.6 Refactoring DefaultAccountManager for testing

```

public class DefaultAccountManager implements AccountManager {
    private Log logger; #A
    private Configuration configuration; #A

    public DefaultAccountManager() {
        this(LogFactory.getLog(DefaultAccountManager.class),
            new DefaultConfiguration("technical"));
    }
}

```

```

    }
    public DefaultAccountManager(Log logger, Configuration configuration) {
        this.logger = logger;
        this.configuration = configuration;
    }

    public Account findAccountForUser(String userId) {
        this.logger.debug("Getting account for user [" + userId + "]");
        this.configuration.getSQL("FIND_ACCOUNT_FOR_USER");
        // Some code logic to load a user account using JDBC
        [...]
    }
[...]
}

```

At #A, we swap the `PropertyResourceBundle` class from the previous listing in favor of a new `Configuration` interface. This swap makes the code more flexible because it introduces an interface (which will be easy to mock), and the implementation of the `Configuration` interface can be anything we want (including using resource bundles). The design is better now because we can use and reuse the `DefaultAccountManager` class with any implementation of the `Log` and `Configuration` interfaces (if we use the constructor that takes two parameters). The class can be controlled from the outside (by its caller). Meanwhile, we have not broken the existing interface because we have added only a new constructor. We kept the original default constructor that still initializes the `logger` and `configuration` field members with default implementations.

8.3.2 Refactoring considerations

With this refactoring, we have provided a trap door for controlling the domain objects from the tests. We retain backward compatibility and pave an easy refactoring path for the future. Calling classes can start using the new constructor at their own pace.

Should you worry about introducing trap doors to make your code easier to test? Here's how Extreme Programming guru Ron Jeffries explains it:

My car has a diagnostic port and an oil dipstick. There is an inspection port on the side of my furnace and on the front of my oven. My pen cartridges are transparent so I can see if there is ink left. And if I find it useful to add a method to a class to enable me to test it, I do so. It happens once in a while, for example in classes with easy interfaces and complex inner function (probably starting to want an Extract Class).

I just give the class what I understand of what it wants, and keep an eye on it to see what it wants next.¹

¹ Ron Jeffries, on the TestDrivenDevelopment mailing list:
<http://groups.yahoo.com/group/testdrivendevelopment/message/3914>.

Design patterns in action: Inversion of Control (IoC)

Applying the IoC pattern to a class means removing the creation of all object instances for which this class is not directly responsible and passing any needed instances instead. The instances may be passed by a specific constructor or a setter, or as parameters of the methods that need them. It becomes the responsibility of the calling code to set these domain objects correctly on the class that is called.²

IoC makes unit testing a breeze. To prove the point, here's how easily we can write a test for the `findAccountByUser` method.

Listing 8.7 The testFindAccountByUser method

```
public void testFindAccountByUser() {  
    MockLog logger = new MockLog(); #A  
    MockConfiguration configuration = new MockConfiguration(); #B  
    configuration.setSQL("SELECT * [...]"); #B  
    DefaultAccountManager am = new DefaultAccountManager(logger, #C  
                                                       configuration);#C  
    Account account = am.findAccountForUser("1234");  
    // Perform asserts here  
    [...]  
}
```

In this listing:

- We use a mock logger that implements the `Log` interface but does nothing (#A).
- Next, we create a `MockConfiguration` instance (#B) and set it up to return a given SQL query when `Configuration.getSQL` is called.
- Finally, we create the instance of `DefaultAccountManager` (#C) that we will test, passing to it the `Log` and `Configuration` instances.

We have been able to completely control the logging and configuration behavior from outside the code to test, in the test code. As a result, the code is more flexible and allows for any logging and configuration implementation to be used. We will implement more of these code refactorings in this chapter and later ones. For now, the developer from Tested Data Systems has solved the issues we previously examined by taking this approach.

One last point to note is that if we write the test first, we will automatically design the code to be flexible. Flexibility is a key point in writing a unit test. If we test first, we will not incur the cost of refactoring the code for flexibility later.

² See Spring for a framework that implements the IoC pattern (<https://spring.io>).

8.4 Mocking an HTTP connection

To see how mock objects work in a practical example, go back to the application developed at Tested Data Systems that opens an HTTP connection to a remote server and reads the content of a page. In chapter 7, we tested that application using stubs, but the developers from Tested Data Systems decided to move to a mock-object approach to simulate the HTTP connection. In addition, they will implement mocks for classes that do not have a Java interface (namely, the `HttpURLConnection` class).

In the full scenario of the implementation, we start with an initial testing implementation, improve the implementation as we go, and modify the original code to make it more flexible. We also use mocks to test for error conditions.

As we dive in, we will keep improving both the test code and the sample application, exactly as we might if we were writing the unit tests for the same application. In the process, you will learn how to reach a simple and elegant testing solution while making the application code more flexible and capable of handling change.

Figure 8.2 introduces the sample HTTP application, which consists of a simple `WebClient.getContent` method performing an HTTP connection to a web resource executing on a web server. We want to be able to unit-test the `getContent` method in isolation from the web resource.

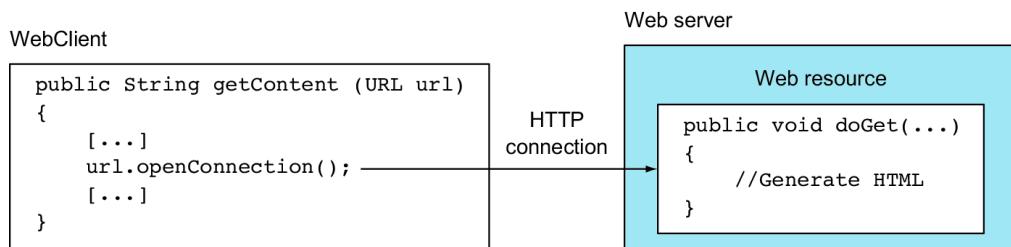


Figure 8.2 The sample HTTP application before the test is introduced

8.4.1 Defining the mock objects

Figure 8.3 illustrates a mock object. The `MockURL` class stands in for the real `URL` class, and all calls to the `URL` class in `getContent` are directed to the `MockURL` class. As you can see, the test is the controller: it creates and configures the behavior that the mock must have for this test, it (somehow) replaces the real `URL` class with the `MockURL` class, and it runs the test.

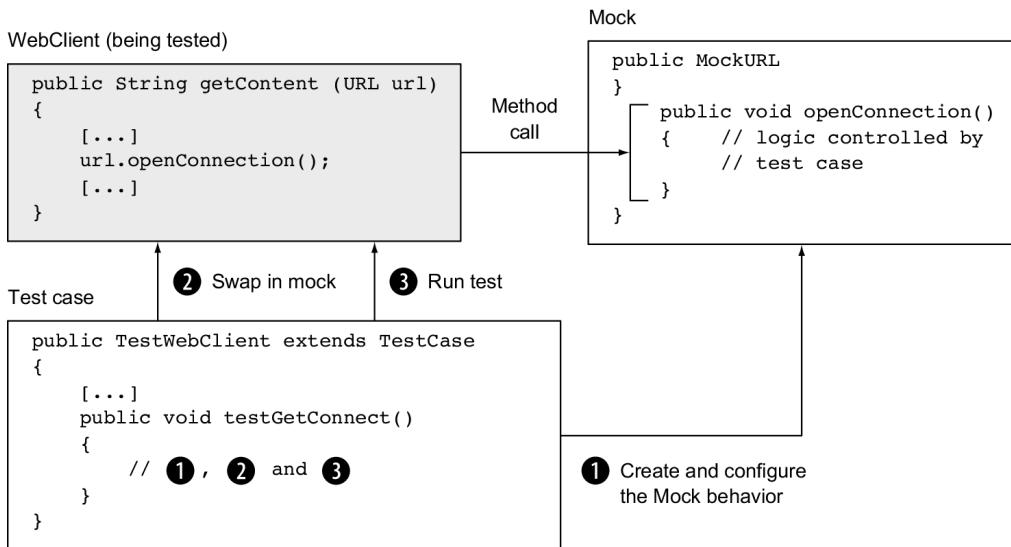


Figure 8.3 The steps involved in a test using mock objects: create and configure the Mock object behavior, swap it in, and run the test.

Figure 8.3 shows an interesting aspect of the mock-objects strategy: the ability to swap in the mock in the production code. The perceptive reader will have noticed that because the `URL` class is final, it is not possible to create a `MockURL` class that extends it.

In the coming sections, we demonstrate how this feat can be performed in a different way (by mocking at another level). In any case, when using the mock-objects strategy, swapping in the mock instead of the real class is the hard part unless we are using dependency injection. This point may be viewed as negative for mock objects because you usually need to modify your code to provide a trap door. Ironically, modifying code to encourage flexibility is one of the strongest advantages of using mocks, as explained in sections 8.3.1 and 8.3.2.

8.4.2 Testing a sample method

The example in listing 8.8 demonstrates a code snippet that opens an HTTP connection to a given URL and reads the content at that URL. Suppose that this code is one method of a bigger application that you want to unit-test.

Listing 8.8 A sample method that opens an HTTP connection

```
[...]
import java.net.URL;
import java.net.HttpURLConnection;
import java.io.InputStream;
import java.io.IOException;

public class WebClient {
```

```

public String getContent(URL url) {
    StringBuffer content = new StringBuffer();
    try {
        HttpURLConnection connection =
            (HttpURLConnection) url.openConnection();          #A
        connection.setDoInput(true);
        InputStream is = connection.getInputStream();       #A
        int count;
        while (-1 != (count = is.read())) {               #B
            content.append( new String( Character.toChars( count ) ) ); #B
        }
    } catch (IOException e) {                           #C
        return null;
    }
    return content.toString();
}

```

In this listing:

- We open an HTTP connection (#A).
- We read all the content that is received (#B).
- If an error occurs, we return `null` (#C). Admittedly, this is not the best possible error-handling solution, but it is good enough for the moment (and our tests will give us the courage to refactor later).

8.4.3 Try #1: easy method refactoring technique

The idea of this first refactoring technique applied at Tested Data Systems is to test the `getContent` method independently of a real HTTP connection to a web server. If you map the knowledge you acquired in section 8.2, it means writing a mock URL in which the `url.openConnection` method returns a mock `HttpURLConnection`. The `MockHttpURLConnection` class would provide an implementation that lets the test decide what the `getInputStream` method returns. Ideally, you would be able to write the following test.

Listing 8.9 The `testGetContentOk` method

```

@Test
public void testGetContentOk() throws Exception {
    MockHttpURLConnection mockConnection = new MockHttpURLConnection(); #A
    mockConnection.setupGetInputStream(
        new ByteArrayInputStream("It works".getBytes())); #A
    MockURL mockURL = new MockURL();                      #B
    mockURL.setupOpenConnection(mockConnection);           #B
    WebClient client = new WebClient();                   #C
    String workingContent = client.getContent(mockURL);   #C
    assertEquals("It works", workingContent);             #D
}

```

In this listing:

- We create a mock `HttpURLConnection` (#A).

- We create a mock URL (#B).
- We test the `getContent` method (#C).
- We assert the result (#D).

Unfortunately, this approach does not work! The `JDK URL` class is a final class, and no `URL` interface is available. So much for extensibility.

We need to find another solution and, potentially, another object to mock. One solution is to stub the `URLStreamHandlerFactory` class. We explored this solution in chapter 7, so we must find a technique that uses mock objects: refactoring the `getContent` method. If you think about it, this method does two things: it gets an `HttpURLConnection` object and then reads the content from it. Refactoring leads to the class shown in listing 8.10. (Changes from listing 8.8 are shown in bold.) We have extracted the part that retrieved the `HttpURLConnection` object.

Listing 8.10 Extracting retrieval of the connection object from `getContent`

```
public class WebClient {
    public String getContent(URL url) {
        StringBuffer content = new StringBuffer();
        try {
            HttpURLConnection connection = createHttpURLConnection(url);    #A
            InputStream is = connection.getInputStream();
            int count;
            while (-1 != (count = is.read())) {
                content.append( new String( Character.toChars( count ) ) );
            }
        }
        catch (IOException e) {
            return null;
        }
        return content.toString();
    }
    protected HttpURLConnection createHttpURLConnection(URL url)           #A
                                         throws IOException {                      #A
        return (HttpURLConnection) url.openConnection();
    }
}
```

In this listing, we call `createHttpURLConnection` (#A) to create the HTTP connection.

How does this solution test `getContent` more effectively? It allows us to apply a useful trick, which writes a test helper class that extends the `WebClient` class and overrides its `createHttpURLConnection` method, as follows:

```
private class TestableWebClient extends WebClient {
    private HttpURLConnection connection;
    public void setHttpURLConnection(HttpURLConnection connection) {
        this.connection = connection;
    }
    public HttpURLConnection createHttpURLConnection(URL url)
                                         throws IOException {
        return this.connection;
    }
}
```

A common refactoring approach called *Method Factory* is especially useful when the class to mock has no interface. The strategy is to extend that class, add some setter methods to control it, and override some of its getter methods to return what we want for the test.

In the test, we can call the `setHttpURLConnection` method, passing it the mock `HttpURLConnection` object. Now the test becomes the following. (Differences are shown in bold.)

Listing 8.11 The modified `testGetContentOk` method

```
@Test
public void testGetContentOk() throws Exception {
    MockHttpURLConnection mockConnection = new MockHttpURLConnection();
    mockConnection.setupInputStream(
        new ByteArrayInputStream("It works".getBytes()));
    TestableWebClient client = new TestableWebClient();                      #A
    client.setHttpURLConnection(mockConnection);                            #A
    String workingContent =                                              #B
        client.getContent(new URL("http://localhost"));                      #B
    assertEquals("It works", workingContent);
}
```

In this listing:

- We configure `TestableWebClient` (#A) so that the `createHttpURLConnection` method returns a mock object.
- Next, the `getContent` method is called (#B).

In the case at hand, the Method Factory approach is OK, but it is not perfect. It is a bit like the Heisenberg Uncertainty Principle: the act of subclassing the class under test changes its behavior, so when you test the subclass, what are you truly testing?

This technique is useful as a means of opening an object to be more testable, but stopping here means testing something that is similar (but not identical) to the class you want to test. You are not writing tests for a third-party library and cannot change the code; you have complete control of the code to test. You can enhance the code and make it more test-friendly in the process.

8.4.4 Try #2: refactoring by using a class factory

The developers at Tested Data Systems want to give another refactoring try by applying the IoC pattern, which says that any resource we use needs to be passed to the `getContent` method or `WebClient` class. The only resource we use is the `HttpURLConnection` object. We could change the `WebClient.getContent` signature to

```
public String getContent(URL url, HttpURLConnection connection)
```

This change means we are pushing the creation of the `HttpURLConnection` object to the caller of `WebClient`. The URL is retrieved from the `HttpURLConnection` class, however, and the signature does not look very nice. Fortunately, a better solution involves creating a `ConnectionFactory` interface, as shown in listings 8.12 and 8.13. The role of classes

implementing the `ConnectionFactory` interface is to return an `InputStream` from a connection, whatever the connection may be (HTTP, TCP/IP, and so on). This refactoring technique is sometimes called a *Class Factory* refactoring.

Listing 8.12 ConnectionFactory.java

```
[...]
import java.io.InputStream;
public interface ConnectionFactory {
    InputStream getData() throws Exception;
}
```

The `WebClient` code becomes as shown in listing 8.13. (Changes from the initial implementation in listing 8.8 are shown in bold.)

Listing 8.13 Refactored WebClient using ConnectionFactory

```
[...]
import java.io.InputStream;

public class WebClient {
    public String getContent(ConnectionFactory connectionFactory) {
        String workingContent;
        StringBuffer content = new StringBuffer();
        try(InputStream is = connectionFactory.getData()) {
            int count;
            while (-1 != (count = is.read())) {
                content.append( new String( Character.toChars( count ) ) );
            }
            workingContent = content.toString();
        }
        catch (Exception e) {
            workingContent = null;
        }
        return workingContent;
    }
}
```

This solution is better because we have made the retrieval of the data content independent of the way we get the connection. The first implementation worked only with URLs using the HTTP protocol. The new implementation can work with any standard protocol (`file://`, `http://`, `ftp://`, `jar://`, and so forth) or even a custom protocol. Listing 8.14 shows the `ConnectionFactory` implementation for the HTTP protocol.

Listing 8.14 HttpURLConnectionFactory.java

```
[...]
import java.io.InputStream;
import java.net.HttpURLConnection;
import java.net.URL;

public class HttpURLConnectionFactory implements ConnectionFactory {
    private URL url;
    public HttpURLConnectionFactory(URL url) {
```

```

        this.url = url;
    }
    public InputStream getData() throws Exception {
        HttpURLConnection connection =
            (HttpURLConnection) this.url.openConnection();
        return connection.getInputStream();
    }
}

```

Now we can easily test the `getContent` method by writing a mock for `ConnectionFactory` (see listing 8.15).

Listing 8.15 MockConnectionFactory.java

```

[...]
import java.io.InputStream;

public class MockConnectionFactory implements ConnectionFactory {
    private InputStream inputStream;

    public void setData(InputStream stream) {
        this.inputStream = stream;
    }
    public InputStream getData() throws Exception {
        return inputStream;
    }
}

```

As usual, the mock does not contain any logic and is completely controllable from the outside (by calling the `setData` method). Now we can easily rewrite the test to use `MockConnectionFactory` as demonstrated in listing 8.16.

Listing 8.16 Refactored WebClient test using MockConnectionFactory

```

[...]
import java.io.ByteArrayInputStream;

public class TestWebClient {

    @Test
    public void testGetContentOk() throws Exception {
        MockConnectionFactory mockConnectionFactory =
            new MockConnectionFactory();
        mockConnectionFactory.setData(
            new ByteArrayInputStream("It works".getBytes()));

        WebClient client = new WebClient();
        String workingContent = client.getContent(mockConnectionFactory);
        assertEquals("It works", workingContent);
    }
}

```

We have achieved our initial goal: to unit-test the code logic of the `WebClient.getContent` method, which returns the content of a given URL. During this process, we had to refactor the

method for the test, which led to a more extensible implementation that is better able to cope with change.

8.5 Using mocks as Trojan horses

Mock objects are Trojan horses, but they are not malicious. Mocks replace real objects from the inside without the calling classes being aware of it. Mocks have access to internal information about the class, making them quite powerful. In the examples so far, we have used them to emulate real behaviors, but we haven't mined all the information they can provide.

It is possible to use mocks as probes by letting them monitor the method calls that the object under test makes. Consider the HTTP connection example. One interesting call we could monitor is the `close` method on the `InputStream`, as we would like to make sure that the programmer will always close the stream; otherwise, we may experience a *resource leak*. A *resource leak* occurs when we do not close a reader, scanner, buffer, or another process that uses resources and needs to clean them out of memory. We have not used a mock object for `InputStream` so far, but we can easily create one and provide a `verify` method to ensure that `close` has been called. Then we can call the `verify` method at the end of the test to verify that all methods that should have been called were called (see listing 8.17). We may also want to verify that `close` has been called exactly once and raise an exception if it was called more than once or not at all. These kinds of verifications are often called *expectations*.

DEFINITION Expectation—When we're talking about mock objects, an expectation is a feature built into the mock that verifies whether the external class calling this mock has the correct behavior. A database connection mock, for example, could verify that the `close` method on the connection is called exactly once during any test that involves code using this mock.

To demonstrate the expectations that the resource has been closed and we avoid a resource leak, take a look at the following listing.

Listing 8.17 Mock InputStream with an expectation on close

```
[...]
import java.io.IOException;
import java.io.InputStream;

public class MockInputStream extends InputStream {
    private String buffer;
    private int position = 0;
    private int closeCount = 0;
    public void setBuffer(String buffer) {
        this.buffer = buffer;
    }
    public int read() throws IOException {
        if (position == this.buffer.length()) {           #A
            return -1;                                  #A
        }
    }
}
```

```

        }
        return this.buffer.charAt(this.position++);
    }
    public void close() throws IOException {
        closeCount++;
        super.close();
    }
    public void verify() throws java.lang.AssertionError {
        if (closeCount != 1) {
            throw new AssertionError ("close() should "
                + "have been called once and once only");
        }
    }
}

```

in this listing:

- We tell the mock what the `read` method should return (#A).
- We count the number of times `close` is called (#B).
- We verify that the expectations are met (#C).

In the case of the `MockInputStream` class, the expectation for `close` is simple: we always want it to be called once. Most of the time, however, the expectation for `closeCount` depends on the code under test. A mock usually has a method such as `setExpectedCloseCalls` so that the test can tell the mock what to expect.

Modify the `TestWebClient.testGetContentOk` test method as follows to use the new `MockInputStream`:

```

[...]
public class TestWebClient {

    @Test
    public void testGetContentOk() throws Exception {
        MockConnectionFactory mockConnectionFactory =
            new MockConnectionFactory();

        MockInputStream mockStream = new MockInputStream();
        mockStream.setBuffer("It works");
        mockConnectionFactory.setData(mockStream);
        WebClient client = new WebClient();
        String workingContent = client.getContent(mockConnectionFactory);

        assertEquals("It works", workingContent);
        mockStream.verify();
    }
}

```

Instead of using a real `ByteArrayInputStream`, as in previous tests, we use the `MockInputStream`. Note that we call the `verify` method of `MockInputStream` at the end of the test to ensure that all expectations are met. The result of the test is shown in figure 8.4.

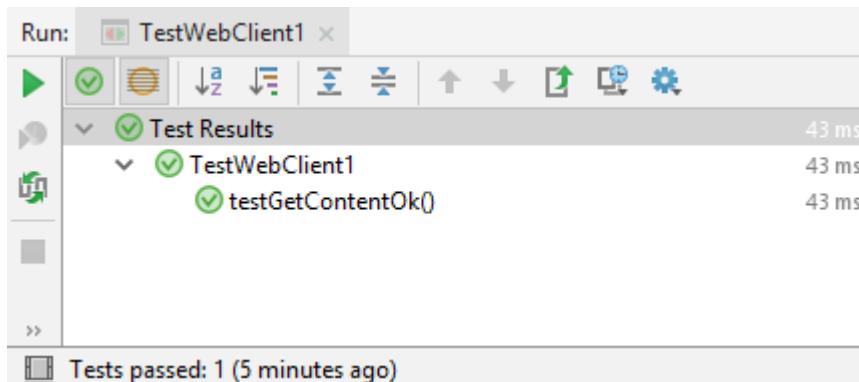


Figure 8.4 The result of running `TestWebClient`

There are other handy uses for expectations. If you have a component manager calling different methods of your component life cycle, for example, you might expect them to be called in a given order. Or you might expect a given value to be passed as a parameter to the mock. The general idea is that aside from behaving the way we want during a test, our mock can provide useful feedback on its use. The test can provide information regarding the number of methods invocations, parameters that are passed to the methods, the order of methods calling.

The next section demonstrates the use of some of the most popular open-source mocking frameworks, which are powerful enough for our needs, and we don't need to implement our mocks from the beginning.

8.6 Introducing Mock frameworks

So far, the engineers at Tested Data Systems have implemented the mock objects from scratch. The task is not tedious, but it recurs. You might guess that we don't need to reinvent the wheel every time we need a mock, and you would be right: a lot of good projects already written can facilitate the use of mocks in projects.

In this section, we will take a closer look at three of the most widely used mock frameworks: EasyMock, JMock, and Mockito.

The developers at Tested Data Systems will try to rework the example HTTP connection application to demonstrate how to use the three frameworks and to come up with a basis for choosing one of the alternatives. People have their own experiences, preferences, and habits, and because the engineers have these three framework alternatives, they would like to compare them and make some conclusions.

8.6.1 Using EasyMock

EasyMock (<http://easymock.org>) is an open-source framework that provides useful classes for mocking objects. To work with it, we need to add to the pom.xml file the dependencies shown in listing 8.18.

Listing 8.18 The EasyMock dependencies from the pom.xml configuration file

```
<dependency>
    <groupId>org.easymock</groupId>
    <artifactId>easymock</artifactId>
    <version>2.4</version>
</dependency>
<dependency>
    <groupId>org.easymock</groupId>
    <artifactId>easymockclassextension</artifactId>
    <version>2.4</version>
</dependency>
```

While trying to introduce EasyMock, the developers at Tested Data Systems revise some of the mocks constructed in the previous sections. The start is simple: reworking the AccountService test from listing 8.2.

Listing 8.19 Reworking the TestAccountService test using EasyMock

```
[...]
import static org.easymock.EasyMock.createMock;                      #A
import static org.easymock.EasyMock.replay;                            #A
import static org.easymock.EasyMock.expect;                           #A
import static org.easymock.EasyMock.verify;                          #A

public class TestAccountServiceEasyMock
{
    private AccountManager mockAccountManager;                         #B

    @BeforeEach
    public void setUp()
    {
        mockAccountManager = createMock( "mockAccountManager",
                                         AccountManager.class ); #C
    }

    @Test
    public void testTransferOk()
    {
        Account senderAccount = new Account( "1", 200 );                 #D
        Account beneficiaryAccount = new Account( "2", 100 );                #D

        // Start defining the expectations
        mockAccountManager.updateAccount( senderAccount );                  #E
        mockAccountManager.updateAccount( beneficiaryAccount );             #E

        expect( mockAccountManager.findAccountForUser( "1" ) )
            .andReturn( senderAccount );                                     #F
        expect( mockAccountManager.findAccountForUser( "2" ) )
            .andReturn( beneficiaryAccount ); #F
```

```

// we're done defining the expectations
replay( mockAccountManager );                                #G

AccountService accountService = new AccountService();
accountService.setAccountManager( mockAccountManager );
accountService.transfer( "1", "2", 50 );                      #H

assertEquals( 150, senderAccount.getBalance() );            #I
assertEquals( 150, beneficiaryAccount.getBalance() );        #I
}

@AfterEach
public void tearDown()
{
    verify( mockAccountManager );                            #J
}
}

```

As you see, the listing is pretty much the same size as listing 8.4 but does not write additional mock classes. in this listing:

- We start the listing defining the imports from the EasyMock library that we need (#A). We rely heavily on static imports.
- We declare the object that we would like to mock (#B). Notice that our `AccountManager` is an interface. The reason is simple: the core EasyMock framework can mock only interface objects.
- We call the `createMock` method to create a mock of the class that we want (#C).
- As in listing 8.4, we create two account objects that we are going to use in our tests (#D). After that, we start declaring our expectations.
- With EasyMock, we declare the expectations in two ways. When the method return type is `void`, we call it on the mock object (#E), or when the method returns any kind of object, we use the `expect-andReturn` methods from the EasyMock API (#F).
- When we finish defining the expectations, we call the `replay` method. The `replay` method passes the mock from the phase where we record the method we expect to be called to where we test. Before, we simply recorded the behavior, but the object was not working as a mock. After calling `replay`, the object works as expected (#G).
- We call the `transfer` method to transfer some money between the two accounts (#H).
- We assert the expected result (#I).
- The `@AfterEach` method that gets executed after every `@Test` method holds the verification of the expectations. With EasyMock, we can call the `verify` method with any mock object (#J) to verify that the method-call expectations we declared were triggered. Including the verification in the `@AfterEach` method allows us to introduce new tests easily, and we'll rely on the execution of the `verify` method from now on.

JUNIT best practices: EasyMock object creation

Here is a nice-to-know tip on the `createMock` method. If you check the API of EasyMock, you see that the `createMock` method comes with numerous signatures. The signature that we use is

```
createMock(String name, Class clazz);
```

but there's also

```
createMock(Class clazz);
```

Which one should we use? `createMock(String name, Class clazz)` is better. If you use `createMock(Class clazz)` and your expectations are not met, you get an error message like the following:

```
java.lang.AssertionError:  
Expectation failure on verify:  
    read(): expected: 7, actual: 0
```

As you see, this message is not as descriptive as it could be. If you use `createMock(String name, Class clazz)` instead and map the class to a given name, you would get something like the following:

```
java.lang.AssertionError:  
Expectation failure on verify:  
    name.read(): expected: 7, actual: 0
```

That was pretty easy, right? So how about moving a step forward and revising a more-complicated example? The next listing demonstrates the reworked `WebClient` test from listing 8.16: verifying the correct value returned by the `getContent` method.

We want to test the `getContent` method of the `WebClient`. For this purpose, we need to mock all the dependencies to that method. In this example, we have two dependencies: `ConnectionFactory` and `InputStream`. It appears to be a problem because EasyMock can mock only interfaces and `InputStream` is a class.

To mock the `InputStream` class, we are going to use the class extensions of EasyMock, which represent an extension project of EasyMock that lets you generate mock objects³ for classes and interfaces. These class extensions are addressed by the second Maven dependency in listing 8.18.

³ final and private methods cannot be mocked.

Listing 8.20 Reworking the WebClient test using EasyMock

```
[...]
import static org.easymock.classextension.EasyMock.createMock;          #A
import static org.easymock.classextension.EasyMock.replay;                #A
import static org.easymock.classextension.EasyMock.verify;                #A

public class TestWebClientEasyMock
{
    private ConnectionFactory factory;                                     #B
    private InputStream stream;                                              #B

    @BeforeEach
    public void setUp()
    {
        factory = createMock( "factory", ConnectionFactory.class );      #C
        stream = createMock( "stream", InputStream.class );                 #C
    }

    @Test
    public void testGetContentOk() throws Exception
    {
        expect( factory.getData() ).andReturn( stream );
        expect( stream.read() ).andReturn( new Integer( (byte) 'W' ) );     #D
        expect( stream.read() ).andReturn( new Integer( (byte) 'o' ) );     #D
        expect( stream.read() ).andReturn( new Integer( (byte) 'r' ) );     #D
        expect( stream.read() ).andReturn( new Integer( (byte) 'k' ) );     #D
        expect( stream.read() ).andReturn( new Integer( (byte) 's' ) );     #D
        expect( stream.read() ).andReturn( new Integer( (byte) '!' ) );     #D
        expect( stream.read() ).andReturn( -1 );                                #D
        stream.close();                                                       #E

        replay( factory );                                                 #F
        replay( stream );                                                 #F

        WebClient2 client = new WebClient2();
        String workingContent = client.getContent( factory );               #G

        assertEquals( "Works!", workingContent );                            #H
    }

    [...]
    @Test
    public void testGetContentCannotCloseInputStream() throws Exception {
        expect( factory.getData() ).andReturn( stream );
        expect( stream.read() ).andReturn( -1 );
        stream.close();                                                    #I
        expectLastCall().andThrow(new IOException("cannot close"));       #J

        replay( factory );
        replay( stream );
        WebClient2 client = new WebClient2();
        String workingContent = client.getContent( factory );

        assertNull( workingContent );
    }

    @AfterEach
    public void tearDown()
```

```
{  
    verify( factory );  
    verify( stream );  
}  
}
```

In this listing:

- We start by importing the objects that we need (#A). Notice that because we use the class extensions of EasyMock, we need to import the `org.easymock.classextension.EasyMock` object instead of `org.easymock.EasyMock`. Now we are ready to create mock objects of classes and interfaces by using the statically imported methods of the class extensions.
- As in the previous listings, we declare the objects that we want to mock (#B) call the `createMock` method to initialize them (#C).
- We define the expectation of the stream when the `read` method is invoked (#D). (Notice that to stop reading from the `stream`, the last thing to return is a `-1`.) Working with a low-level stream, we define how to read one byte at a time, as `InputStream` is reading byte by byte. We expect the `close` method to be called on the `stream` (#E).
- To denote that we are done declaring our expectations, we call the `replay` method (#F). The `replay` method passes the mock from the phase where we record the method that we expect to be called to where we test. Before, we simply recorded the behavior, but the object is not working as a mock. After calling `replay`, the object works as expected.
- The rest is invoking the method under test (#G) and asserting the expected result (#H).
- We also add another test to simulate a condition when we cannot close the `InputStream`. We define an expectation in which we expect the `close` method of the `stream` to be invoked (#I).
- On the next line, we declare that an `IOException` should be raised if this call occurs (#J).

As the name of the framework suggests, using EasyMock is very easy, and it is an option for the projects. But to make you aware of the whole mocking picture, we would like to introduce another framework, following the experiments of three developers at Tested Data Systems. We will mock parts of the account service and of the web client using three mocking frameworks—EasyMock, JMock, and Mockito—to give you a better idea of what the mocking is.

8.6.2 Using JMock

So far, you've seen how to implement your own mock-objects and use the EasyMock framework. In this section, we introduce the JMock framework (<http://jmock.org>). We'll follow the same scenario that the engineers from Tested Data Systems are following to evaluate the capabilities of a mock framework and compare them with those of other frameworks: testing money transfer with the help of a mock `AccountManager`, this time using JMock.

To work with JMock, we need to add to the pom.xml file the dependencies shown in listing 8.21.

Listing 8.21 The JMock dependencies from the pom.xml configuration file

```
<dependency>
    <groupId>org.jmock</groupId>
    <artifactId>jmock-junit5</artifactId>
    <version>2.12.0</version>
</dependency>
<dependency>
    <groupId>org.jmock</groupId>
    <artifactId>jmock-legacy</artifactId>
    <version>2.5.1</version>
</dependency>
```

As in section 8.6.1, we start with a simple example: reworking listing 8.4 by means of JMock, as shown in listing 8.22.

Listing 8.22 Reworking the TestAccountService test using JMock

```
[...]
import org.jmock.Expectations;                                #A
import org.jmock.Mockery;                                     #A
import org.jmock.junit5.JUnit5Mockery;                         #A

public class TestAccountServiceJMock
{
    @RegisterExtension                                         #B
    Mockery context = new JUnit5Mockery();                      #B

    private AccountManager mockAccountManager;                  #C

    @BeforeEach
    public void setUp()
    {
        mockAccountManager = context.mock( AccountManager.class );      #D
    }

    @Test
    public void testTransferOk()
    {
        Account senderAccount = new Account( "1", 200 );           #E
        Account beneficiaryAccount = new Account( "2", 100 );         #E

        context.checking( new Expectations()                         #F
        {
            {
                oneOf( mockAccountManager ).findAccountForUser( "1" ); #G
                will( returnValue( senderAccount ) );
                oneOf( mockAccountManager ).findAccountForUser( "2" );
                will( returnValue( beneficiaryAccount ) );

                oneOf( mockAccountManager ).updateAccount( senderAccount );
                oneOf( mockAccountManager )
                    .updateAccount( beneficiaryAccount );
            }
        });
    }
}
```

```

} );

AccountService accountService = new AccountService();
accountService.setAccountManager( mockAccountManager );
accountService.transfer( "1", "2", 50 );                                #H
assertEquals( 150, senderAccount.getBalance() );                      #I
assertEquals( 150, beneficiaryAccount.getBalance() );                  #I
}
}

```

In this listing:

- As always, we by importing all the objects we need (#A). Unlike EasyMock, the JMock framework does not rely on any static import features.
- JUnit5 provides a programmatic way to register extensions. For JMock, this way is annotating a JUnit5Mockery nonprivate instance field with @RegisterExtension. (Part 4 discusses the JUnit 5 extension model in detail.) The context object serves us to create mocks and define expectations (#B).
- We declare the AccountManager that we would like to mock (#C). Just like EasyMock, the core JMock framework provides mocking only of interfaces.
- In the @BeforeEach method that gets executed before each of the @Test methods, we create the mock programmatically by means of the context object (#D).
- As in previous listings, we declare two accounts that we are going to transfer money between (#E).
- We start declaring the expectations by constructing a new Expectations object (#F).
- We declare the first expectation (#G), with each expectation having the form

```

invocation-count (mock-object).method(argument-constraints);
    inSequence(sequence-name);
    when(state-machine.is(state-name));
    will(action);
    then(state-machine.is(new-state-name));

```

All the clauses are optional except for the bold ones: `invocation-count` and `mock-object`. We need to specify how many invocations will occur and on which object. After that, in case the method returns some object, we can declare the return object by using the `will(returnValue())` construction.

- We start the transfer from one account to the other one (#H) and then assert the expected results (#I). It's just as simple as that!

But wait—what happened with the verification of the invocation count? In all the previous examples, we needed to verify that the invocations of the expectations happened the expected number of times. Well, with JMock, you don't have to do that. The JMock extension takes care of this task, and if the expected calls were not made, the test will fail.

Following the EasyMock pattern, we rework listing 8.20, showing the WebClient test, this time using JMock.

Listing 8.23 Reworking the TestWebClient test using JMock

```

[...]
public class TestWebClientJMock
{
    @RegisterExtension
    Mockery context = new JUnit5Mockery()
    {
        {
            setImposteriser( ClassImposteriser.INSTANCE );
        }
    };
}

@Test
public void testGetContentOk() throws Exception
{
    ConnectionFactory factory =
        context.mock( ConnectionFactory.class );
    InputStream mockStream =
        context.mock( InputStream.class );

    context.checking( new Expectations()
    {
        {
            oneOf( factory ).getData();
            will(returnValue( mockStream ) );

            atLeast( 1 ).of( mockStream ).read();
            will( onConsecutiveCalls(
                returnValue( Integer.valueOf( (byte) 'W' ) ),
                returnValue( Integer.valueOf( (byte) 'o' ) ),
                returnValue( Integer.valueOf( (byte) 'r' ) ),
                returnValue( Integer.valueOf( (byte) 'k' ) ),
                returnValue( Integer.valueOf( (byte) 's' ) ),
                returnValue( Integer.valueOf( (byte) '!' ) ),
                returnValue( -1 ) ) );

            oneOf( mockStream ).close();
        }
    } );
}

WebClient2 client = new WebClient2();
String workingContent = client.getContent( factory );
assertEquals( "Works!", workingContent );
}

@Test
public void testGetContentCannotCloseInputStream() throws Exception
{
    ConnectionFactory factory =
        context.mock( ConnectionFactory.class );
    InputStream mockStream = context.mock( InputStream.class );

    context.checking( new Expectations()
    {
        {
            oneOf( factory ).getData();
            will(returnValue( mockStream ) );
            atLeast( 1 ).of( mockStream ).read();
            will( onConsecutiveCalls(
                returnValue( Integer.valueOf( (byte) 'W' ) ),
                returnValue( Integer.valueOf( (byte) 'o' ) ),
                returnValue( Integer.valueOf( (byte) 'r' ) ),
                returnValue( Integer.valueOf( (byte) 'k' ) ),
                returnValue( Integer.valueOf( (byte) 's' ) ),
                returnValue( Integer.valueOf( (byte) '!' ) ),
                returnValue( -1 ) ) );
        }
    } );
}

```

```

        oneOf( factory ).getData();
        will(.returnValue( mockStream ) );
        oneOf( mockStream ).read();
        will(.returnValue( -1 ) );
        oneOf( mockStream ).close();                                #H
        will( throwException(
                new IOException( "cannot close" ) ) );      #I
    }
}

WebClient2 client = new WebClient2();

String workingContent = client.getContent( factory );

assertNull( workingContent );
}
}

```

In this listing:

- We start the test-case by registering the JMock extension. The JUnit5Mockery nonprivate instance field `context` is annotated with `@RegisterExtension (#A)`. The JUnit 5 extension model will be analyzed in part 4.
- To tell JMock to create mock objects not only for interfaces but also for classes, we need to set the `impostoriser` property of the context (#B). Now we can continue creating mocks the normal way.
- We declare and programmatically initialize the two objects of which we want to create mocks (#C).
- We start declaring the expectations (#D) Notice the fine way in which we declare the consecutive execution of the `read()` method of the stream (#E) and the returned values.
- We call the method under test (#F).
- We assert the expected result (#G).
- For a full view of how to use the JMock mocking library, we also provide another `@Test` method that tests our `WebClient` under exceptional conditions. We declare the expectation of the `close()` method being triggered (#H), and we instruct JMock to raise an `IOException` when this trigger happens (#I).

As you can see, the JMock library is as easy to use as the EasyMock one but provides better integration with JUnit 5. We can register the `Mockery context` field programmatically. But we still have to look at the third proposed framework, Mockito, which is closer to the JUnit 5 paradigm.

8.6.3 Using Mockito

This section introduces Mockito (<https://site.mockito.org>), another popular mocking framework. The engineers at Tested Data Systems want to evaluate it and eventually introduce it into their projects.

To work with Mockito, you need to add to the pom.xml file the dependency shown in listing 8.24.

Listing 8.24 The Mockito dependency from the pom.xml configuration file

```
<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-junit-jupiter</artifactId>
    <version>2.21.0</version>
    <scope>test</scope>
</dependency>
```

As with EasyMock and JMock, rework the example from listing 8.4 (testing money transfer with the help of a mock AccountManager), this time by means of Mockito, as shown in listing 8.25.

Listing 8.25 Reworking the TestAccountService test using Mockito

[...]

```
import org.junit.jupiter.api.extension.ExtendWith;          #A
import org.mockito.Mock;                                     #A
import org.mockito.Mockito;                                  #A
import org.mockito.junit.jupiter.MockitoExtension;           #A

@ExtendWith(MockitoExtension.class)                         #B
public class TestAccountServiceMockito
{

    @Mock
    private AccountManager mockAccountManager;              #C

    @Test
    public void testTransferOk()
    {
        Account senderAccount = new Account( "1", 200 );      #D
        Account beneficiaryAccount = new Account( "2", 100 );   #D

        Mockito.lenient()
            .when(mockAccountManager.findAccountForUser("1"))
            .thenReturn(senderAccount);
        Mockito.lenient()
            .when(mockAccountManager.findAccountForUser("2"))
            .thenReturn(beneficiaryAccount);                      #E

        AccountService accountService = new AccountService();
        accountService.setAccountManager( mockAccountManager );
        accountService.transfer( "1", "2", 50 );                #F

        assertEquals( 150, senderAccount.getBalance() );         #G
        assertEquals( 150, beneficiaryAccount.getBalance() );
    }
}
```

In this listing:

- As usual, we start by importing all the objects we need (#A). This example does not rely on static import features.
- We extend this test by using MockitoExtension (#B). @ExtendWith is a repeatable annotation that is used to register extensions for the annotated test class or test method. (We discuss the JUnit 5 extension model in detail in part 4.) For this Mockito example, we'll only note that this extension is needed to create mock objects through annotations (#C). This code tells Mockito to create a mock object of type AccountManager.
- As in the previous listings, we declare two accounts that we are going to transfer money between (#D).
- We start declaring the expectations by using the when method (#E). Additionally, we use the lenient method to modify the strictness of object mocking. Without this method, only one expectation declaration is allowed for the same findAccountForUser method, whereas we need two (one for the "1" argument and one for the "2" argument).
- We start the transfer from one account to the other (#F).
- We assert the expected results (#G).

Following the pattern in the previous sections, try to rework listing 8.20, which shows the WebClient test, with Mockito.

Listing 8.26 Reworking the TestWebClient test using Mockito

```
[...]
import org.mockito.Mockito;                                     #A
import org.mockito.junit.jupiter.MockitoExtension;           #A
import static org.mockito.Mockito.doThrow;                   #A
import static org.mockito.Mockito.when;                      #A

@ExtendWith(MockitoExtension.class)                           #B
public class TestWebClientMockito
{
    @Mock
    private ConnectionFactory factory;                         #C
    @Mock
    private InputStream mockStream;                            #C

    @Test
    public void testGetContentOk() throws Exception
    {
        when(factory.getData()).thenReturn(mockStream);       #D
        when(mockStream.read()).thenReturn((int) 'W')          #E
            .thenReturn((int) 'o')                            #E
            .thenReturn((int) 'r')                            #E
            .thenReturn((int) 'k')                            #E
            .thenReturn((int) 's')                            #E
            .thenReturn((int) '!')                            #E
            .thenReturn(-1);                                #E

        WebClient2 client = new WebClient2();
    }
}
```

```

        String workingContent = client.getContent( factory );           #F
        assertEquals( "Works!", workingContent );                         #G
    }

    @Test
    public void testGetContentCannotCloseInputStream()
        throws Exception
    {
        when(factory.getData()).thenReturn(mockStream);                 #H
        when(mockStream.read()).thenReturn(-1);                          #I
        doThrow(new IOException( "cannot close" ))                      #J
            .when(mockStream).close();                                  #J

        WebClient2 client = new WebClient2();

        String workingContent = client.getContent( factory );
        assertNull( workingContent );
    }
}

```

In this listing:

- We import the needed dependencies, static and nonstatic in this example (#A).
- We extend this test by using MockitoExtension (#B).
- In this example, the extension is needed to create mock objects through annotations (#C.) It tells Mockito to create one mock object of type `ConnectionFactory` and one mock object of type `InputStream`.
- We start the declaration of the expectations (#D). Notice the fine way that we declare the consecutive execution of the `read()` method of the stream (#E) and the returned values.
- We call the method under test (#F).
- We assert the expected result (#G).
- We provide another `@Test` method that tests our `WebClient` under exceptional conditions. We declare the expectation of the `factory.getData()` method (#H), and we declare the expectation of the `mockStream.read()` method (#I). Then we instruct Mockito to raise an `IOException` when we close the stream (#J).

As you can see, the Mockito framework may be used with the new JUnit 5 extension model—not programmatically, as JMock, but through the use of the JUnit 5 `@ExtendWith` and the Mockito `@Mock` annotation. We use it to a greater extent in other chapters, as it is better integrated with JUnit 5.

Chapter 9 addresses another approach to unit-testing components: in-container unit testing, or integration unit testing.

8.7 Summary

This chapter has covered the following:

- Demonstrating the mock objects technique. This one lets you unit-test code in isolation from other domain objects and from the environment. When it comes to writing fine-grained unit tests, it helps you abstract yourself from the executing environment.
- Proving a nice side effect of writing mock-object tests: it forces you to rewrite some of the code under test. In practice, code is often not well enough written. With mock objects, you must think differently about the code and apply better design patterns, like interfaces and inversion of control (IoC).
- Implementing and comparing the method factory and class factory techniques for refactoring a code that is using a mock HTTP connection.
- Using three mocking frameworks: EasyMock, JMock and Mockito and implementing two examples of how to work with them in a real environment: mocking parts of the account service and of the web client. From the three frameworks, we have demonstrated that Mockito has at this time the best integration with JUnit 5, in particular with the JUnit 5 extension model.

9

In-container testing

This chapter covers

- Analyzing the limitations of mock objects
- Using in-container testing
- Evaluating and comparing stubs, mock objects, and in-container testing
- Working with Arquillian

The secret of success is sincerity. Once you can fake that you've got it made.

—Jean Giraudoux

This chapter examines one approach to unit-testing components in an application container: in-container unit testing, or integration testing. These components are modules that may be developed by different programmers or teams and that need to be tested together or integrated. We will analyze in-container testing pros and cons, and show what you can achieve by using the mock-objects approach introduced in chapter 8; where mock objects fall short; and how in-container testing enables you to write integration unit tests. We will also work with Arquillian, a Java EE container-agnostic framework for integration testing, and show you how to use it to conduct integration testing. Finally, we will compare the stubs, mock objects, and in-container approaches covered in this part of this book.

9.1 Limitations of standard unit testing

Start with the example servlet in listing 9.1, which implements the `HttpServletRequest` method `isAuthenticated`, the method we want to unit-test. A *servlet* is a Java software application that extends the capabilities of a server. The example company Tested Data Systems uses

servlets to develop web applications, one of which is an online shop that serves new customers. To access the online shop, users need to connect to the frontend interface. The online shop needs authentication mechanisms so that the client knows who is making the operations, and the Tested Data Systems engineers would like to test a method that verifies whether a user is authenticated.

Listing 9.1 Servlet implementing isAuthenticated

```
[...]
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpSession;

public class SampleServlet extends HttpServlet {
    public boolean isAuthenticated(HttpServletRequest request) {
        HttpSession session = request.getSession(false);
        if (session == null) {
            return false;
        }
        String authenticationAttribute =
            (String) session.getAttribute("authenticated");
        return Boolean.valueOf(authenticationAttribute).booleanValue();
    }
}
```

This servlet, although it is simple enough, allows us to show the limitation of standard unit testing. To test the method `isAuthenticated`, we need a valid `HttpServletRequest`. Because `HttpServletRequest` is an interface, we cannot just call new `HttpServletRequest`. The `HttpServletRequest` life cycle and implementation are provided by the container (in this case, a servlet container). The same is true of other server-side objects, such as `HttpSession`. JUnit alone is not enough to write a test for the `isAuthenticated` method and for servlets in general.

DEFINITION `component` and `container`—A `component` is an application or a part of an application. A `container` is an isolated space where a component executes. A container offers services for the components it is hosting, such as life cycle, security, transactions, and so forth.

In the case of servlets and Java Server Pages (JSP), the container is a servlet container like Jetty and Tomcat. Other types of containers include JBoss (renamed WildFly), which is an Enterprise Java Beans (EJB) container. Java code can run in all these containers.

As long as a container creates and manages objects at run time, we cannot use the standard JUnit techniques (the features of JUnit 5, stubs, and mock objects) to test those objects.

9.2 The mock-objects solution

The engineers at Tested Data Systems have to test the authentication mechanisms of the online shop. The first solution they consider for unit-testing the `isAuthenticated` method

(listing 9.1) is to mock the `HttpServletRequest` class, using the approach described in chapter 8. Although mocking works, we need to write a lot of code to create a test. We can achieve the same result more easily by using the open-source EasyMock¹ framework (see chapter 8), as listing 9.2 demonstrates.

Listing 9.2 Testing a servlet with EasyMock

```
[...]
import javax.servlet.http.HttpServletRequest;                      #A
import static org.easymock.EasyMock.createStrictMock;            #A
import static org.easymock.EasyMock.expect;                      #A
import static org.easymock.EasyMock.replay;                      #A
import static org.easymock.EasyMock.verify;                      #A
import static org.easymock.EasyMock.eq;                          #A
import static org.junit.jupiter.api.Assertions.assertFalse;        #A
import static org.junit.jupiter.api.Assertions.assertTrue;          #A
[...]
public class EasyMockSampleServletTest {

    private SampleServlet servlet;
    private HttpServletRequest mockHttpServletRequest;               #B
    private HttpSession mockHttpSession;                         #B

    @BeforeEach
    public void setUp() {                                         #C
        servlet = new SampleServlet();
        mockHttpServletRequest =
            createStrictMock(HttpServletRequest.class);             #C
        mockHttpSession = createStrictMock(HttpSession.class);   #C
    }

    @Test
    public void testIsAuthenticatedAuthenticated() {
        expect(mockHttpServletRequest.getSession(eq(false)))      #D
            .andReturn(mockHttpSession);
        expect(mockHttpSession.getAttribute(eq("authenticated")))  #D
            .andReturn("true");
        replay(mockHttpServletRequest);                            #E
        replay(mockHttpSession);                             #E
        assertTrue(servlet.isAuthenticated(mockHttpServletRequest)); #F
    }

    @Test
    public void testIsAuthenticatedNotAuthenticated() {
        expect(mockHttpSession.getAttribute(eq("authenticated")))
            .andReturn("false");
        replay(mockHttpSession);
        expect(mockHttpServletRequest.getSession(eq(false)))
            .andReturn(mockHttpSession);
        replay(mockHttpServletRequest);
    }
}
```

¹ <http://easymock.org>

```

        assertFalse(servlet.isAuthenticated(mockHttpServletRequest));
    }

    @Test
    public void testIsAuthenticatedNoSession() {
        expect(mockHttpServletRequest.getSession(eq(false))).andReturn(null);
        replay(mockHttpServletRequest);
        replay(mockHttpSession);
        assertFalse(servlet.isAuthenticated(mockHttpServletRequest));
    }

    @AfterEach
    public void tearDown() {                                #G
        verify(mockHttpServletRequest);                   #H
        verify(mockHttpSession);                         #H
    }
}

```

In this listing:

- We start by importing the necessary classes and methods by using static imports. We use the EasyMock class extensively (#A).
- Next, we declare instance variables for the objects (#B) we want to mock: `HttpServletRequest` and `HttpSession`.
- The `setUp` method annotated with `@BeforeEach` (#C) runs before each call to `@Test` methods; this is where we instantiate all the mock objects.
- Next, we implement our tests, following this pattern:
 - Set our expectations by using the EasyMock API (#D).
 - Invoke the `replay` method to finish declaring our expectations (#E). The `replay` method passes the mock from the phase where we record the method that we expect to be called to where we test. Before, we simply recorded the behavior, but the object was not working as a mock. After calling `replay`, the object works as expected.
 - Assert test conditions on the servlet #F.
- The `@AfterEach` method executed after each `@Test` method (#G) calls the EasyMock `verify` API method (#H) to check whether the mocked objects met all of our programmed expectations.

Mocking a minimal portion of a container is a valid approach for testing components. But mocking can be complicated and require a lot of code. The source code provided with this book also includes the servlet testing versions for the JMock and Mockito frameworks. As with other kinds of tests, when the servlet changes, the test expectations must change to match. In the next section, the engineers at Tested Data Systems attempt to ease the task of testing the online shop's authentication mechanism.

9.3 The step to in-container testing

The next approach in testing the `SampleServlet` is running the test cases where the `HttpServletRequest` and `HttpSession` objects live: in the container itself. This approach eliminates the need to mock any objects; we simply access the objects and methods we need in the real container.

For our example of testing the online shop's authentication mechanism, we need the web request and the session to be real `HttpServletRequest` and `HttpSession` objects managed by the container. Using a mechanism to deploy and execute our tests in a container, we have in-container testing. The next section covers options for in-container tests.

9.3.1 Implementation strategies

Two architectural choices drive in-container tests: server-side and client-side. As stated in section 9.3, we can drive the tests directly by controlling the server-side container and the unit tests. Alternatively, we can drive the tests from the client-side, as shown in figure 9.1.

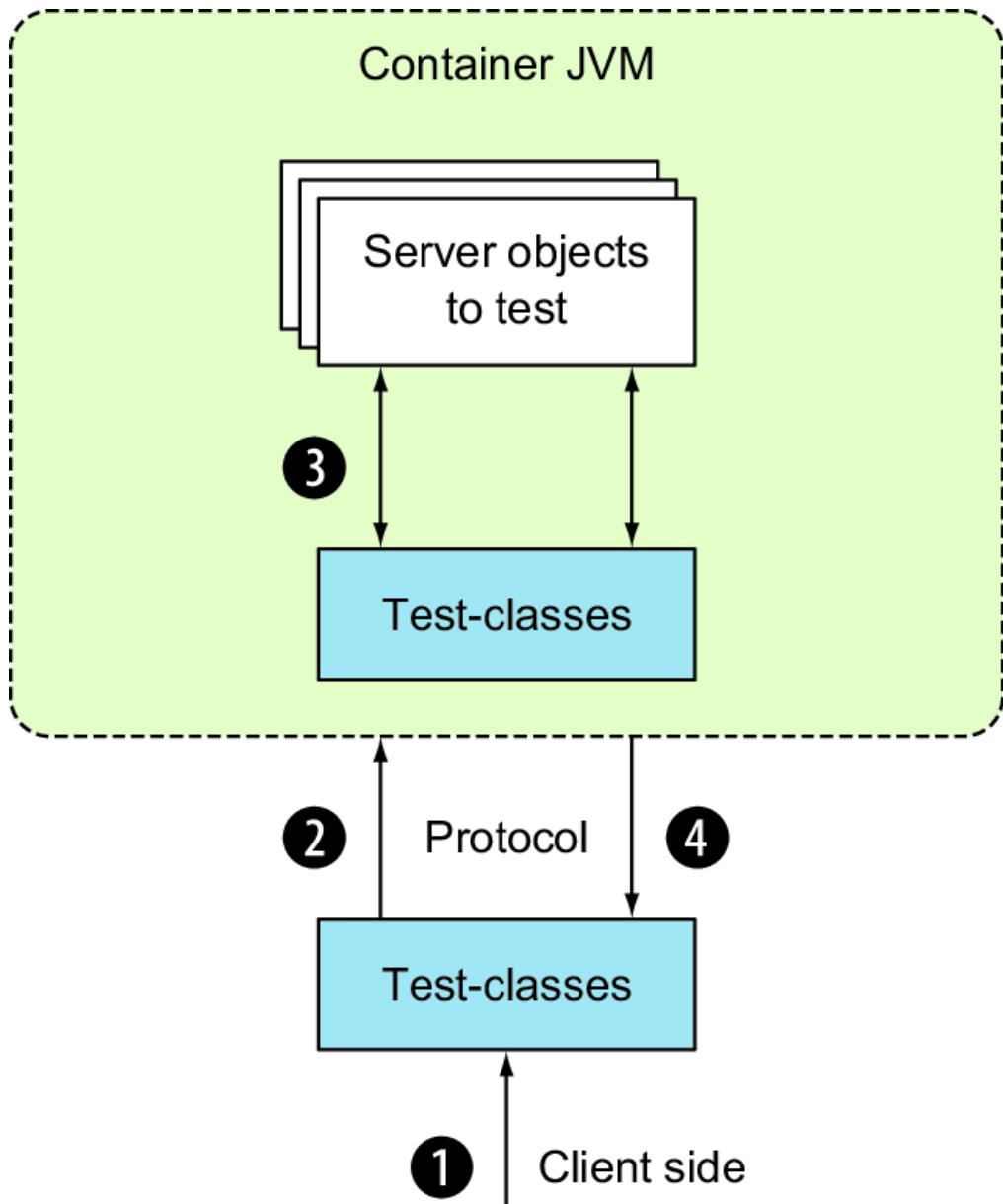


Figure 9.1 Life cycle of a typical in-container test: (1) executing the client test classes, (2) calling the same test case on the server-side, (3) testing the domain objects, and (4) returning the results to the client

When the tests are packaged and deployed in the container and to the client, the JUnit test runner executes the test classes on the client (1). A test class opens a connection via a protocol such as HTTP(S) and calls the same test case on the server-side (2). The server-side test case operates on server-side objects that are normally available (`HttpServletRequest`, `HttpServletResponse`, `HttpSession`, `ServletContext`, and so on) and tests our domain objects (3). The server returns the result of the tests to the client (4), which an IDE or Maven can gather.

9.3.2 In-container testing frameworks

As we have just seen, in-container testing is applicable when code interacts with a container, and tests cannot create valid container objects (`HttpServletRequest` in the preceding section).

Our example uses a servlet container, but many other types of containers are available, including Java EE, web server, applets, and EJB. In all these cases, the in-container testing strategy can be applied.

9.4 Comparing stubs, mock objects, and in-container testing

This section compares² the approaches we presented to test components: stubs, mocks, and in-container testing. It draws on many questions in forums and mailing lists.

9.4.1 Stubs evaluation

Chapter 7 introduces using stubs as an out-of-container testing technique. Stubs work well to isolate a given class for testing and assert the state of its instances. Stubbing a servlet container allows us to track the number of requests made, the state of the server is, and which URLs were requested, for example. But stubs have some predefined behaviors from the beginning of their existence.

When using mock objects, we code and verify expectations. The engineers at Tested Data Systems were able to check at every step that test-execute the methods, including the business logic and how many times tests call these methods.

One of the biggest advantages of stubs compared with mocks is that stubs are easier to understand. Stubs isolate a class with little extra code compared with mock objects, which require an entire framework to function. The drawbacks of stubs are that they rely on external tools and hacks and that they do not track the state objects they fake.

² For an in-depth comparison of stubs and mocks technology, see "Mocks Aren't Stubs," by Martin Fowler, at <http://martinfowler.com/articles/mocksArentStubs.html>.

In chapter 7, the engineers at Tested Data Systems easily faked a servlet container with stubs. Doing so with mock objects would be much harder, however, because they would need to fake container objects with state and behavior.

Here is a summary of stub pros and cons:

Pros:

- Fast and lightweight
- Easy to write and understand
- Powerful
- More coarse-grained tests

Cons:

- Specialized methods required to verify the state
- Do not test the behavior of faked objects
- Time-consuming for complicated interactions
- Require more maintenance when the code changes

9.4.2 Mock-objects evaluation

The greatest advantage of mock objects over in-container testing is that mocks do not require a running container to execute tests. Tests can be set up and quickly run. The main drawback is that the tested components do not run in the container in which we will deploy them. The tests cannot verify the interaction between components and the container. Also, the tests do not test the interactions among the components themselves as they run in the container.

We still need a way to perform integration tests—to check that modules developed by different developers or teams work together. Writing and running functional tests could achieve this goal. The problem with functional tests is that they are coarse-grained and test only a full use case; we lose the benefits of fine-grained unit testing. We will not be able to test as many cases with functional tests as we can with unit tests.

Mock objects have other disadvantages. There are many mock objects to set up, for example, which may prove to be non-negligible overhead – the operating costs of managing all these mocks may be significant. Obviously, the cleaner the code (small and focused methods) is, the easier tests are to set up.

Another important drawback of mock objects is that to set up a test, we usually must know exactly how the mocked API behaves, which may require some knowledge of a domain outside our own. We know the behavior of our own API, but that may not be the case with other APIs, such as the Servlet API that the engineers at Tested Data Systems need to use for the online shop.

Even though all containers of a given type implement the same API, not all containers behave the same way, so we may have to deal with bugs, tricks, and hacks for various third-party libraries in any project.

To wrap up this section, here are the advantages and drawbacks of unit testing with mock objects.

Advantages:

- Do not require a running container to execute tests
- Are quick to set up and run
- Allow fine-grained unit testing

Drawbacks:

- Do not test interactions with the container or between the components
- Do not test the deployment of components
- Require good knowledge of the API to mock, which can be difficult (especially for external libraries)
- Do not provide confidence that the code will run in the target container
- Offer more fine-grained testing, which may lead to testing code being swamped with interfaces
- Like stubs, require maintenance when the code changes

9.4.3 In-container testing evaluation

So far, we have examined the advantages of in-container unit testing. This approach also has the disadvantages discussed in the following sections.

SPECIFIC TOOLS REQUIRED

A major drawback is that although the concept is generic, the tools that implement in-container unit testing are specific to the tested API, such as Jetty or Tomcat for servlets and WildFly for EJBs. With mock objects, because the concept is generic, you can test almost any API.

NO GOOD IDE SUPPORT

A significant drawback of most in-container testing frameworks is the lack of good IDE integration. In most cases, you can use Maven or Gradle to execute tests in an embedded container, which also allows you to run a build in a Continuous Integration Server (CIS; see chapter 13). Alternatively, IDEs can execute tests that use mock objects as normal JUnit tests.

In-container testing falls into the category of integration testing, which means that you do not need to execute your in-container tests as often as normal unit tests and will most likely run them in a CIS, alleviating the need for IDE integration.

LONGER EXECUTION TIME

Another issue is performance. For a test to run in a container, you need to start and manage the container, which can be time-consuming. The overhead in time and memory depends on the container. Startup overhead is not limited to the container. If a unit test hits a database, for example, the database must be in an expected state before the test starts. In terms of execution time, integration unit tests cost more than mock objects; consequently, you may not run them as often as unit tests.

COMPLEX CONFIGURATION

The biggest drawback of in-container testing is that the tests are complex to configure. Because the application and its tests run in a container, your application must be packaged (usually, as a .war or .ear file) and deployed to the container. Then you must start the container and run the tests.

On the other hand, because you must perform the same tasks for production, it is a best practice to automate this process as part of the build and reuse it for testing purposes. As one of the most complex tasks of a Java EE project, providing automation for packaging and deployment becomes a win-win situation. The need to provide in-container testing drives the creation of this automated process at the beginning of the project, which facilitates continuous integration.

To further this goal, most in-container testing frameworks include support for build tools such as Maven and Gradle and allow the embedded startup of the container. This support helps hide the complexity involved in building various run-time artifacts, running tests, and gathering reports.

Taking into consideration the evaluation we have provided for in-container testing, the next section introduces Arquillian as a container-agnostic integration testing framework for Java EE.

9.5 Testing with Arquillian

Arquillian is a testing framework for Java that leverages JUnit to execute test cases against a Java container. The Arquillian framework is broken \into three major sections:

- Test runners (JUnit, in our case)
- Containers (WildFly, Tomcat, Glassfish, Jetty, and so on)
- Test enrichers (the injection of container resources and beans directly into the test class)

Despite the lack of integration with JUnit 5 (Arquillian does not have a JUnit 5 extension, at least when this chapter was written), it is very popular and has been largely adopted in projects up to JUnit 4. It greatly eases the task of managing containers, deployments, and framework initializations.

On the other side, as Arquillian is testing Java EE applications, its use in our examples requires some basic knowledge, in particular about CDI (Contexts and Dependency Injection), a Java EE standard for the inversion of control design pattern.

ShrinkWrap is an external dependency used with Arquillian and a simple way to create archives in Java. Using the fluent ShrinkWrap API, developers at Tested Data Systems can assemble .jar, .war, and .ear files to be deployed directly by Arquillian during testing. Such files are archives that may contain all classes needed to run an application. ShrinkWrap helps define the deployments and the descriptors to be loaded to the Java container being tested against.

One project under development at Tested Data Systems is a flight-management application. The application creates and sets up flights; it also adds and removes passengers. The engineers want to test the integration between two classes: Passenger and Flight. The test will check whether each passenger can be added to or removed from a flight correctly, as well as whether the number of passengers exceeds the number of seats. (Nobody wants to get on a flight without having a seat!) The following two listings show the Passenger and Flight classes, as well as their logic.

Listing 9.3 The Passenger class

```
public class Passenger {  
  
    private String identifier; #A  
    private String name; #A  
  
    public Passenger(String identifier, String name) { #B  
        this.identifier = identifier;  
        this.name = name;  
    }  
  
    public String getIdentifier() { #C  
        return identifier;  
    }  
  
    public String getName() { #C  
        return name;  
    }  
  
    @Override  
    public String toString() { #D  
        return "Passenger " + getName() +  
               " with identifier: " + getIdentifier();  
    }  
}
```

In this listing:

- We provide two fields—`identifier` and `name`—to describe a `Passenger` (#A).
- We provide a constructor with these two parameters (#B).
- We provide two getters (#C) and the overridden `toString` method (#D).

Listing 9.4 The Flight class

```
public class Flight {  
  
    private String flightNumber; #A  
    private int seats; #A  
    Set<Passenger> passengers = new HashSet<>(); #A  
  
    public Flight(String flightNumber, int seats) { #B  
        this.flightNumber = flightNumber;  
        this.seats = seats;  
    }  
  
    public String getFlightNumber() { #C
```

```

        return flightNumber;
    }

    public int getSeats() { #C
        return seats;
    }

    public int getNumberOfPassengers () { #C
        return passengers.size();
    }

    public void setSeats(int seats) { #D
        if(passengers.size() > seats) {
            throw new RuntimeException(
                "Cannot reduce seats under the number of existing passengers!");
        }
        this.seats = seats;
    }

    public boolean addPassenger(Passenger passenger) { #E
        if(passengers.size() >= seats) {
            throw new RuntimeException(
                "Cannot add more passengers than the capacity of the flight!");
        }
        return passengers.add(passenger);
    }

    public boolean removePassenger(Passenger passenger) { #F
        return passengers.remove(passenger);
    }

    @Override
    public String toString() { #G
        return "Flight " + getFlightNumber();
    }
}

```

In this listing:

- We provide three fields—`flightNumber`, `seats`, and `passengers` — to describe a Flight (#A.)
- We provide a constructor with the first two mentioned parameters (#B).
- We provide three getters (#C) and one setter (#D) to address the fields we have defined. The setter on the `seats` field checks that the value of the field does not arrive under the number of existing passengers.
- The `addPassenger` method adds a passenger to the flight. It also compares the number of passengers with the number of seats so the flight will not be overbooked. (#E).
- We provide the capability to remove a passenger from a flight (#F).
- We override the `toString` method (#G).

Listing 9.5 describes the 20 passengers on the flight by identifier and name. The list is stored in a CSV file.

Listing 9.5 The flights_information.csv file

```
1236789; John Smith
9006789; Jane Underwood
1236790; James Perkins
9006790; Mary Calderon
1236791; Noah Graves
9006791; Jake Chavez
1236792; Oliver Aguilar
9006792; Emma McCann
1236793; Margaret Knight
9006793; Amelia Curry
1236794; Jack Vaughn
9006794; Liam Lewis
1236795; Olivia Reyes
9006795; Samantha Poole
1236796; Patricia Jordan
9006796; Robert Sherman
1236797; Mason Burton
9006797; Harry Christensen
1236798; Jennifer Mills
9006798; Sophia Graham
```

Listing 9.6 implements the `FlightUtilBuilder` class, which parses the CSV file and populates the flight with the corresponding passengers. Thus, the code brings the information from an external file to the memory of the application.

Listing 9.6 The FlightUtilBuilder class

```
public class FlightBuilderUtil {

    public static Flight buildFlightFromCsv() throws IOException {
        Flight flight = new Flight("AA1234", 20);                                #A
        try(BufferedReader reader =
            new BufferedReader(new FileReader(
                "src/test/resources/flights_information.csv"))){                  #B
            {
                String line = null;
                do {
                    line = reader.readLine();                                         #C
                    if (line != null) {
                        String[] passengerString = line.toString().split(";");   #D
                        Passenger passenger =
                            new Passenger(passengerString[0].trim(),           #E
                                            passengerString[1].trim());          #E
                        flight.addPassenger(passenger);                           #F
                    }
                } while (line != null);
            }
            return flight;                                                       #G
        }
    }
}
```

In this listing:

- We create a flight (#A).
- We open the CSV file to parse (#B).
- We read line by line (#C), split each line (#D,) create a passenger based on the information that has been read (#E), and add the passenger to the flight (#F).
- We return the fully populated flight from the method (#G).

So far, all classes that have been implemented for the development of the flight and passengers are pure Java classes; no particular framework and technology have been used so far. As a testing framework that executes test cases against a Java container, Arquillian requires some understanding of notions related to Java EE and CDI. As it is a widespread framework for integration testing, we have decided to introduce it here and we'll try to explain the most important ideas so that you can quickly adopt it within your projects.

Arquillian abstracts the container or application startup logic from the unit tests; instead, it drives a deployment run-time paradigm with the application, allowing the deployment of the program to a Java EE application server. Therefore, the framework is ideal for the in-container testing implemented in this chapter. In addition, it eliminates the need for specific tools that implement in-container testing.

Arquillian deploys the application to the targeted run time to execute test cases. The targeted run time can be an application server, embedded or managed.

Listing 9.7 shows the dependencies that we need to add to the Maven pom.xml configuration file to work with Arquillian.

Listing 9.7 Required pom.xml dependencies

```
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.jboss.arquillian</groupId>                                #A
            <artifactId>arquillian-bom</artifactId>                                #A
            <version>1.4.0.Final</version>
            <scope>import</scope>
            <type>pom</type>
        </dependency>
    </dependencies>
</dependencyManagement>
<dependencies>
    <dependency>
        <groupId>org.jboss.spec</groupId>                                         #B
        <artifactId>jboss-javaee-7.0</artifactId>                                #B
        <version>1.0.3.Final</version>
        <type>pom</type>
        <scope>provided</scope>
    </dependency>
    <dependency>
        <groupId>org.junit.vintage</groupId>                                         #C
        <artifactId>junit-vintage-engine</artifactId>                            #C
        <version>5.6.0</version>
        <scope>test</scope>
    </dependency>
    <dependency>
        <groupId>org.jboss.arquillian.junit</groupId>                            #D
    </dependency>
```

```

<artifactId>arquillian-junit-container</artifactId> #D
<scope>test</scope>
</dependency>
<dependency>
    <groupId>org.jboss.arquillian.container</groupId> #E
    <artifactId>arquillian-weld-ee-embedded-1.1</artifactId> #E
    <version>1.0.0.CR9</version>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.jboss.weld</groupId> #F
    <artifactId>weld-core</artifactId> #F
    <version>2.3.5.Final</version>
    <scope>test</scope>
</dependency>
</dependencies>

```

This listing adds

- The Arquillian API dependency (#A).
- The Java EE 7 API dependency (#B).
- The JUnit Vintage Engine dependency (#C). As mentioned earlier, at least for the moment, Arquillian is not yet integrated with JUnit 5. Because Arquillian lacks a JUnit 5 extension, we'll have to use the JUnit 4 dependencies and annotations to run our tests.
- The Arquillian JUnit integration dependency (#D).
- The container adapter dependencies (#E and #F). To execute our tests against a container, we must include the dependencies that correspond to that container. This requirement demonstrates one of the strengths of Arquillian: it abstracts the container from the unit tests and is not tightly coupled to specific tools that implement in-container unit testing.

An Arquillian test, as implemented in listing 9.8, looks just like a unit test, with some additions. The test is named `FlightWithPassengersTest` to show the goal of the integration testing between the two classes.

Listing 9.8 The FlightWithPassengersTest class

```

@RunWith(Arquillian.class) #A
public class FlightWithPassengersTest {

    @Deployment #B
    public static JavaArchive createDeployment() { #B
        return ShrinkWrap.create(JavaArchive.class) #B
            .addClasses(Passenger.class, Flight.class) #B
            .addAsManifestResource(EmptyAsset.INSTANCE, "beans.xml"); #B
    }

    @Inject #C
    Flight flight; #C

    @Test(expected = RuntimeException.class) #D
    public void testNumberOfSeatsCannotBeExceeded() throws IOException { #D
        assertEquals(20, flight.getNumberOfPassengers()); #D
    }
}

```

```

        flight.addPassenger(new Passenger("1247890", "Michael Johnson"));
    }

    @Test
    public void testAddRemovePassengers() throws IOException { #E
        flight.setSeats(21);
        Passenger additionalPassenger =
            new Passenger("1247890", "Michael Johnson");
        flight.addPassenger(additionalPassenger);
        assertEquals(21, flight.getNumberOfPassengers());
        flight.removePassenger(additionalPassenger);
        assertEquals(20, flight.getNumberOfPassengers());
        assertEquals(21, flight.getSeats());
    }
}

```

As this listing shows, an Arquillian test case must have three things:

- A `@RunWith(Arquillian.class)` annotation on the class (#A). The `@RunWith` annotation tells JUnit to use Arquillian as the test controller.
- A public static method annotated with `@Deployment` that returns a ShrinkWrap archive (#B). The purpose of the test archive is to isolate the classes and resources that the test needs. The archive is defined with ShrinkWrap. The micro deployment strategy lets us focus on precisely the classes we want to test. As a result, the test remains very lean and manageable. For the moment, we have included only the Passenger and the Flight classes. We try to inject a Flight object as a class member, using the CDI `@Inject` annotation (#C). The `@Inject` annotation allows us to define injection points inside classes. In this case, `@Inject` instructs CDI to inject into the test a field of type reference to a Flight object.
- At least one method annotated with `@Test` (#D and #E). Arquillian looks for a public static method annotated with the `@Deployment` annotation to retrieve the test archive. Then each `@Test` annotated method is run inside the container environment.

When the ShrinkWrap archive is deployed to the server, it becomes a real archive. The container has no knowledge that the archive was packaged by ShrinkWrap.

We have provided the infrastructure for using Arquillian in our project, we'll run our integration tests with its help! If we run the tests now, we are going to get an error (figure 9.2).

```
org.jboss.weld.exceptions.DeploymentException: WELD-001408: Unsatisfied dependencies for type Flight with qualifiers @Default
at injection point [BackedAnnotatedField] @Inject com.manning.junitbook.ch09.airport.FlightWithPassengersTest.flight
at com.manning.junitbook.ch09.airport.FlightWithPassengersTest.flight(FlightWithPassengersTest.java:0)
```

Figure 9.2 The result of running FlightWithPassengersTest

The error says Unsatisfied dependencies for type Flight with qualifiers `@Default`. It means that the container is trying to inject the dependency, as it has been instructed through the CDI `@Inject` annotation, but it is unsatisfied. Why? What have the

developers from Tested Data Systems missed? The `Flight` class provides only a constructor with arguments, and it has no default constructor to be used by the container for the creation of the object. The container does not know how to invoke the constructor with parameters and which parameters to pass to it to create the `Flight` object that must be injected.

What is the solution in this case? Java EE offers the producer methods that are designed to inject objects that require custom initialization (listing 9.9). The solution fixes the issue and is easy to put into practice, even by a junior developer.

Listing 9.9 The FlightProducer class

```
public class FlightProducer {  
  
    @Produces  
    public Flight createFlight() throws IOException {  
        return FlightBuilderUtil.buildFlightFromCsv();  
    }  
}
```

In this listing, we create the `FlightProducer` class with the `createFlight` method, which invokes `FlightBuilderUtil.buildFlightFromCsv()`. We can use such a method to inject objects that require custom initialization, and in this case, we are injecting a flight that has been configured based on the CSV file. We annotated the `createFlight` method with `@Produces`, which is also a Java EE annotation. The container will automatically invoke this method to create the configured flight; then it injects the method into the `Flight` field, annotated with `@Inject` from the `FlightWithPassengersTest` class.

Listing 9.10 adds the `FlightProducer` class to the `ShrinkWrap` archive.

Listing 9.10 The modified deployment method from the FlightWithPassengersTest class

```
@Deployment  
public static JavaArchive createDeployment() {  
    return ShrinkWrap.create(JavaArchive.class)  
        .addClasses(Passenger.class, Flight.class,  
                   FlightProducer.class)  
        .addAsManifestResource(EmptyAsset.INSTANCE, "beans.xml");  
}
```

If we run the tests now, they will be green. The container injected the correctly configured flight (figure 9.3).

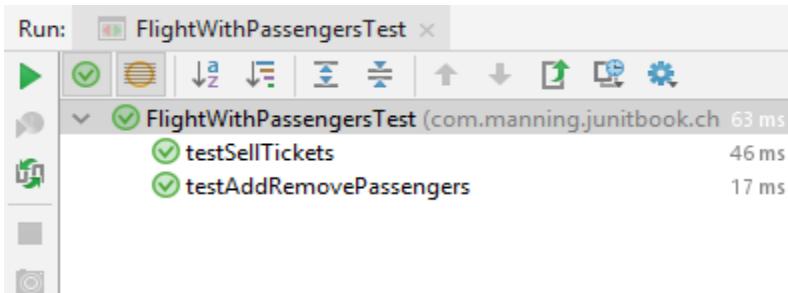


Figure 9.3 The result of running `FlightPassengersTest` after introducing the producer method

Considering the advantages of in-container testing, which eliminates tight coupling with specific tools, it is likely to receive more attention and applicability inside Tested Data Systems. The developers are waiting for one more thing: the creation of a JUnit 5 extension. (We are looking forward to it as well, so fingers crossed!)

Chapter 10 starts the third part of this book by taking a deeper look at the integration of JUnit into the build process with Maven.

9.6 Summary

This chapter has covered the following:

- Analyzing the limitations of unit testing, including limitations given by the usage of mock objects: cannot use objects whose implementation is provided by a container; require a lot of code; the test expectations must continuously change to match the code evolution; do not provide a running isolated environment to run.
- Examining the need for and the step to in-container testing: running the test cases where the objects actually live, in the container itself.
- Evaluating the testing using stubs, mock objects, and in-container testing, comparing the advantages and drawbacks of each of them.
- Working with Arquillian, a container-agnostic integration testing framework for Java EE and implementing integration tests with its help. Despite its current lack of integration with JUnit 5 (it is still missing a JUnit 5 extension), it is used for many years into Java EE projects, as it leverages JUnit to execute test cases against a Java container.

10

Running JUnit tests from Maven 3

This chapter covers

- Creating a Maven project from the scratch
- Testing the Maven project with JUnit 5
- Using Maven plugins
- Using the Maven Surefire Plugins

The conventional view serves to protect us from the painful job of thinking.

—John Kenneth Galbraith

This chapter discusses a very common build system tool called Maven. In the previous chapters, with provided Maven projects, you needed only to look at some external dependencies, run some very simple commands, or run the tests from inside the IDE. This chapter gives you a brief introduction to the Maven build system, which will be very useful if you need a systematic way to start your tests.

Maven addresses two aspects of building software. First, it describes how software is built; then it describes the needed dependencies. Unlike earlier tools, such as Apache Ant, it uses conventions for the build procedure, and only exceptions need to be written down. It relies on an .xml file to describe its full configuration—most important here the meta-information about the software project being built, the needed dependencies on other external components, and the required plugins.

At the end of this chapter, you will know how to build Java projects with Maven, including managing their dependencies, executing JUnit tests, and generating JUnit reports.

For the basic Maven concepts and how to set it up, please have a look at Appendix A.

10.1 Setting up a Maven project

If we have installed Maven, we are ready to use it. The first time we execute a plugin, our Internet connection must be on, because Maven automatically downloads from the web all the third-party .jar files that the plugin requires.

First, create the `C:\junitbook\` folder. This directory is our work directory and where we will set up the Maven examples. Type the following on the command line:

```
mvn archetype:generate -DgroupId=com.manning.junitbook  
-DartifactId=maven-sampling  
-DarchetypeArtifactId=maven-artifact-mojo
```

After we press Enter, wait for the appropriate artifacts to be downloaded, and accept the default options, we should see a folder named `maven-sampling` being created. If we open the new project into IntelliJ IDEA, its structure should look like figure 10.1.

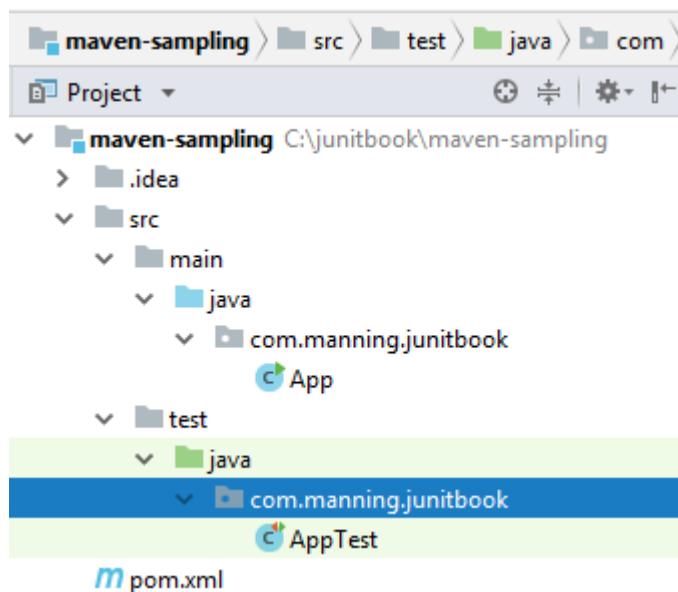


Figure 10.1 Folder structure after the project is created

What happened here? We invoked the `maven-archetype-plugin` from the command line and told it to generate a new project from scratch with the given parameters. As a result, this Maven plugin created a new project with a new folder structure, following the convention of the folder structure. Further, it created a sample `App.java` class with the `main` method and a corresponding `AppTest.java` file that is a unit test for our application. After looking at this folder structure, you should understand what files stay in `src/main/java` and what files stay in `src/test/java`.

The Maven plugin also generated a pom.xml file for us. Listing 10.1 explains some parts of the descriptor.

Listing 10.1 pom.xml for the maven-sampling project

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  http://maven.apache.org/maven-v4_0_0.xsd">

  <modelVersion>4.0.0</modelVersion>
  <groupId>com.manning.junitbook</groupId>
  <artifactId>maven-sampling</artifactId>
  <version>1.0-SNAPSHOT</version>
  <name>maven-sampling</name>
  <!-- FIXME change it to the project's website -->
  <url>http://www.example.com</url>
  [...]
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.11</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
  [...]
</project>
```

This code is the build descriptor for our project. It starts with a global `<project>` tag with the appropriate namespaces. Inside it, we place all our components:

- `modelVersion`—Represents the version of the model of the pom being used. Currently, the only supported version is 4.0.0.
- `groupId`—Acts as the Java packaging in the file system, grouping different projects from one organization, company, or group of people. We provide this value in the command line when invoking Maven.
- `artifactId`—Represents the name that the project is known by. Again, the value here is the one we specified in the command line.
- `version`—Identifies the current version of our project (or project artifact). The SNAPSHOT ending indicates that this artifact is still in development mode; we have not released it yet.
- `dependencies`—Lists our dependencies.

Now that we have our project descriptor, we can improve it a little bit (listing 10.2). First, we need to change the version of the JUnit dependency, because we are using JUnit Jupiter 5.4.2, and the one that the plugin generated is 4.11. After that, we can insert some additional info to make the pom.xml more descriptive, such as a `developers` section. This information not only makes the pom.xml more descriptive but also will be included later when we build the website.

Listing 10.2 Changes and additions to the pom.xml

```
<dependencies>
    <dependency>
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter-api</artifactId>
        <version>5.4.2</version>
        <scope>test</scope>
    </dependency>
    <dependency>
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter-engine</artifactId>
        <version>5.4.2</version>
        <scope>test</scope>
    </dependency>
</dependencies>

<developers>
    <developer>
        <name>Catalin Tudose</name>
        <id>ctudose</id>
        <organization>Manning</organization>
        <roles>
            <role>Java Developer</role>
        </roles>
    </developer>
    <developer>
        <name>Petar Tahchiev</name>
        <id>ptahchiev</id>
        <organization>Apache Software Foundation</organization>
        <roles>
            <role>Java Developer</role>
        </roles>
    </developer>
</developers>
```

We also specify organization, description and inceptionYear:

Listing 10.3 Description elements to the pom.xml

```
<description>
    “JUnit in Action III” book, the sample project for the “Running JUnit
    tests from Maven” chapter.
</description>
<organization>
    <name>Manning Publications</name>
    <url>http://manning.com/</url>
</organization>
<inceptionYear>2019</inceptionYear>
```

Now we can start developing our software. What if we want to use a Java IDE other than IntelliJ IDEA, such as Eclipse? No problem. Maven offers additional plugins that let us import the project into our favorite IDE. If we want to use Eclipse, we open a terminal and navigate to the directory that contains the project descriptor (pom.xml). Then we type the following and press Enter:

```
mvn eclipse:eclipse
```

©Manning Publications Co. To comment go to [liveBook](#)

This command invokes the `maven-eclipse-plugin`, which, after downloading the necessary artifacts, produces the two files (`.project` and `.classpath`) that Eclipse needs to recognize our project as an Eclipse project. Now we can import our project into Eclipse. All the dependencies listed in the `pom.xml` file have been added to the project (figure 10.2).

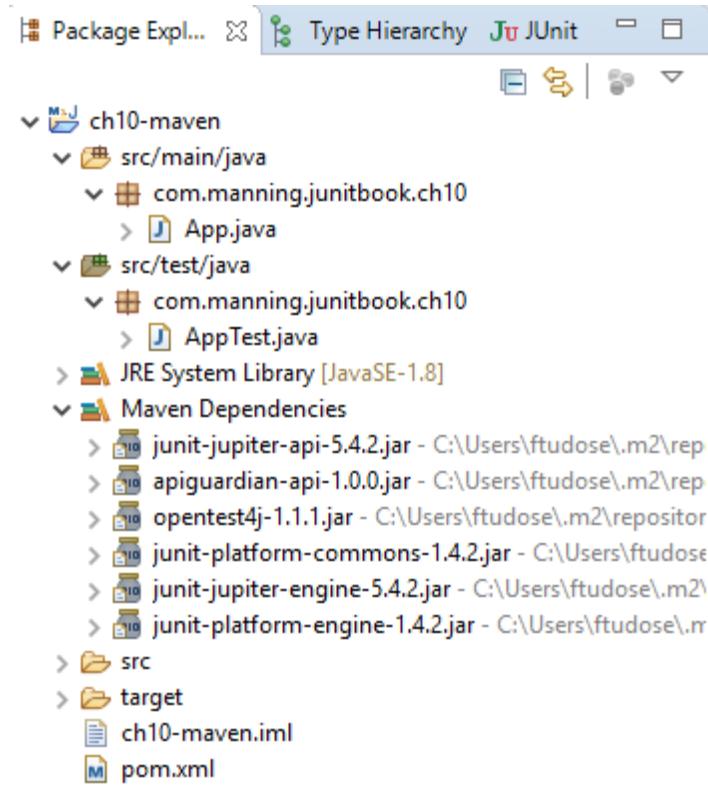


Figure 10.2 The imported project, with the Maven folders structure and needed dependencies (including the JUnit 5 ones)

Developers who use IntelliJ IDEA can import the project directly (refer to figure 10.1), as this IDE invokes the Maven plugin when such a project is open.

10.2 Using the Maven plugins

We have seen what Maven is and how to use it to start a project from scratch. We have also seen how to generate the project documentation and how to import our project in Eclipse and in IntelliJ.

Whenever we would like to clean the project from the previous activities, we may execute the following command:

```
mvn clean
```

This command causes Maven to go through the clean phase and invokes all the plugins that are attached to this phase — in particular, the `maven-clean-plugin`, which deletes the `target/` folder, where our generated site resides.

10.2.1 Maven compiler plugin

Like any other build system, Maven is supposed to build our projects (compile our software and package in an archive). Every task in Maven is performed by an appropriate plugin, the configuration of which is in the `<plugins>` section of the project descriptor. To compile the source code, all we need to do is invoke the `compile` phase on the command line

```
mvn compile
```

which causes Maven to execute all the plugins attached to the `compile` phase (in particular it will invoke the `maven-compiler-plugin`). But before invoking the `compile` phase, as already discussed, Maven goes through the validate phase, downloads all the dependencies listed in the `pom.xml` file, and includes them in the classpath of the project. When the compilation process is complete, we can go to the `target/classes/` folder and see the compiled classes there.

Next, we'll try to configure the compiler plugin, escaping from the convention-over-configuration principle.

So far, the conventional compiler plugin has worked well. But what if we need to include the `-source` and `-target` attributes in the compiler invocation to generate class files for the specific version of the JVM? We should add the following code to the `<build>` section of our build file.

Listing 10.4 Configuring the maven-compiler-plugin

```
<build>
  <plugins>
    <plugin>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>2.3.2</version>
      <configuration>
        <source>1.8</source>
        <target>1.8</target>
      </configuration>
    </plugin>
  </plugins>
</build>
```

This code is a general way to configure each of our Maven plugins: enter a `<plugins>` section in our `<build>` section. There, we list each plugin that we want to configure—in this case, the `maven-compiler-plugin`. We need to enter the configuration parameters in the `plugin configuration` section. We can get a list of parameters for every plugin from the

Maven website. Without the `<source>` and `<target>` parameters, the Java version to be used will be 5, which is quite old...

As we see in the declaration of the `maven-compiler-plugin` in listing 10.4, we have not set the `groupId` parameter. `maven-compiler-plugin` is one of the core Maven plugins that has a `org.apache.maven.plugins` `groupId`, and plugins with such a `groupId` can skip the `groupId` parameter.

10.2.2 Maven surefire plugin

To process the unit tests from our project, Maven uses (of course) a plugin. The Maven plugin that executes the unit tests is called `maven-surefire-plugin`. The `surefire` plugin executes the unit tests for our code, but these unit tests are not necessarily JUnit tests.

There are other frameworks for unit testing, and the `surefire` plugin can execute their tests, too. Listing 10.5 shows the configuration of the Maven `surefire` plugin.

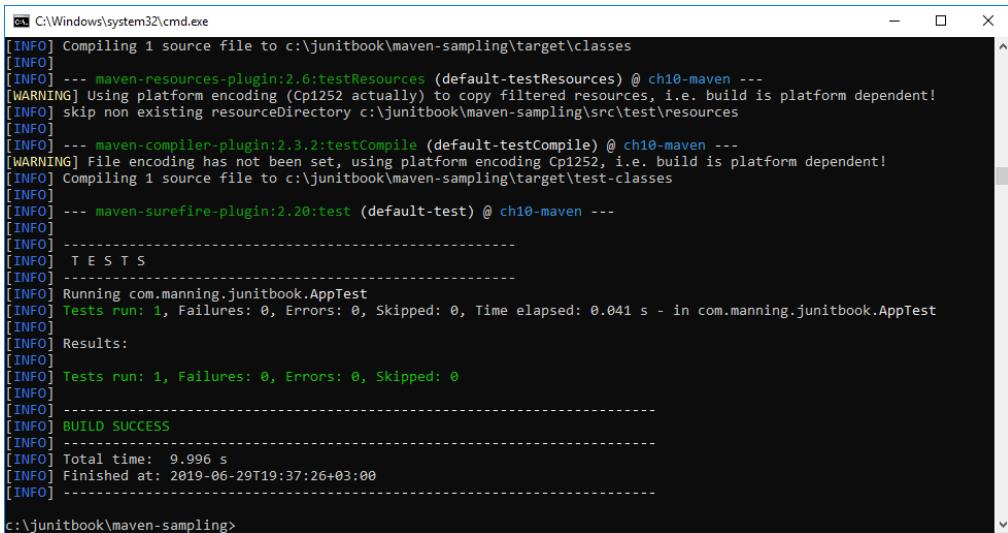
Listing 10.5 The maven-surefire-plugin

```
<build>
  <plugins>
    <plugin>
      <artifactId>maven-surefire-plugin</artifactId>
      <version>2.22.2</version>
    </plugin>
  </plugins>
</build>
```

The conventional way to start the `maven-surefire-plugin` is very simple: invoke the test phase of Maven. This way, Maven first invokes all the phases that are supposed to come before the test phase (validate and compile) and then invokes all the plugins that are attached to the test phase, this way invoking the `maven-surefire-plugin`. So by calling

```
mvn clean test
```

Maven first cleans the `target/` directory, then compiles the source code and the tests, and finally lets JUnit 5 execute all the tests that are in the `src/test/java` directory (remember the convention). The output should be similar to figure 10.3.



```
C:\Windows\system32\cmd.exe
[INFO] Compiling 1 source file to c:\junitbook\maven-sampling\target\classes
[INFO]
[INFO] --- maven-resources-plugin:2.6:testResources (default-testResources) @ ch10-maven ---
[WARNING] Using platform encoding (Cp1252 actually) to copy filtered resources, i.e. build is platform dependent!
[INFO] skip non existing resourceDirectory c:\junitbook\maven-sampling\src\test\resources
[INFO]
[INFO] --- maven-compiler-plugin:2.3.2:testCompile (default-testCompile) @ ch10-maven ---
[WARNING] File encoding has not been set, using platform encoding Cp1252, i.e. build is platform dependent!
[INFO] Compiling 1 source file to c:\junitbook\maven-sampling\target\test-classes
[INFO]
[INFO] --- maven-surefire-plugin:2.20:test (default-test) @ ch10-maven ---
[INFO]
[INFO] -----
[INFO] T E S T S
[INFO] -----
[INFO] Running com.manning.junitbook.AppTest
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.041 s - in com.manning.junitbook.AppTest
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] -----
[INFO] BUILD SUCCESS
[INFO]
[INFO] Total time: 9.996 s
[INFO] Finished at: 2019-06-29T19:37:26+03:00
[INFO]
c:\junitbook\maven-sampling
```

Figure 10.3 Execution of JUnit tests with Maven3

That's great, but what if we want to execute only a single test case? This execution is unconventional, so we need to configure the `maven-surefire-plugin` to do it. Ideally, a parameter for the plugin allows us to specify the pattern of test cases that we want to execute. We configure the surefire plugin in absolutely the same way that we configure the compiler plugin, as shown in listing 10.6.

Listing 10.6 Configuration of the `maven-surefire-plugin`

```
<build>
  <plugins>
    [...]
    <plugin>
      <artifactId>maven-surefire-plugin</artifactId>
      <version>2.22.2</version>
      <configuration>
        <includes>**/*Test.java</includes>
      </configuration>
    [...]
  </plugin>
  [...]
</plugins>
</build>
```

We have specified the `includes` parameter to denote that we want only the test cases matching the given pattern to be executed. But how do we know what parameters the `maven-surefire-plugin` accepts? No one knows all the parameters by heart, of course, but we can always consult the `maven-surefire-plugin` documentation (and any other plugin documentation) on the Maven website (<http://maven.apache.org>).

The next step is generating some documentation for the project. But wait for a second—how are we supposed to do that with no files to generate the documentation from? This is another one of Maven’s great benefits: with a little bit of configuration and description that we have, we can produce a fully functional website skeleton.

First, add the `maven-site-plugin` to the Maven `pom.xml` configuration file:

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-site-plugin</artifactId>
  <version>3.7.1</version>
</plugin>
```

Then type

```
mvn site
```

on the command line where our `pom.xml` file is. Maven should start downloading its plugins, and after their successful installation, it produces the nice website shown in figure 10.4.



Figure 10.4 Maven produces nice website documentation for the project.

This website is generated in the Maven build directory—another convention. Maven uses the `target/` folder for all the needs of the build itself. The source code is compiled in the `target/classes/` folder, and the documentation is generated in `target/site/`.

After we examine the project, we probably notice that this website is more like a skeleton of a website. That is absolutely true. Remember that we entered a small amount of data in the first place. We could enter more data and web pages in the `src/site`, and Maven would include it on the website, thus generating full-blown documentation.

10.2.3 HTML JUnit reports with Maven

Maven can generate nice reports from the JUnit XML output. Because by default, Maven produces plain and XML-formatted output (by convention, they go in the `target/surefire-reports` folder), we do not need any other configuration to produce HTML surefire reports for the JUnit tests.

As you already guess, the job of producing these reports is done by a Maven plugin. The name of the plugin is `maven-surefire-report-plugin`, and by default, it is not attached

to any of the core phases that we already know. (Many people do not need HTML reports every time they build software.) We can't invoke the plugin by running a certain phase (as we did with both the compiler plugin and the surefire plugin). Instead, we have to call it directly from the command line:

```
mvn surefire-report:report
```

Maven tries to compile the source files and the test cases; then it invokes the surefire plugin to produce the plain-text and XML-formatted output of the tests. After that, the surefire-report plugin tries to transform all the XMLs from the target/surefire-reports/ directory into an HTML report that will be placed in the target/site directory. (Remember that this is the convention for the folder—to keep all the generated documentation of the project—and that the HTML reports are considered to be documentation.)

If we try to open the generated HTML report, it should look something like figure 10.5.

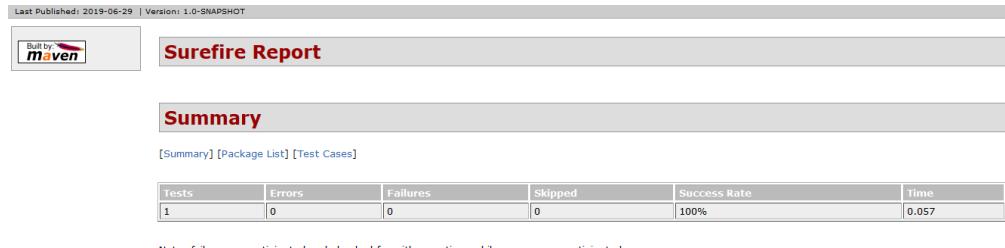


Figure 10.5. HTML report from the maven-surefire-report plugin

10.3 Putting it all together

This section demonstrates all the steps involved in creating a JUnit 5 project that is managed by Maven.

As discussed in chapter 9, one of the projects under development at Test It is a flight-management application. To start such an application, George, a project developer, creates the C:\Work\ folder, which will become the working directory and the one where he will set up the Maven project. He types the following on the command line:

```
mvn archetype:generate -DgroupId=com.testeddatasystems.flights  
-DartifactId=flightsmanagement -DarchetypeArtifactid=maven-artifact-mojo
```

After pressing Enter and waiting for the appropriate artifacts to be downloaded, George accepts the default options (figure 10.6). Figure 10.7 shows the structure of the project: src/main/java/ is the Maven folder in which the Java code for the project resides, and src/main/test/ is the unit tests.

```
C:\WINDOWS\system32\cmd.exe - mvn archetype:generate -DgroupId=com.testeddatasystems.flights -DartifactId=flightsmanagement -DarchetypeArtifactId=maven-artifact-mojo
2525: remote -> us.fatehi:schemacrawler-archetype-plugin-command (-)
2526: remote -> us.fatehi:schemacrawler-archetype-plugin-dbconnector (-)
2527: remote -> us.fatehi:schemacrawler-archetype-plugin-lint (-)
2528: remote -> ws.osiris:osiris-archetype (Maven Archetype for Osiris)
2529: remote -> xyz.luan.generator:xyz-gae-generator (-)
2530: remote -> xyz.luan.generator:xyz-generator (-)
2531: remote -> za.co.absa.hyperdrive:component-archetype (-)
Choose a number or apply filter (format: [groupId:]artifactId, case sensitive contains): 1449:
Choose org.apache.maven.archetypes:maven-archetype-quickstart version:
1: 1.0-alpha-1
2: 1.0-alpha-2
3: 1.0-alpha-3
4: 1.0-alpha-4
5: 1.0
6: 1.1
7: 1.3
8: 1.4
Choose a number: 8:
[INFO] Using property: groupId = com.testeddatasystems.flights
[INFO] Using property: artifactId = flightsmanagement
Define value for property 'version' 1.0-SNAPSHOT:
[INFO] Using property: package = com.testeddatasystems.flights
Confirm properties configuration:
groupId: com.testeddatasystems.flights
artifactId: flightsmanagement
version: 1.0-SNAPSHOT
package: com.testeddatasystems.flights
Y: :
```

Figure 10.6 Creating a Maven 3 project and accepting the default options

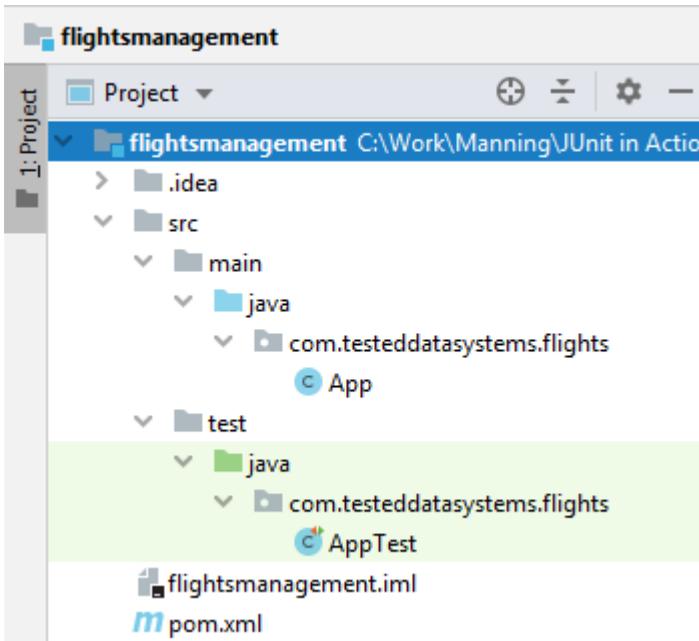


Figure 10.7 The newly created flight-management project

George invoked the maven-archetype-plugin from the command line and told it to generate a new project from scratch with the given parameters. As a result, the Maven plugin created a new project with the default folder structure, as well as a sample `App.java` class with the main method and a corresponding `AppTest.java` file that is a unit test for the application.

The Maven plugin also generated a `pom.xml` file, as shown in listing 10.7.

Listing 10.7 pom.xml for the flightsmanagement project

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  http://maven.apache.org/maven-v4_0_0.xsd">

  <modelVersion>4.0.0</modelVersion>
  <groupId>com.testeddatasystems.flights</groupId>
  <artifactId>flightsmanagement</artifactId>
  <version>1.0-SNAPSHOT</version>
  <name> flightsmanagement </name>
  [...]
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.11</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
  [...]
</project>
```

This code is the build descriptor for the project. It includes the `modelVersion` (the version of the model of the `pom` being used, which currently is 4.0.0), the `groupId` (`com.testeddatasystems.flights`), the `artifactId` (the name by which the project is known—`flightsmanagement` in this case) and the `version`. `1.0-SNAPSHOT` indicates that the artifact is still in development mode.

George needs to change the version of the JUnit dependency (listing 10.8) because he is using JUnit Jupiter 5.6, and the one that the plugin generated is 4.11.

Listing 10.8 Changes and additions to the pom.xml

```
<dependencies>
  <dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-api</artifactId>
    <version>5.6</version>
    <scope>test</scope>
  </dependency>
  <dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-engine</artifactId>
    <version>5.6</version>
    <scope>test</scope>
```

```
</dependency>  
</dependencies>
```

George removes the existing autogenerated App and AppTest classes and instead introduces the two classes that will start his application: Passenger (listing 10.9) and PassengerTest (listing 10.10).

Listing 10.9 The Passenger class

```
public class Passenger {  
  
    private String identifier; #A  
    private String name; #A  
  
    public Passenger(String identifier, String name) { #B  
        this.identifier = identifier; #B  
        this.name = name; #B  
    } #B  
  
    public String getIdentifier() { #C  
        return identifier; #C  
    } #C  
  
    public String getName() { #C  
        return name; #C  
    } #C  
  
    @Override  
    public String toString() { #D  
        return "Passenger " + getName() + #D  
               " with identifier: " + getIdentifier(); #D  
    } #D  
}
```

The Passenger class contains the following:

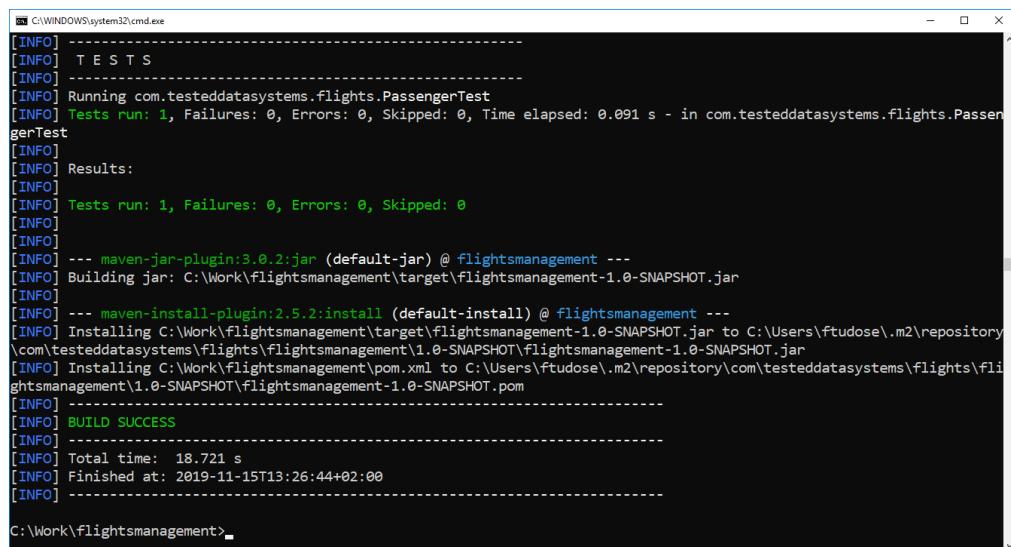
- Two fields: identifier and name (#A)
- A constructor, receiving as arguments the identifier and the name (#B)
- Getters for the identifier and name fields (#C).
- The overridden `toString` method (#D).

Listing 10.10 The PassengerTest class

```
public class PassengerTest {  
  
    @Test  
    void testPassenger() { #A  
        Passenger passenger = new Passenger("123-456-789", "John Smith"); #A  
        assertEquals("Passenger John Smith with identifier: 123-456-789", #A  
                    passenger.toString()); #A  
    } #A  
}
```

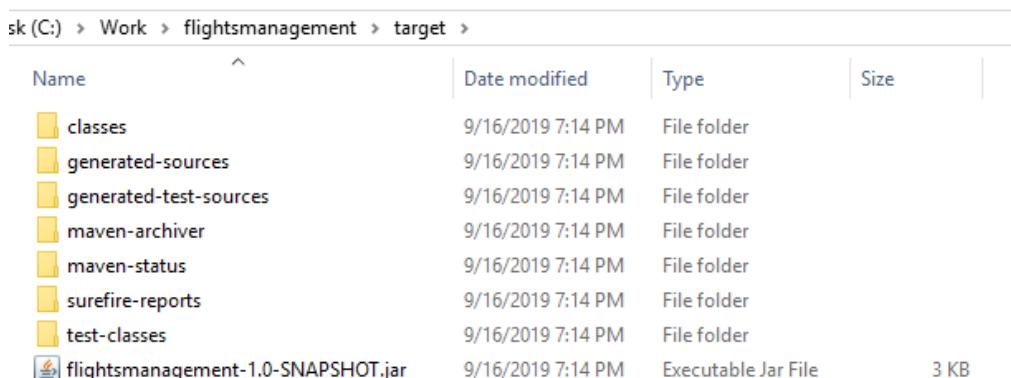
The PassengerTest class contains one test: `testPassenger`, which verifies the output of the overridden `toString` method (#A).

Next, George executes the `mvn clean install` command at the level of the project folder (figure 10.8). This command first cleans the project, removing existing artifacts. Then it compiles the source code of the project, tests the compiled source code (using JUnit 5), packages the compiled code in JAR format (figure 10.9), and installs the package in the local Maven repository (figure 10.10). The local Maven repository is located at `~/.m2/repository/` in UNIX or `C:\Documents and Settings\<UserName>\.m2\repository\` in Windows.



```
[INFO] -----
[INFO] T E S T S
[INFO] -----
[INFO] Running com.testeddatasystems.flights.PassengerTest
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.091 s - in com.testeddatasystems.flights.PassengerTest
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO]
[INFO] --- maven-jar-plugin:3.0.2:jar (default-jar) @ flightsmanagement ---
[INFO] Building jar: C:\Work\flightsmanagement\target\flightsmanagement-1.0-SNAPSHOT.jar
[INFO]
[INFO] --- maven-install-plugin:2.5.2:install (default-install) @ flightsmanagement ---
[INFO] Installing C:\Work\flightsmanagement\target\flightsmanagement-1.0-SNAPSHOT.jar to C:\Users\ftudose\.m2\repository\com\testeddatasystems\flights\flightsmanagement\1.0-SNAPSHOT\flightsmanagement-1.0-SNAPSHOT.jar
[INFO] Installing C:\Work\flightsmanagement\pom.xml to C:\Users\ftudose\.m2\repository\com\testeddatasystems\flights\flightsmanagement\1.0-SNAPSHOT\flightsmanagement-1.0-SNAPSHOT.pom
[INFO]
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 18.721 s
[INFO] Finished at: 2019-11-15T13:26:44+02:00
[INFO] -----
```

Figure 10.8 Executing the `mvn clean install` command for the Maven project under development



Name	Date modified	Type	Size
classes	9/16/2019 7:14 PM	File folder	
generated-sources	9/16/2019 7:14 PM	File folder	
generated-test-sources	9/16/2019 7:14 PM	File folder	
maven-archiver	9/16/2019 7:14 PM	File folder	
maven-status	9/16/2019 7:14 PM	File folder	
surefire-reports	9/16/2019 7:14 PM	File folder	
test-classes	9/16/2019 7:14 PM	File folder	
flightsmanagement-1.0-SNAPSHOT.jar	9/16/2019 7:14 PM	Executable Jar File	3 KB

Figure 10.9 The JAR-packaged file in the Maven `target` folder of the flight-management project

isk (C:) > Users > ftudose > .m2 > repository > com > testit > flights > flightsmanagement > 1.0-SNAPSHOT			
Name	Date modified	Type	Size
_remote.repositories	9/16/2019 7:14 PM	REPOSITORIES File	1 KB
flightsmanagement-1.0-SNAPSHOT.jar	9/16/2019 7:14 PM	Executable Jar File	3 KB
flightsmanagement-1.0-SNAPSHOT.pom	9/16/2019 6:45 PM	POM File	3 KB
maven-metadata-local.xml	9/16/2019 7:14 PM	XML Document	1 KB

Figure 10.10 The local Maven repository containing the flight-management project's artifacts

George has created a fully functional Maven project to develop the flight-management application and to test it with JUnit 5. From now on, he can continue to add classes and tests and to execute the Maven commands to package the application and run the tests.

10.4 Maven challenges

A lot of people who have used Maven agree that it is really easy to start and that the idea behind the project is amazing. Things seem to be more challenging when we need to do something unconventional, however.

What is great about Maven is that it sets up a frame for us and constrains us to think inside that frame—to think the Maven way and do things the Maven way.

In most cases, Maven will not let us execute any nonsense. It restricts us and shows us the way things need to be done. But these restrictions may be a real challenge if we are accustomed to doing things our own way and to having real freedom of choice as build engineers.

Chapter 11 shows how to run JUnit tests with another build automation tool inspired by Maven: Gradle.

10.5 Summary

This chapter has covered the following:

- A very brief introduction to what Maven is and how to use it in a development environment to build our source code.
- Demonstrating how to manage the project dependencies the Maven style, by adding these dependencies to the pom.xml (Project Object Model) in a declarative way.
- Creating a Maven project from the scratch, opening it into the IntelliJ IDEA and testing it using JUnit 5.
- Comparing how to prepare our Maven JUnit 5 project to be used with both Eclipse and IntelliJ IDEA.
- Analyzing Maven plugins, using the compiler plugin, the surefire plugin to process the unit tests from our project, the site plugin to build a site from our project and the

surefire report plugin to build reports for the project.

11

Running JUnit tests from Gradle 6

This chapter covers

- Introducing Gradle
- Setting up a Gradle project
- Using Gradle plugins
- Creating a Gradle project from the scratch and testing it with JUnit 5
- Comparing Gradle and Maven

Mixing one's wines may be a mistake, but old and new wisdom mix admirably.

-Bertold Brecht

In this chapter, we will analyze the last comer to the world of build system tools. Gradle is an open-source build automation system that started from the concepts of Apache Ant and Apache Maven. Instead of the XML form that Apache Maven is using, as you have seen in chapter 10, Gradle introduces a DSL (domain-specific language) based on Groovy for declaring the project configuration.

A domain-specific language is a computer language dedicated to addressing a specific application domain. The idea is to have languages whose purpose is to solve problems belonging to a specific domain. In the case of builds, one of the results of applying the DSL idea is Gradle.

Groovy is a Java-syntax-compatible object-oriented programming language that runs on the JVM (Java Virtual Machine).

11.1 Introducing Gradle

We'll take a look at various aspects of using Gradle to manage the building and testing Java applications, with a focus on the testing.

We have worked with Maven and you know from chapter 10 that this one is using convention over configuration. Gradle has also a series of building conventions that we can follow when we do the build. This allows other developers who are also using Gradle to easily follow our builds configuration.

The conventions may be easily overridden. As we have explained, the build language of Gradle is a Domain Specific Language based on Groovy. This allows the developers to configure the builds in an easy manner. This DSL replaces the XML approach promoted by Maven.

XML has been used for years to store information. It is not surprising that XML has been used to configure the build files of Apache Ant and, then, the ones of Apache Maven, as we have discussed in chapter 10. Maven came with the idea of introducing conventions over configuration, which was lacking in Ant. And we have previously discussed the Maven folders structure, about the support for dependencies and about the Maven build lifecycles.

Gradle has a declarative build language that expresses the intent of the build – this means that you tell what you would like to happen, not how you would like it to happen.

The engineers from *Tested Data Systems Inc.* would like to consider the usage of Gradle for some of their projects. Consequently, they are weighing its capabilities, apply it into some pilot projects and try to differentiate it towards other possible alternatives, mainly Maven.

To make a first impression, the engineers at *Tested Data Systems Inc.* put face to face a simple Maven pom.xml configuration file (as you have previously seen into this book – listing 11.1) and a simple build.gradle file (this is the default name of the build descriptor in Gradle – listing 11.2).

Listing 11.1 A simple Maven pom.xml file

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.manning.junitbook</groupId>
  <artifactId>example-pom</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <dependencies>
    <dependency>
      <groupId>org.junit.jupiter</groupId>
      <artifactId>junit-jupiter-api</artifactId>
      <version>5.6.0</version>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>org.junit.jupiter</groupId>
      <artifactId>junit-jupiter-engine</artifactId>
      <version>5.6.0</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

```
</dependencies>
</project>
```

Listing 11.2 A simple Gradle build.gradle file

```
plugins {
    // Apply the java plugin to add support for Java
    id 'java'

    // Apply the application plugin to support building a CLI application
    id 'application'
}

repositories {
    // Use jcenter for resolving dependencies.
    // You can declare any Maven/Ivy/file repository here.
    jcenter()
}

dependencies {
    // Use JUnit Jupiter API for testing.
    testImplementation 'org.junit.jupiter:junit-jupiter-api:5.6.0'

    // Use JUnit Jupiter Engine for testing.
    testRuntimeOnly 'org.junit.jupiter:junit-jupiter-engine:5.6.0'
}

application {
    // Define the main class for the application
    mainClassName = 'com.manning.junitbook.ch11.App'
}

test {
    // Use junit platform for unit tests
    useJUnitPlatform()
}
```

You may be looking at this first build.gradle configuration file and wonder what is happening inside it. Its purpose is to provide you a first taste about the way it works. We'll get into details during this chapter. Additionally, you see many comments to explain what is happening there. The DSL approach, one of the Gradle strengths, becomes largely self-explanatory and relatively easy to maintain and manage.

For the Gradle installation procedure and a brief explanation of its main concepts, please have a look at Appendix B.

11.2 Setting up a Gradle project

It is time now for the engineers at *Tested Data Systems Inc.* to move on. We remind that they would like to consider the usage of Gradle for some of their projects and weighing its capabilities by creating JUnit 5 pilot projects with Gradle.

Oliver is involved in developing such a pilot project. He will create a new folder, called `junit5withgradle`, and here he will open a command prompt and execute the `gradle init` command.

Then, he will choose the following options:

- Select the type of project to generate: application
- Select implementation language: Java
- Select build script DSL: Groovy
- Select test framework: JUnit Jupiter

The result of running this command is shown in figure 11.1.

```
C:\Windows\system32\cmd.exe
C:\Work\Manning\junit5withgradle>gradle init

Select type of project to generate:
  1: basic
  2: application
  3: library
  4: Gradle plugin
Enter selection (default: basic) [1..4] 2

Select implementation language:
  1: C++
  2: Groovy
  3: Java
  4: Kotlin
Enter selection (default: Java) [1..4] 3

Select build script DSL:
  1: Groovy
  2: Kotlin
Enter selection (default: Groovy) [1..2] 1

Select test framework:
  1: JUnit 4
  2: TestNG
  3: Spock
  4: JUnit Jupiter
Enter selection (default: JUnit 4) [1..4] 4

Project name (default: junit5withgradle):

Source package (default: junit5withgradle): com.manning.junitbook.ch11

> Task :init
Get more help with your project: https://docs.gradle.org/5.5.1/userguide/tutorial_java_projects.html

BUILD SUCCESSFUL in 3s
2 actionable tasks: 2 executed
```

Fig 11.1 The result of executing the `gradle init` command

As Oliver has initialized a new Java project with Gradle and made a few options, Gradle has created its folder structure, which is following the Maven one. More exactly:

- `src/main/java` contains the Java source code
- `src/test/java` contains the Java tests

To start the build, Oliver will type `gradle build` on the command line prompt, into the folder that contains the `build.gradle` file and the whole project that has just been created.

We'll now take a look at the *build.gradle* file that has been created, explain its content and demonstrate how Oliver can extend it.

Listing 11.3 Definition of repositories into the build.gradle file

```
repositories {  
    // Use jcenter for resolving dependencies.  
    // You can declare any Maven/Ivy/file repository here.  
    jcenter()  
}
```

By default, Gradle uses JCenter as the repository for the applications that it manages. JCenter is the largest Java Repository in the world. This means that whatever is available on Maven Central is available on JCenter as well. Of course, we may specify the Maven Central repository to be used, our own repository, or multiple repositories to be used. Listing 11.4 shows that Oliver has chosen to use the Maven Central repository and the own repository from *Tested Data Systems Inc.*, which allows access to proprietary dependencies.

Listing 11.4 Definition of 2 repositories into the build.gradle file

```
repositories {  
    mavenCentral()  
  
    testit {  
        url "https://testedadatasystems.com/repository"  
    }  
}
```

Based on the options Oliver has provided, Gradle has added some dependencies to the *build.gradle* file configuration, as shown in listing 11.5.

Listing 11.5 Definition of dependencies into the build.gradle file

```
dependencies {  
    // This dependency is used by the application.  
    implementation 'com.google.guava:guava:27.1-jre'  
  
    // Use JUnit Jupiter API for testing.  
    testImplementation 'org.junit.jupiter:junit-jupiter-api:5.6.0'  
  
    // Use JUnit Jupiter Engine for testing.  
    testRuntimeOnly 'org.junit.jupiter:junit-jupiter-engine:5.6.0'  
}
```

The dependency configuration declares the external dependencies which we would like to be downloaded from the repository or repositories in use. It defines the different standard configurations from table 11.1.

Table 11.1 Standard dependency configurations and their meaning

Standard configuration	Meaning
implementation	The dependencies are required to compile the production source of the project.
runtime	The dependencies are required by the production classes at runtime. By default, it also includes the compile-time dependencies.
testImplementation	The dependencies required are to compile the test source of the project. By default, it includes compiled production classes and the compile-time dependencies.
testRuntime	The dependencies are required to run the tests. By default, it includes runtime and test compile dependencies.
runtimeOnly	The dependencies are only required at runtime, and not at compile time.
testRuntimeOnly	The dependencies are only required at test runtime, and not at test compile time.

Based on the option to create an application that Oliver has provided, Gradle has also added the configuration shown in listing 11.6.

Listing 11.6 Configuring the main class of the application into the build.gradle file

```
application {  
    // Define the main class for the application  
    mainClassName = 'com.manning.junitbook.ch11.App'  
}
```

It defines `com.manning.junitbook.ch11.App` as the main class of the application – the execution entry point.

The code from listing 11.7 enables the JUnit Platform support in `build.gradle`. More exactly, it specifies that the JUnit Platform should be used to execute the tests – remember that we were allowed to choose between a few platforms when Gradle has created the project for us.

Listing 11.7 Enabling the JUnit Platform support into the build.gradle file

```
test {  
    // Use junit platform for unit tests  
    useJUnitPlatform()  
}
```

`useJUnitPlatform` may get additional options. For example, we may specify the tags to be included or excluded when the tests are executed. The configuration may look like in listing 11.8.

Listing 11.8 Including and excluding tags to be executed inside the build.gradle file

```
test {  
    // Use junit platform for unit tests  
    useJUnitPlatform {
```

```
        includeTags 'individual'
        excludeTags 'repository'
    }
}
```

These options will mean that the tests tagged by JUnit 5 as 'individual' will be executed by Gradle, while the tests tagged by JUnit 5 as 'repository' will be excluded (listing 11.9).

Listing 11.9 Gradle will selectively execute differently tagged tests

```
@Tag("individual")
public class CustomerTest {
...
}

@Tag("repository")
public class CustomersRepositoryTest {
...
}
```

When a Gradle project is created, it will also create a *wrapper*. This wrapper represents a script that invokes a declared Gradle version, first downloading it, if it is not available. The script is contained in the gradlew.bat or gradlew file (depending on the operating system). If you build a project and distribute it to your customers, they will be able to quickly run that project without any manual installation and having the surety that they will use exactly that Gradle version that has been used when the project has been created.

We may create reports concerning the execution of the tests. When running the `gradle test` command (or `gradlew test`, if we use the wrapper), a report will be created into the `build/reports/tests/test` folder. Accessing the `index.html` file, we'll get the report shown in fig. 11.2. We have included in our project the tagged tests introduced in chapter 2 and executed them through Gradle, including the 'individual' tests and excluding the 'repository' ones, as described in the `build.gradle` file.

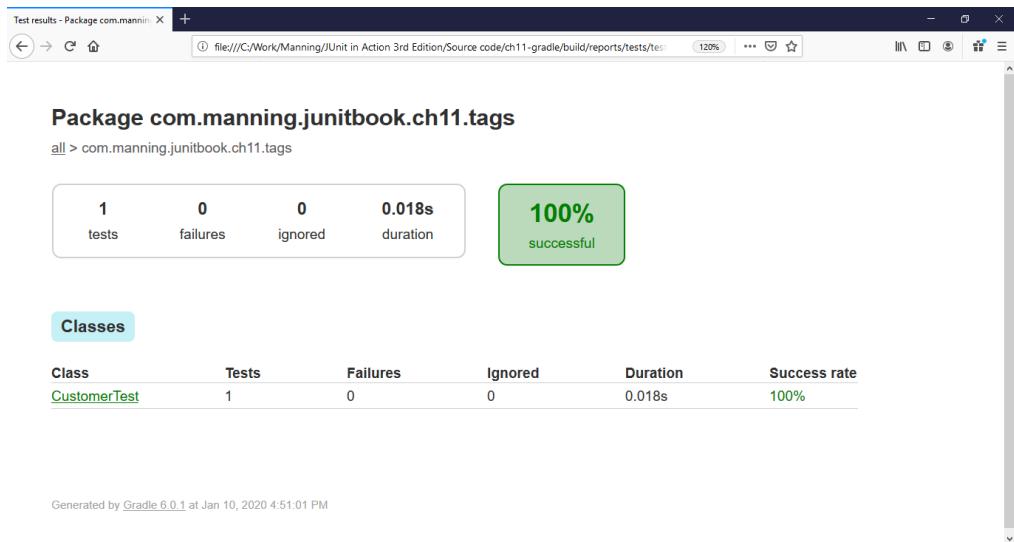


Fig 11.2 The report generated the Gradle includes the execution of CustomerTest, tagged as ‘individual’

At the moment of writing this chapter, there are some known limitations of using JUnit 5 with Gradle: classes and tests are still displayed by their names instead of `@DisplayName`. They are intended to be fixed into the future releases of Gradle.

11.3 Using Gradle plugins

So far so good – we have seen what Gradle is and how Oliver has used it to start a pilot project at *Tested Data Systems Inc.*, from scratch. We have also seen how Gradle is able to manage dependencies, by using a DSL (Domain Specific Language).

Oliver would still like to add plugins to his project. A Gradle plugin is, in fact, a set of tasks. A lot of common tasks such as compiling or setting up source files are handled by plugins. Applying a plugin to a project means that we allow the plugin to extend the project capabilities.

There are two types of plugins in Gradle: script plugins and binary plugins.

Script plugins represent tasks that are defined and can be applied from a script on the local filesystem or at a remote location.

What is of particular interest for us as Java developers using JUnit 5 are the binary plugins. Each binary plugin is identified by an id. The core plugins are using short names to be applied.

The code from listing 11.10 shows how Oliver applies two plugins to the `build.gradle` file.

Listing 11.10 Applying plugins into the build.gradle file

```
plugins {
    // Apply the java plugin to add support for Java
```

```
id 'java'

// Apply the application plugin to add support for building a CLI application
id 'application'
}
```

You see that this piece of code does two things: first, it applies the Java plugin to add support for the programming language – remember that during the initialization with the *gradle init* command, Oliver has chosen Java as language, and Gradle has added the needed support. Second, as he has chosen “application” as the type of project to generate, the plugin that supports building a CLI application has been added as well.

If we put head to head the pieces of code that we have discussed into listings 11.3-11.10, you will get Oliver’s Gradle configuration file from listing 11.2. We have promised at the time of showing it that we will detail everything. It is time to admit that we have really done this!

11.4 Creating a Gradle project from the scratch and testing it with JUnit 5

We’ll now demonstrate all the steps necessary for the creation of a JUnit 5 project that is managed by Gradle.

Tested Data Systems Inc. is an outsourcing company creating software projects for different companies. At *Tested Data Systems Inc.*, one of the projects under development for one of the customers is a flights management application.

In order to start such an application Oliver, project developer, will create the C:\Work\flightsmanagement folder that will become the working directory and where he will set up the Gradle project. He types the following on the command line:

```
gradle init
```

The result of running this command is shown in figure 11.3.

```
C:\WINDOWS\system32\cmd.exe
Select type of project to generate:
 1: basic
 2: application
 3: library
 4: Gradle plugin
Enter selection (default: basic) [1..4] 2

Select implementation language:
 1: C++
 2: Groovy
 3: Java
 4: Kotlin
Enter selection (default: Java) [1..4] 3

Select build script DSL:
 1: Groovy
 2: Kotlin
Enter selection (default: Groovy) [1..2] 1

Select test framework:
 1: JUnit 4
 2: TestNG
 3: Spock
 4: JUnit Jupiter
Enter selection (default: JUnit 4) [1..4] 4

Project name (default: flightsmanagement):
Source package (default: flightsmanagement): com.testeddatasystems.flightsmanagement
```

Fig 11.3 The result of executing the `gradle init` command to create the flights management application

The structure of the project is the one shown in fig. 11.4: the `src/main/java/` is the Gradle folder where the Java code for the project resides, while `src/main/test/` is the unit-tests for the tests.

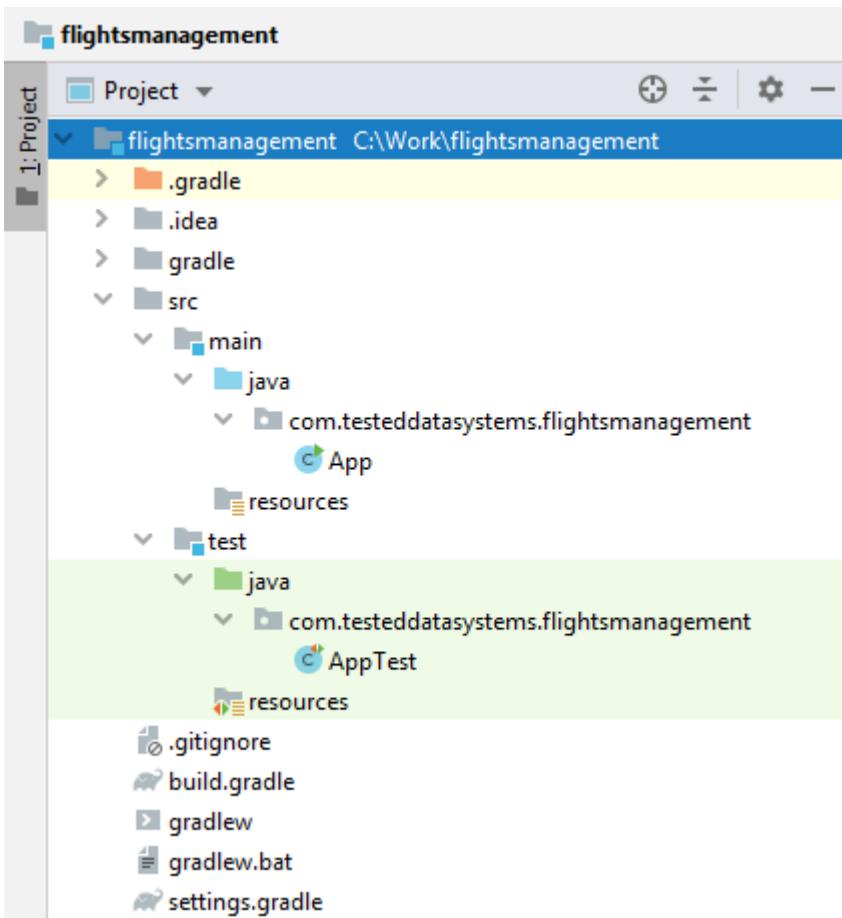


Fig 11.4 The newly created flights management project

The execution of the gradle init command also generated a build.gradle file (listing 11.11)

Listing 11.11 The build.gradle file for the flights management application

```
plugins {  
    // Apply the java plugin to add support for Java  
    id 'java'  
  
    // Apply the application plugin to add support for building a CLI application  
    id 'application'  
}  
  
repositories {  
    // Use jcenter for resolving dependencies.
```

```

    // You can declare any Maven/Ivy/file repository here.
    jcenter()
}

dependencies {
    // Use JUnit Jupiter API for testing.
    testImplementation 'org.junit.jupiter:junit-jupiter-api:5.6.0'

    // Use JUnit Jupiter Engine for testing.
    testRuntimeOnly 'org.junit.jupiter:junit-jupiter-engine:5.6.0'
}

application {
    // Define the main class for the application
    mainClassName = 'com.testeddatasystems.flightsmanagement.App'
}

test {
    // Use junit platform for unit tests
    useJUnitPlatform()
}

```

Oliver will remove the existing auto-generated `App` and `AppTest` classes. He will instead introduce the two classes that will start his application: `Passenger` (listing 11.12) and `PassengerTest` (listing 11.13).

Listing 11.12 The Passenger class

```

public class Passenger {

    private String identifier;                      #A
    private String name;                           #A

    public Passenger(String identifier, String name) {      #B
        this.identifier = identifier;                 #B
        this.name = name;                            #B
    }                                              #B

    public String getIdentifier() {                  #C
        return identifier;                         #C
    }                                              #C

    public String getName() {                      #C
        return name;                             #C
    }                                              #C

    @Override
    public String toString() {                     #D
        return "Passenger " + getName() +          #D
               " with identifier: " + getIdentifier(); #D
    }                                              #D

    public static void main (String args[]) {       #E
        Passenger passenger = new Passenger("123-456-789", "John Smith"); #E
        System.out.println(passenger);             #E
    }                                              #E
}

```

The Passenger class contains the following:

- Two fields, identifier and name (#A).
- A constructor receiving as arguments the identifier and the name (#B).
- Getters for the identifier and the name fields (#C).
- The overridden `toString` method (#D).
- The `main` method will create a passenger and display him (#E).

Listing 11.13 The PassengerTest class

```
public class PassengerTest {  
  
    @Test  
    void testPassenger() {  
        Passenger passenger = new Passenger("123-456-789", "John Smith");  
        assertEquals("Passenger John Smith with identifier: 123-456-789",  
                    passenger.toString());  
    }  
}
```

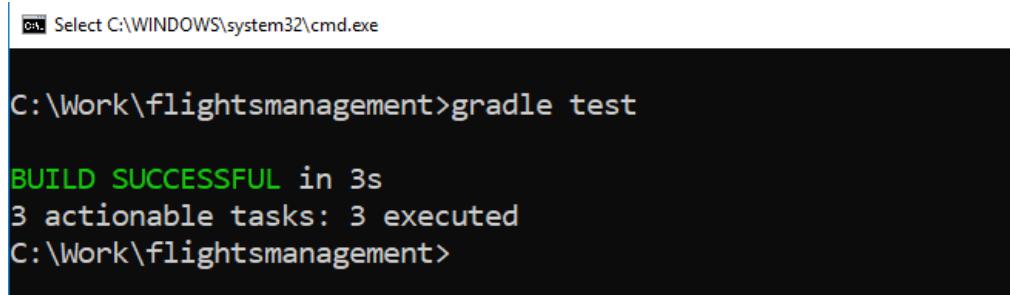
The PassengerTest class contains one test:

1. `testPassenger`, that verifies the output of the overridden `toString` method (#A).

Oliver will change, in the `build.gradle` file, the main class to be:

```
application {  
    // Define the main class for the application  
    mainClassName = 'com.testeddatasystems.flightsmanagement.Passenger'  
}
```

Oliver will then execute the `gradle test` command at the level of the project folder. This command will simply successfully run the JUnit 5 test that has been introduced into the application (fig. 11.5).

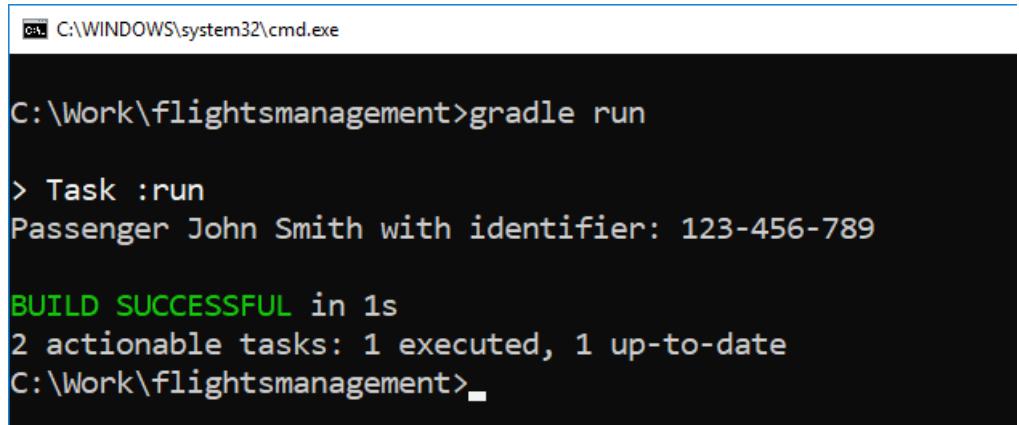


The screenshot shows a terminal window with the following text:

```
cmd Select C:\WINDOWS\system32\cmd.exe  
  
C:\Work\flightsmanagement>gradle test  
  
BUILD SUCCESSFUL in 3s  
3 actionable tasks: 3 executed  
C:\Work\flightsmanagement>
```

Fig 11.5 Executing the `gradle test` command for the Gradle project under development

Oliver will then execute the `gradle run` command at the level of the project folder. This command will simply run the main class of the application, as described in the `build.gradle` file and display the information about the passenger John Smith (fig. 11.6).



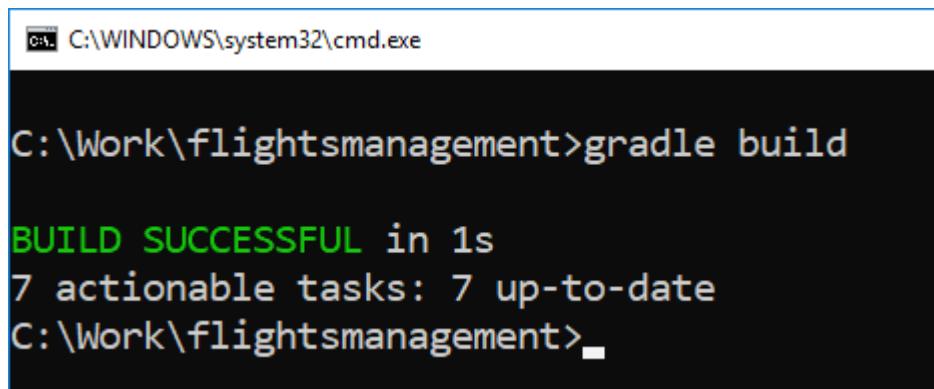
```
C:\WINDOWS\system32\cmd.exe
C:\Work\flightsmanagement>gradle run

> Task :run
Passenger John Smith with identifier: 123-456-789

BUILD SUCCESSFUL in 1s
2 actionable tasks: 1 executed, 1 up-to-date
C:\Work\flightsmanagement>
```

Fig 11.6 Executing the `gradle run` command for the Gradle project under development

Running the `gradle build` command at the level of the project folder (fig. 11.7) will also create the JAR artifact into the `build/libs` folder (fig. 11.8).



```
C:\WINDOWS\system32\cmd.exe
C:\Work\flightsmanagement>gradle build

BUILD SUCCESSFUL in 1s
7 actionable tasks: 7 up-to-date
C:\Work\flightsmanagement>
```

Fig 11.7 Executing the `gradle build` command for the Gradle project under development

Local Disk (C:) > Work > flightsmanagement > build > libs			
Name	Date modified	Type	Size
 flightsmanagement.jar	11/15/2019 2:28 PM	Executable Jar File	2 KB

Fig 11.8 The JAR packaged file into the Gradle `build/libs` folder of the `flightsmanagement` project

At this time, Oliver has created a fully functional Gradle project to develop the flights management application and test it using JUnit 5. From now on, he can continue to add new classes and tests and execute the Gradle commands to package the application and to run the tests.

11.5 Comparing Gradle and Maven

At the end of this chapter, we may say that we have used and compared two build tools that will help us manage our JUnit 5 projects. The engineers at *Tested Data Systems Inc.* do have some freedom of choice: some projects may use Maven, as it is an older and reliable tool, that most of them know well. Or, in some projects, they may decide to adopt Gradle. The reasons to do this may be that: they need to write more custom tasks, that are harder to do with Maven; they consider XML as tedious, and they prefer to use a DSL (Domain Specific Language) that is close to Java; or, they are challenged to adopt something new.

As you see, it may be a matter of personal or project preference – and it may be the same for you. In general, once you master one of these two build tools, joining a project that uses the other one is not a difficult task.

We have mentioned that, at least of the moment when this chapter is written, there are some known issues of using JUnit 5 with Gradle: classes and tests are still displayed by their names instead of `@DisplayName`. They are expected to be solved into the future releases of Gradle.

We have to mention that the projects at the global level are still predominantly using Maven – about three-quarters of them, according to our researches. Besides this, the numbers are quite stable, meaning that, during the years, Gradle hasn't had much advancement on the market. This is the reason why, across this book, we are mostly using Maven.

11.6 Summary

This chapter has covered the following:

- The Gradle build tool has been introduced and differentiated from other build tools.
- A Java project using JUnit 5 has been created with the help of Gradle.
- Gradle plugins have been introduced and their operation has been demonstrated.
- A Gradle project has been created from scratch, opened it into the IntelliJ IDEA and tested using JUnit 5.
- Gradle and Maven have been compared as two build tools alternatives.

12

JUnit 5 IDE support

This chapter covers

- Introducing IDEs
- Examining the usage of JUnit 5 from IntelliJ IDEA
- Executing JUnit 5 tests from IntelliJ IDEA
- Examining the usage of JUnit 5 from Eclipse
- Executing JUnit 5 tests from Eclipse
- Examining the usage of JUnit 5 from NetBeans
- Executing JUnit 5 tests from NetBeans
- Comparing JUnit 5 usage in IntelliJ IDEA, Eclipse, and NetBeans

Enjoy the little things, for one day you may look back and realize they were the big things.

-Robert Brault

In this chapter, we will analyze and compare the main IDEs that may be used for developing Java applications tested with the help of JUnit 5.

12.1 Introducing IDEs

An *IDE* (Integrated Development Environment) is a software application that provides software development facilities for computer programmers. Java IDEs represent a great tool to more easily write and debug Java programs.

Taking into account the popularity of the Java programming language, there are many IDEs that have been created for the help of developers. From these ones, we have selected three for our analysis and comparison, the most popular ones in use: IntelliJ IDEA, Eclipse,

and NetBeans. At the time of developing this chapter, different market shares analysis were evaluating IntelliJ IDEA and Eclipse at about 40% or more, each of them, while NetBeans is evaluated at about 10% or more. We have also included this one in our analysis as it is still the third option on the market and it is popular in some regions of the world.

Which of the possible IDEs you will choose may be a matter of personal preference or a matter of the tradition inside the project. It is possible that all of them are in use, at the level of the same company or even of the same project. In fact, this is what happens inside our previously introduced *Tested Data Systems Inc.* company. We'll point out the various reasons that determined which of the options the developers chose.

Our analysis and demonstration will focus on the capabilities of these IDEs using JUnit 5 – the just-in-time information that you need now. For more comprehensive guidance, there are many other available sources, starting with the official documentation of each of the IDEs. We'll demonstrate the usage of JUnit 5 together with each of the three options, in this order – IntelliJ IDEA, Eclipse, and NetBeans, as this is the order they have made the integration with JUnit 5.

12.2 Using JUnit 5 with IntelliJ IDEA

IntelliJ IDEA is an IDE developed by JetBrains. It is available as an Apache 2 Licensed community edition and in a proprietary commercial edition. For brief instructions about the IntelliJ IDEA installation, please look at appendix C.

In order to demonstrate the work of the JUnit 5 tests through different IDEs, we have made a selection of the tests from chapter 2. Those tests were very comprehensive, as they were covering the capabilities of JUnit 5. From them, we have selected the ones that are significant in the context of the usage from inside an IDE. You may review the new JUnit 5 features and their use cases in chapter 2. Here, we are most interested in the interaction with JUnit 5 from inside each IDE, but we'll briefly remind in each case why a developer would use that particular feature.

You may launch IntelliJ IDEA, then open a project by accessing the File -> Open menu. We open the project from the source code of this chapter, the IDE will look like in figure 12.1. The project contains the tests that demonstrate the capabilities of JUnit 5: displaying names, nested, parameterized, dynamic, repeated and dynamic tests, using tags.

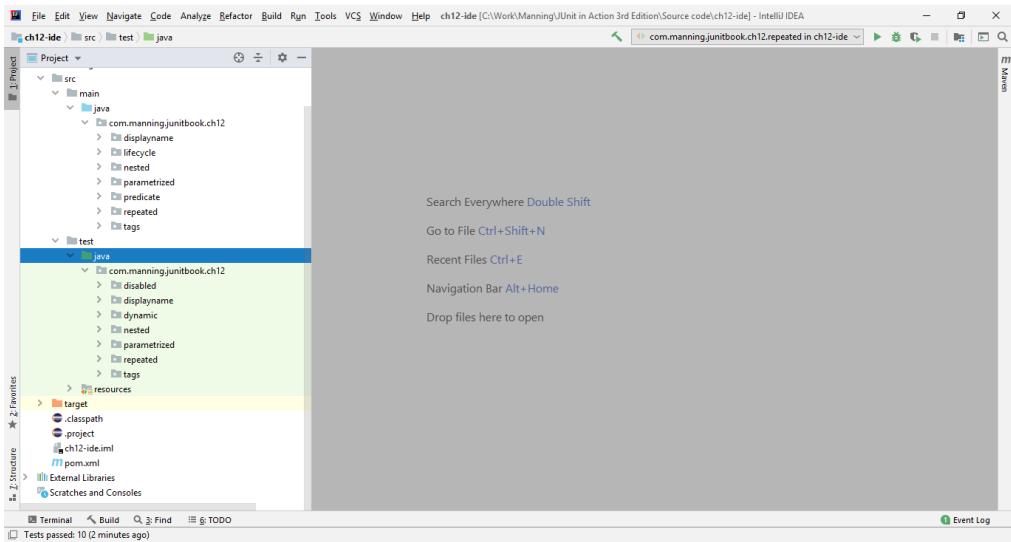


Fig 12.1 IntelliJ IDEA with an open JUnit 5 project

From here, you may execute all tests by right-clicking on the highlighted test/java folder, then choosing *Run 'All Tests'*. You will get a result like the one shown in figure 12.2. You will most likely want to periodically quickly run all the tests while you are working on the implementation of a particular feature – you should make sure that the full project is working fine, and that your implementation does not affect the previously existing functionality.

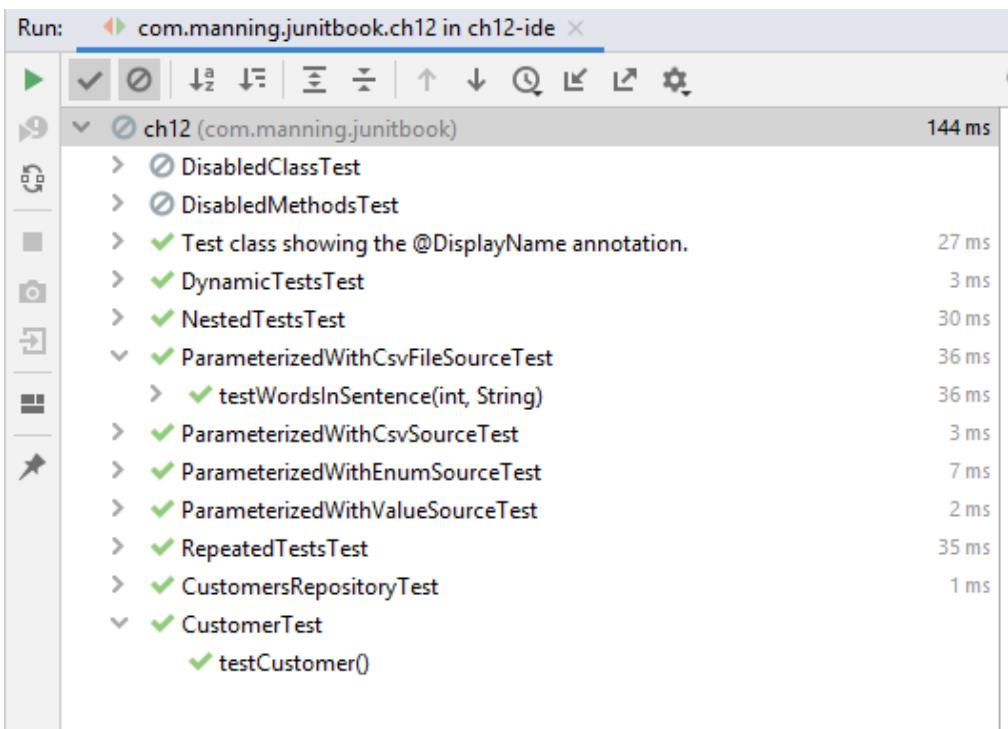


Fig 12.2 Executing all tests from inside IntelliJ IDEA

Let's have a look at how to run particular test types with the help of IntelliJ IDEA. In figure 12.2 you see that the disabled tests are shown in gray, and they are marked as "ignored" after executing the whole suite. You may, however, force the execution of a particular test, even if it is disabled if you right-click on it and run it directly, as shown in figure 12.3. You may want to do that if you would like to check, from time to time, if the conditions that prevented the correct execution of that test have ceased to exist. For example, your test may wait for the implementation of a feature by a different team or for the availability of a resource, and you will verify their fulfillment.

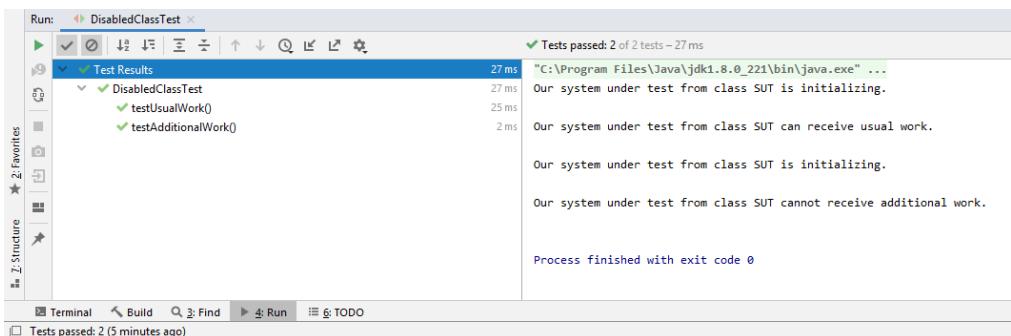


Fig 12.3 Forcing the execution of a disabled test from IntelliJ

When you run the tests that are annotated with `@DisplayName`, they will be nicely shown as in figure 12.4. You will use this annotation and will work with this kind of test from IntelliJ IDEA when you would like to watch some significant information while you are developing from the IDE. Details about the functionality of this annotation are to be found in chapter 2.

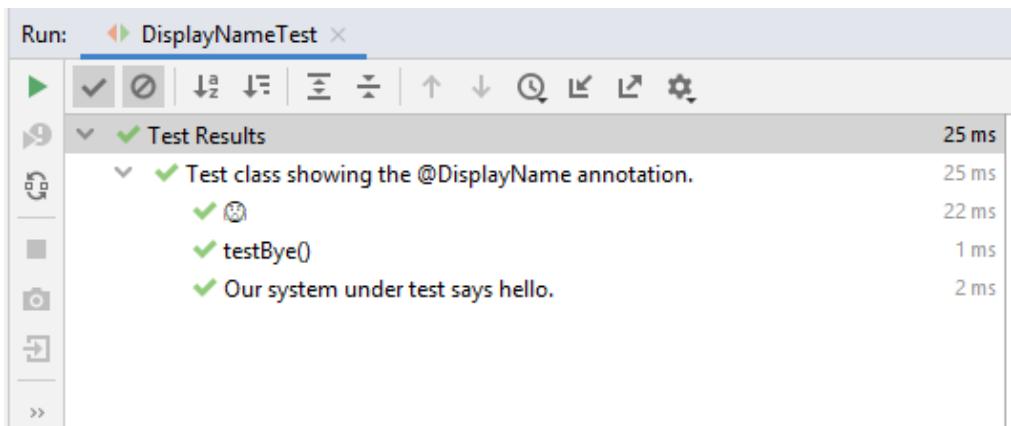


Fig 12.4 Running a test annotated with `@DisplayName` from IntelliJ IDEA

When you run the dynamic tests that are annotated with `@TestFactory`, they will be nicely shown as in figure 12.5. You will use this annotation and will work with this kind of test from IntelliJ IDEA when you would like to generate your tests at runtime writing some reasonable amount of code and to display them in a nice way. Details about the functionality of this annotation are to be found in chapter 2.

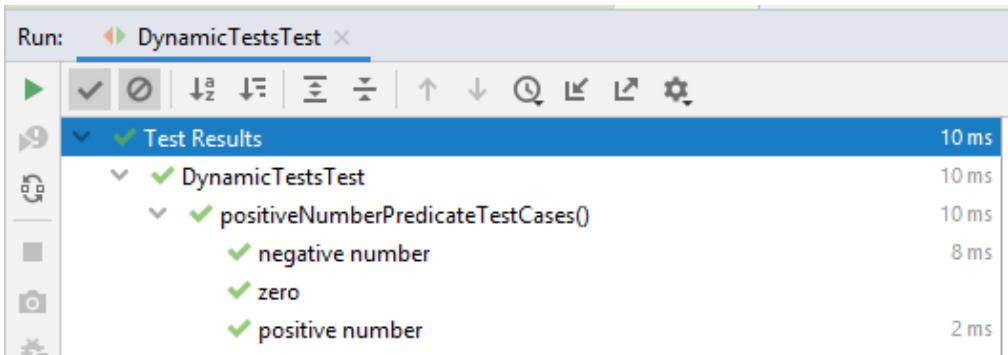


Fig 12.5 Running a dynamic test from IntelliJ IDEA

When you run the nested tests, they will be nicely shown in a hierarchical way, as in figure 12.6, where we have also taken benefit of the `@DisplayName` annotation capabilities. You will use nested tests to express the relationship among several groups of tests that are tightly coupled and IntelliJ IDEA will nicely picture the hierarchy. Details about the functionality of the `@Nested` annotation are to be found in chapter 2.

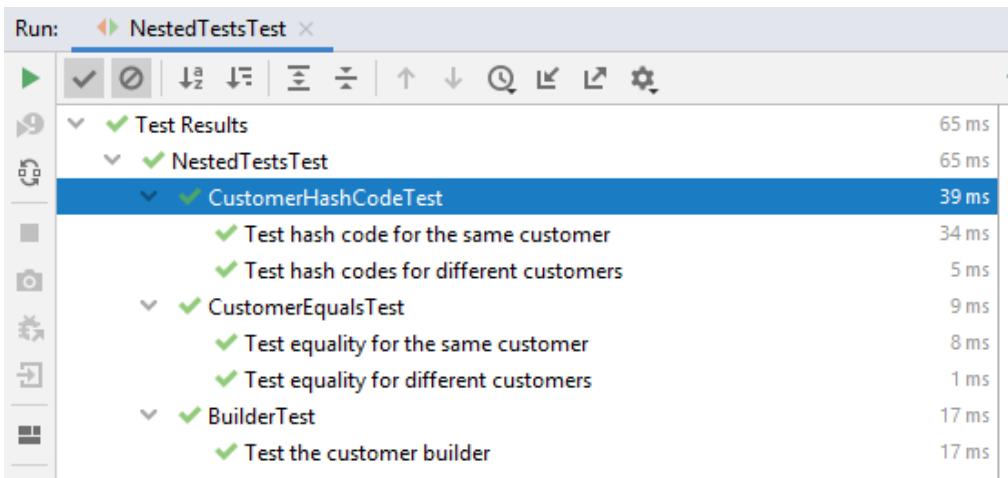


Fig 12.6 Running nested tests from IntelliJ IDEA

When you run parameterized tests, they will be displayed in detail, showing all the parameters that each of them is receiving, as in figure 12.7. You will use parameterized tests to write one single test, and then do the testing on a series of different arguments. Details about the functionality of the annotation related to parameterized tests are to be found in chapter 2.

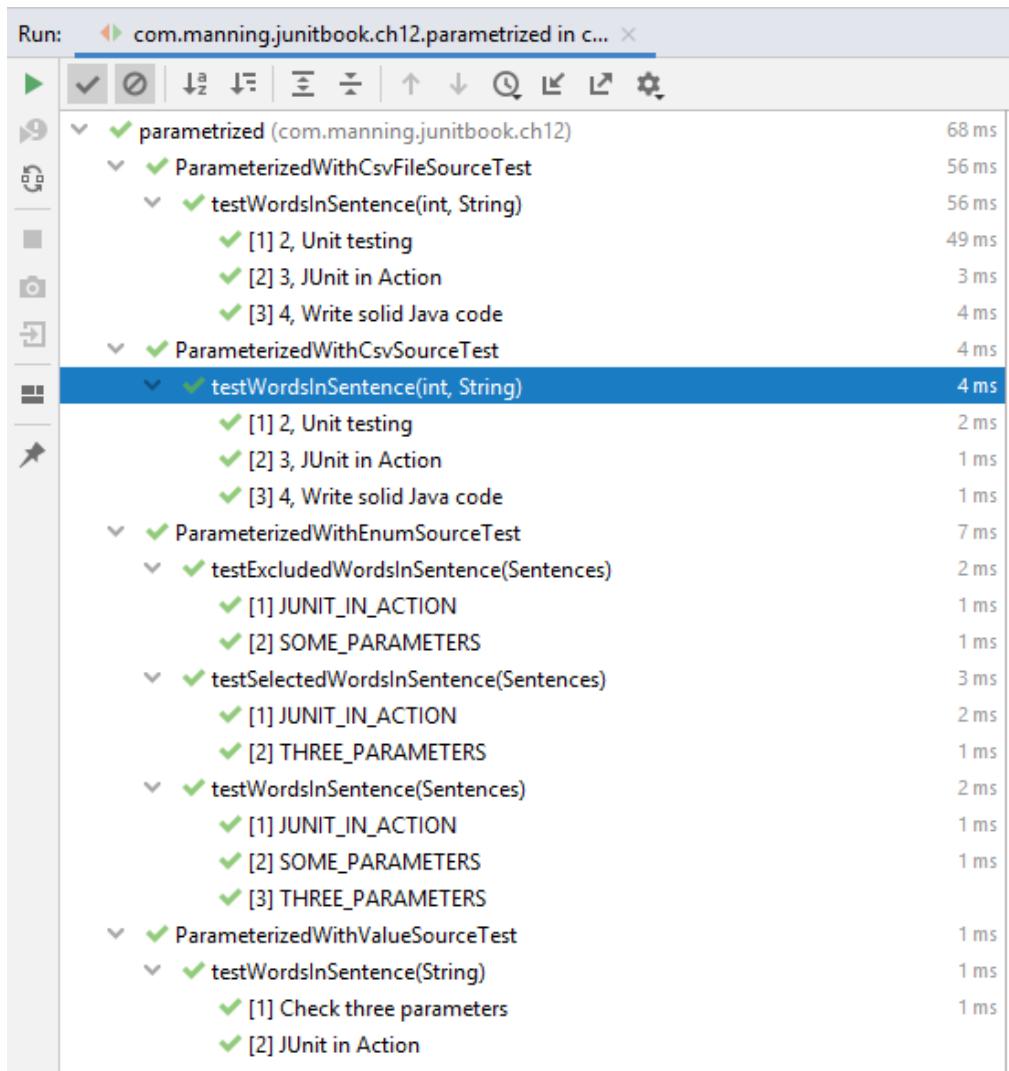


Fig 12.7 Running parameterized tests from IntelliJ IDEA

When you run repeated tests, they will be displayed in detail, showing all the required information about the currently repeated test, as in figure 12.8. You will use repeated tests if the running conditions may change from one execution to another one of the same test. Details about the functionality of the `@RepeatedTest` annotation are to be found in chapter 2.

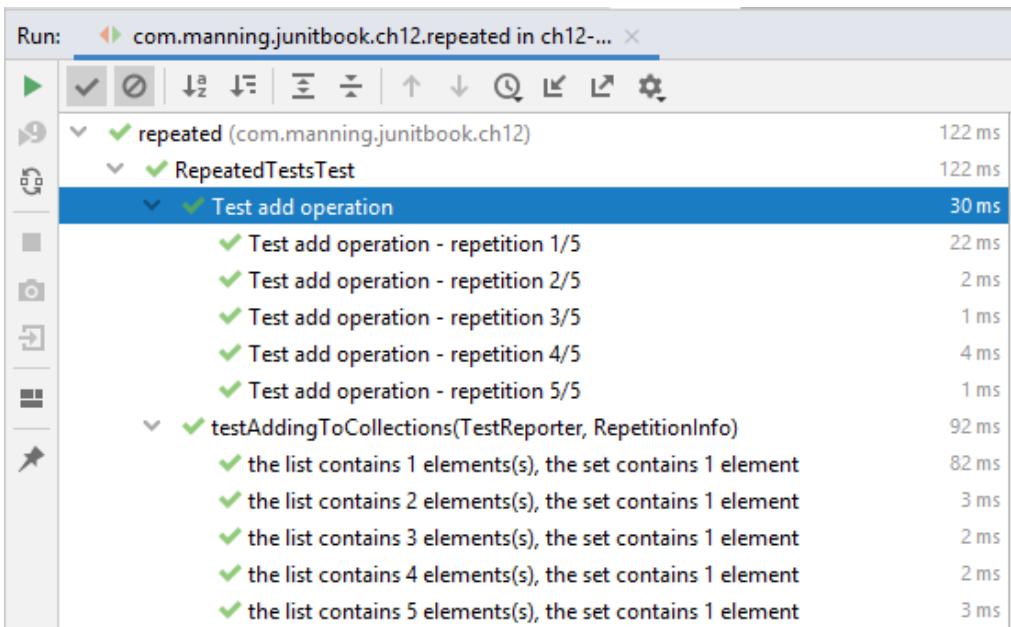


Fig 12.8 Running repeated tests from IntelliJ IDEA

When running the whole tests suite from IntelliJ IDEA, the tagged tests are also run. If you would like to run only some particular tagged tests, the IDE will provide this option. You have to go to *Run -> Edit Configurations...* and to choose from *Test kind -> Tags*, as shown in figure 12.9. Then, you will be able to run only that particular configuration addressing tagged tests. You would like to do this when you have a larger suite of tests and you need to focus on particular ones and not to spend time with the whole suite. Details about the functionality of the @Tag annotation are to be found in chapter 2.

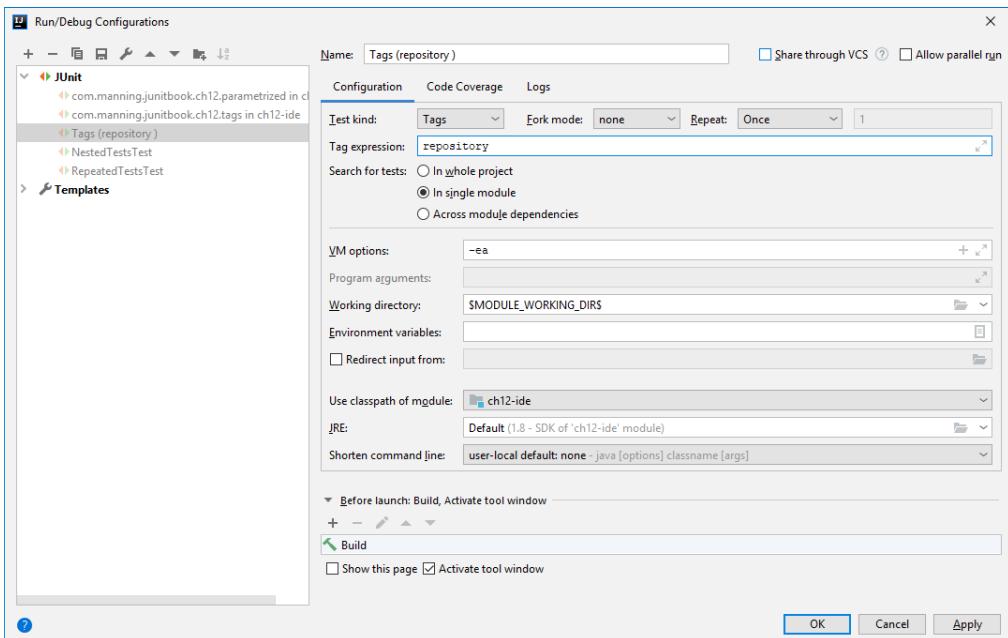


Fig 12.9 Configuring the option of running only tagged tests from IntelliJ IDEA

IntelliJ IDEA also provides an easy possibility of running tests with code coverage. If you right-click on the test or suite to be run, then choose *Run with Coverage*, you will get a table like the one from figure 12.10. You may investigate at the level of the classes, and get information like the one from figure 12.11. Or you may press on the *Generate Coverage Report* button and obtain an HTML report, as in figure 12.12.

We have discussed in more detail about code coverage in chapter 6. We remind that it is a metric of the quality of tests, showing how much of the production code is covered by the tests we are executing. While developing your code and your tests, you would like to get quick feedback not only about the fact if they run successfully or not but also about their coverage. IntelliJ IDEA strongly supports you in both goals.

At Tested Data Systems Inc., the developers that have decided to use IntelliJ for their projects are not only the ones that were already using it before moving to JUnit 5, but also the ones that have introduced this version of the testing framework during its early days. At that time, IntelliJ IDEA was the only IDE providing support for JUnit 5, and it was the only option.

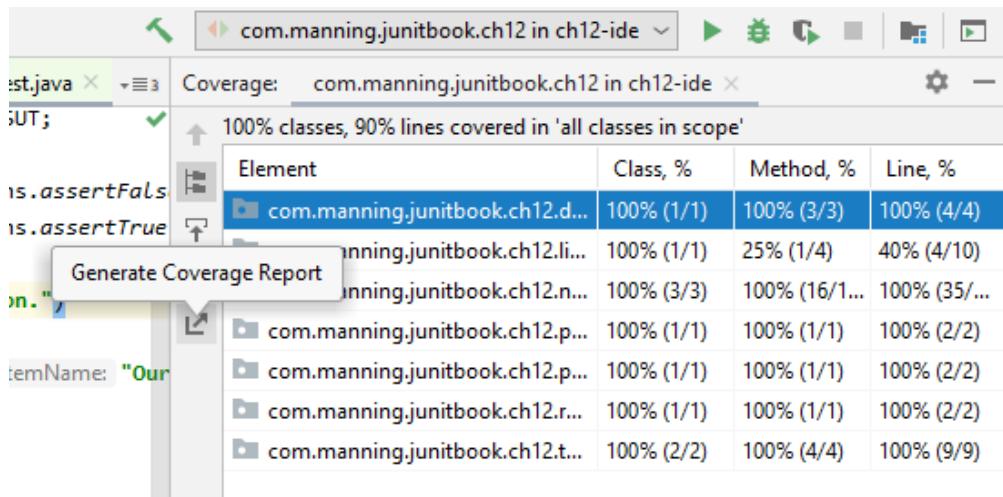


Fig 12.10 The table produced by IntelliJ IDEA after running *Test with Coverage*

The screenshot shows the IntelliJ IDEA interface with the code for the `SUT.java` class. The code defines a class `SUT` with methods `canReceiveUsualWork()` and `canReceiveAdditionalWork()`. The coverage report for this class is shown in the bottom right corner of the code editor, indicating 100% coverage for both methods. The code editor also shows other parts of the project structure and some terminal output at the bottom.

```

public class SUT {
    private String systemName;

    public SUT(String systemName) {
        this.systemName = systemName;
        System.out.println(systemName + " from class " + getClass().getSimpleName() + " is initialized");
    }

    public boolean canReceiveUsualWork() {
        System.out.println(systemName + " from class " + getClass().getSimpleName() + " can receive usual work");
        return true;
    }

    public boolean canReceiveAdditionalWork() {
    }
}

```

Fig 12.11 The code coverage at the level of individual classes

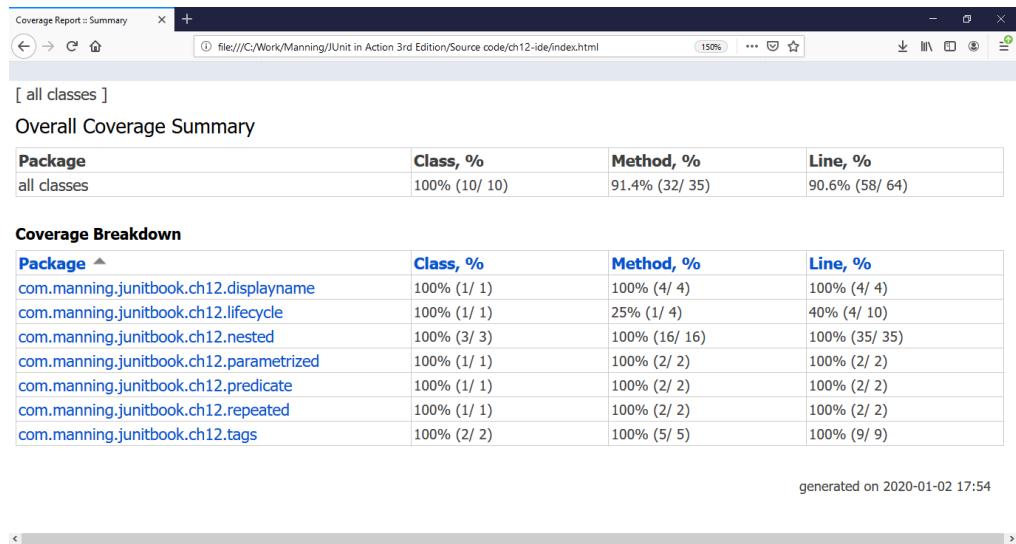


Fig 12.12 The code coverage HTML report obtained with the help of IntelliJ IDEA

12.3 Using JUnit 5 with Eclipse

Eclipse is an IDE written mostly in Java. It may be used to develop applications in many programming languages, but its primary use is for developing Java applications. For brief instructions about the Eclipse installation, please look at appendix C.

We'll pass through a similar demonstration as for IntelliJ IDEA: we have made a selection of the tests from chapter 2 that are significant in the context of the usage from inside an IDE.

You may launch Eclipse, then import a project by selecting *File -> Import -> General -> Existing Projects into Workspace*, then choose the folder where the project is located. The IDE will look like in figure 12.13.

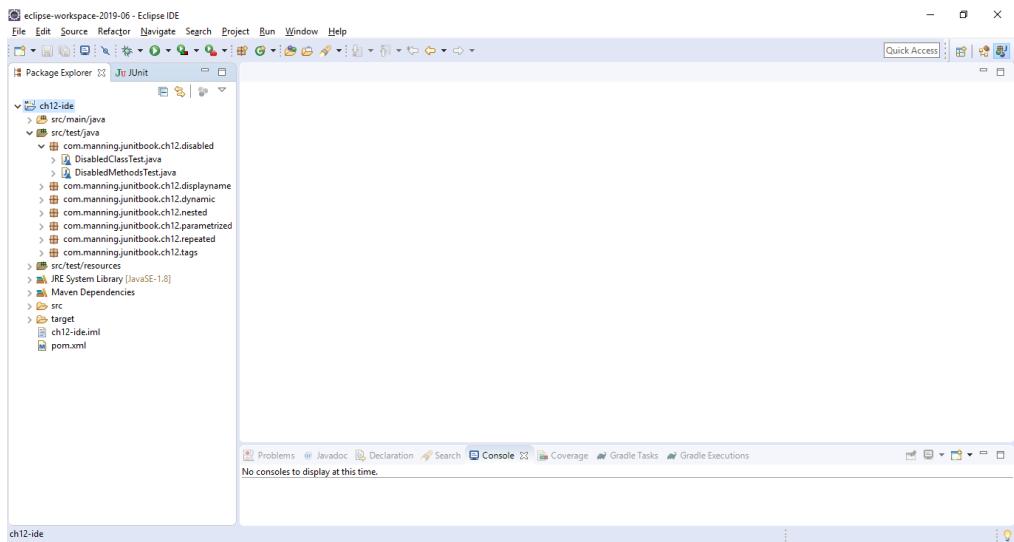


Fig 12.13 Eclipse with an open JUnit 5 project

From here, you may execute all tests by right-clicking on the project, then choosing *Run As -> JUnit test*. You will get a result like the one shown in figure 12.14. You would most likely want to periodically run all the tests while you are working on the implementation of a particular feature – you would like to make sure that the full project is working fine, and that your implementation does not affect the previously existing functionality.

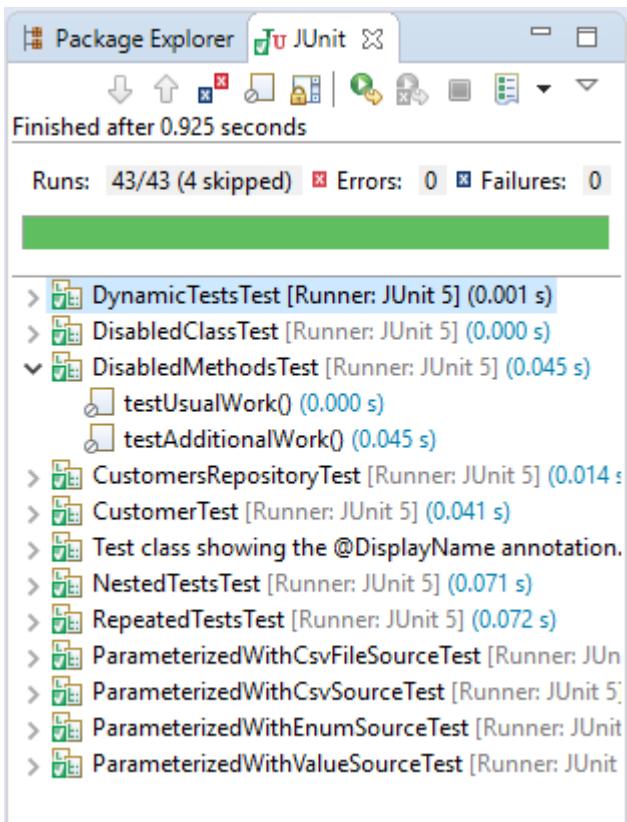


Fig 12.14 Executing all tests from inside Eclipse

Let's have a look at how to run particular test types with the help of Eclipse. In figure 12.14 you see that the disabled tests are shown in gray, and they are marked as "skipped" after executing the whole suite. In contrast to the capabilities of IntelliJ IDEA, you cannot force the execution of a particular disabled test.

When you run the tests that are annotated with `@DisplayName`, they will be nicely shown as in figure 12.15. You will use this annotation and will work with this kind of test from Eclipse when you would like to watch some significant information while you are developing from the IDE. Details about the functionality of this annotation are to be found in chapter 2.

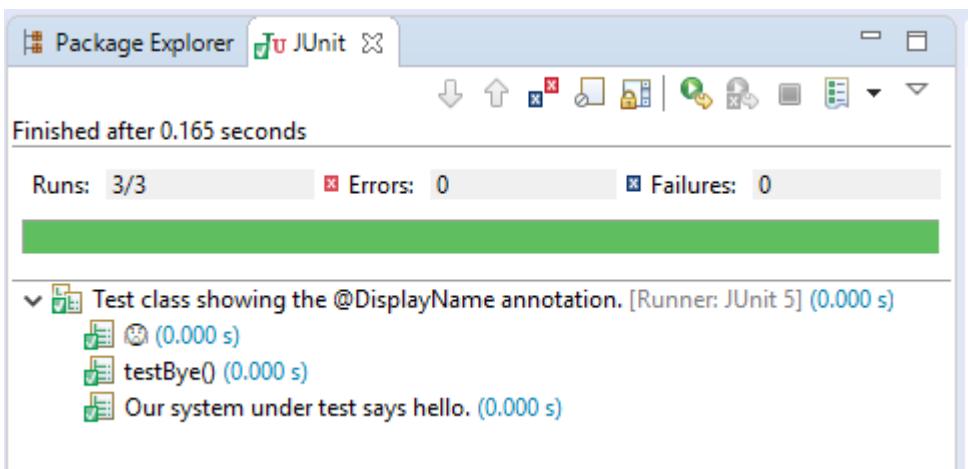


Fig 12.15 Running a test annotated with `@DisplayName` from Eclipse

When you run the dynamic tests that are annotated with `@TestFactory`, they will be nicely shown as in figure 12.16. You will use this annotation and will work with this kind of test from Eclipse when you would like to generate your tests at runtime writing some reasonable amount of code and to display them in a nice way. Details about the functionality of this annotation are to be found in chapter 2.

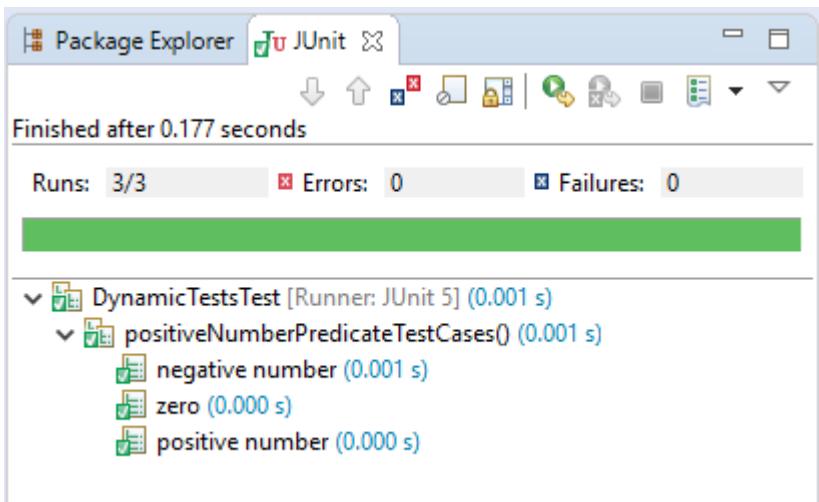


Fig 12.16 Running a dynamic test from Eclipse

When you run the nested tests, they will be nicely shown in a hierarchical way, as in figure 12.17, where we have also taken the benefit of the `@DisplayName` annotation capabilities. You will use nested tests to express the relationship among several groups of tests that are tightly coupled and Eclipse will nicely portray the hierarchy. Details about the functionality of the `@Nested` are to be found in chapter 2.

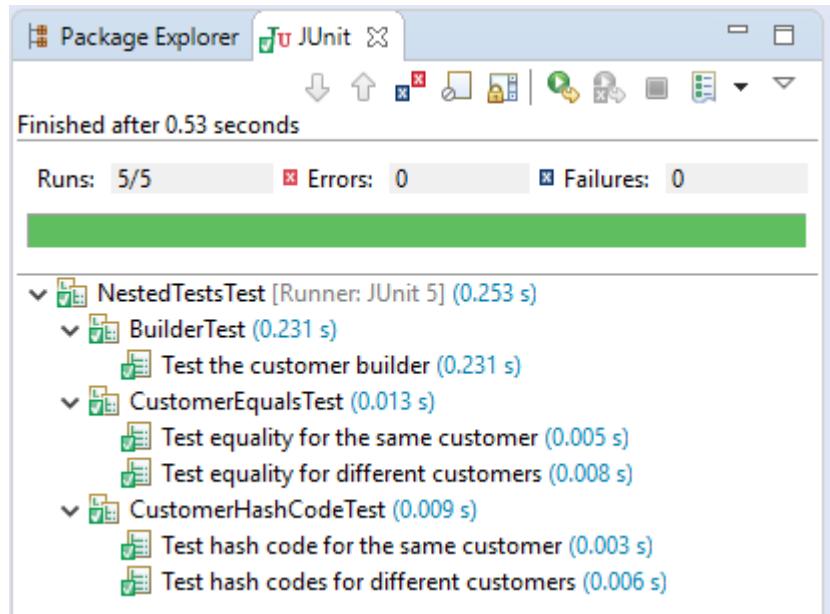


Fig 12.17 Running nested tests from Eclipse

When you run parameterized tests, they will be displayed in detail, showing all the parameters that each of them is receiving, as in figure 12.18. You will use parameterized tests to write one single test, and then do the testing on a series of different arguments. Details about the functionality of the annotation for parameterized tests are to be found in chapter 2.

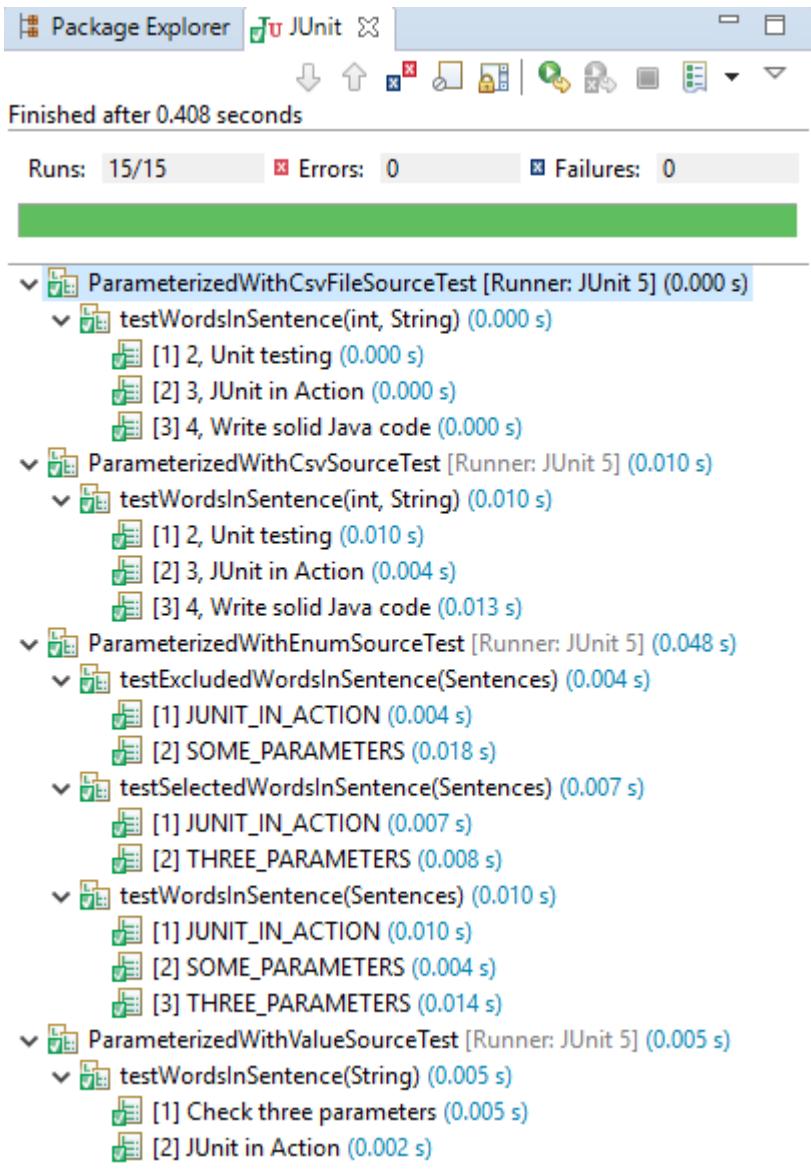


Fig 12.18 Running parameterized tests from Eclipse

When you run repeated tests, they will be displayed in detail, showing all the required information about the currently repeated test, as in figure 12.19. You will use repeated tests if the running conditions may change from one execution to another one of the same test.

Details about the functionality of the `@RepeatedTest` annotation are to be found in chapter 2.

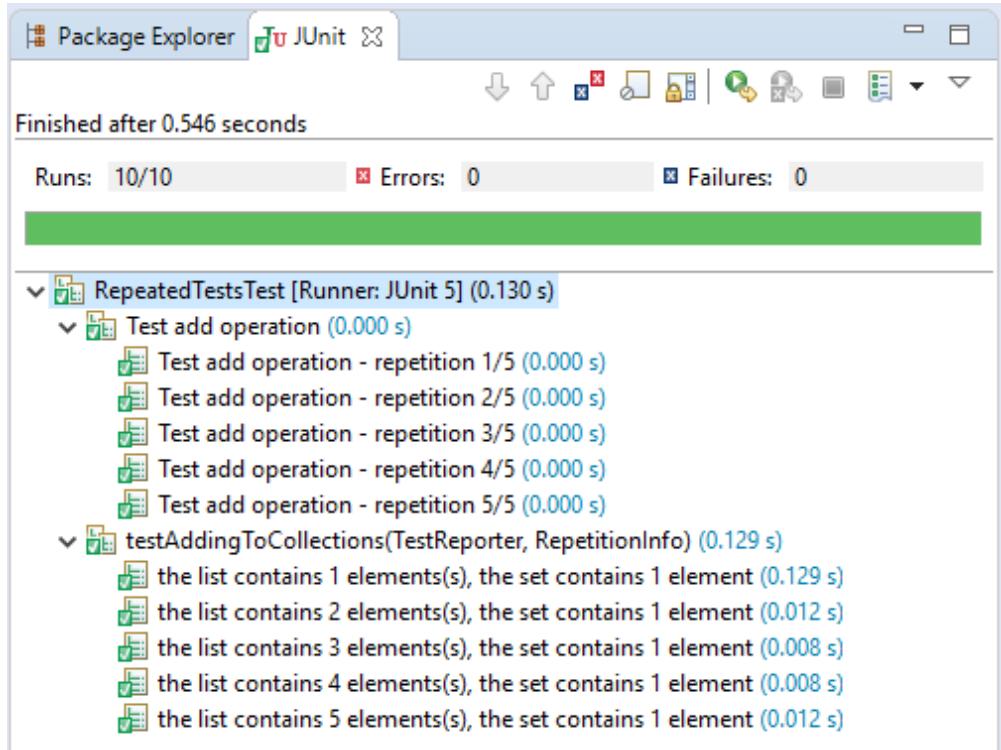


Fig 12.19 Running repeated tests from Eclipse

When running the whole tests suite from Eclipse, the tagged tests are also run. If you would like to run only some particular tagged tests, the IDE will provide this option. You have to go to *Run -> Run Configurations... -> JUnit* and to choose from *Include and Exclude Tags*, as shown in figure 12.20. Then, you will be able to run only that particular configuration addressing tagged tests. You would like to do this when you have a larger suite of tests and you need to focus on particular ones and not to spend time with the whole suite. Details about the functionality of the `@Tag` annotation are to be found in chapter 2.

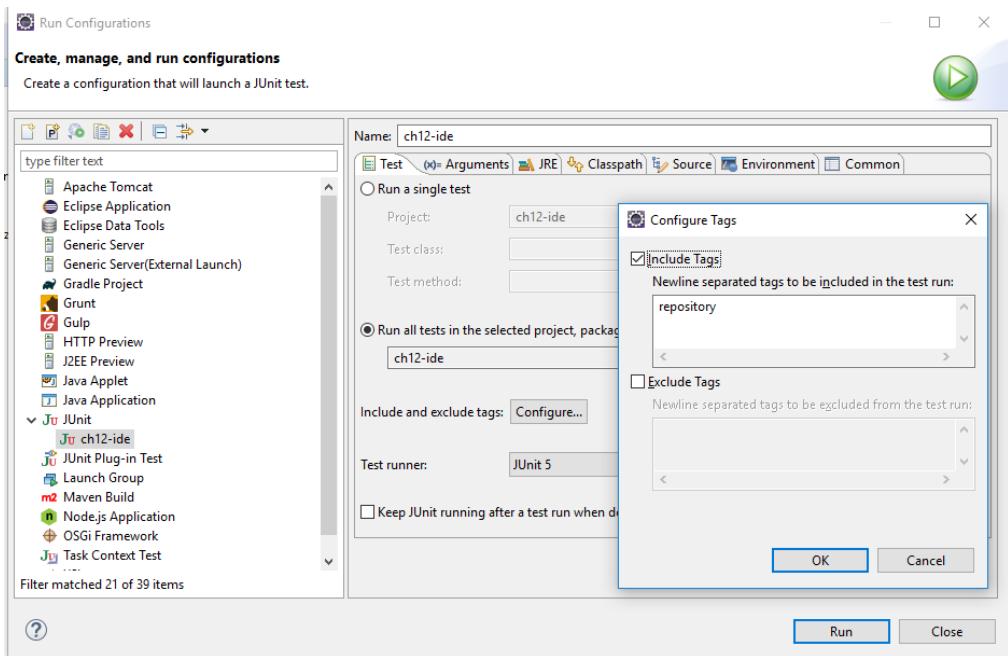


Fig 12.20 Configuring the option of running only tagged tests from Eclipse

Eclipse also provides an easy possibility of running tests with code coverage. If you right-click on the project, test or suite to be run, then choose *Coverage As -> JUnit Test*, you will get a table like the one from figure 12.21. You may investigate at the level of the individual classes, and get information like the one from figure 12.22. Or you may right-click on the *Coverage* table, choose *Export Session* and obtain an HTML report, as in figure 12.23.

While developing your code and your tests, you would like to get quick feedback not only about the fact if they run successfully or not but also about their coverage. Eclipse strongly supports you in both goals. The code coverage is a metric of the quality of tests, showing how much of the production code is covered by the tests we are executing.

At Tested Data Systems Inc., the developers that have decided to use Eclipse for their projects are generally the ones that were already using it before moving to JUnit 5 and were not under pressure to adopt it. This is simply because Eclipse has come with JUnit 5 support at a later time than IntelliJ IDEA.

Element	Coverage	Covered Instructions	Missed Instructions	Total Instructions
ch12-idb	88.9 %	776	97	873
src/main/java	81.0 %	255	60	315
com.manning.junitbook.ch12.lifecycle	28.2 %	22	56	78
SUT.java	28.2 %	22	56	78
com.manning.junitbook.ch12.nested	97.6 %	165	4	169
Customer.java	97.2 %	141	4	145
Gender.java	100.0 %	24	0	24
com.manning.junitbook.ch12.displayname	100.0 %	9	0	9
com.manning.junitbook.ch12.parameterized	100.0 %	8	0	8
com.manning.junitbook.ch12.predicate	100.0 %	9	0	9
com.manning.junitbook.ch12.repeated	100.0 %	7	0	7
com.manning.junitbook.ch12.tags	100.0 %	35	0	35
src/test/java	93.4 %	521	37	558

Fig 12.21 The table produced by Eclipse after running *Coverage As -> JUnit Test*

```

4* * Licensed to the Apache Software Foundation (ASF) under one or more
21
22 package com.manning.junitbook.ch12.lifecycle;
23
24 public class SUT {
25     private String systemName;
26
27     public SUT(String systemName) {
28         this.systemName = systemName;
29         System.out.println(systemName + " from class " + getClass().getSimpleName() + " is initializing.");
30     }
31
32     public boolean canReceiveUsualWork() {
33         System.out.println(systemName + " from class " + getClass().getSimpleName() + " can receive usual work.");
34         return true;
35     }
36
37     public boolean canReceiveAdditionalWork() {
38         System.out.println(systemName + " from class " + getClass().getSimpleName() + " cannot receive additional work.");
39         return false;
40     }
41
42     public void close() {
43         System.out.println(systemName + " from class " + getClass().getSimpleName() + " is closing.");
44     }
45
46
47 }

```

Fig 12.22 The code coverage at the level of individual classes

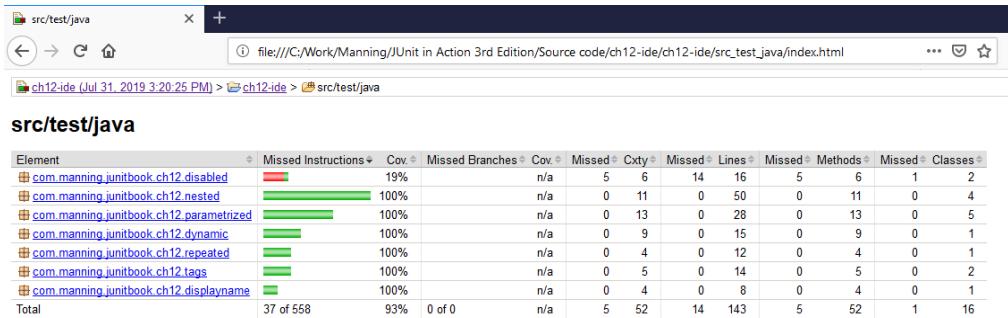


Fig 12.23 The code coverage HTML report obtained with the help of Eclipse

12.4 Using JUnit 5 with NetBeans

NetBeans is an IDE that may be used by developers for writing code in a few programming languages, including Java. For brief instructions about the NetBeans installation, please look at appendix C.

We'll pass through a similar demonstration as for IntelliJ IDEA and Eclipse: we have made a selection of the JUnit 5 tests from chapter 2 that are significant in the context of the usage from inside an IDE: displaying names, nested, parameterized, dynamic, repeated and dynamic tests, using tags.

You may launch NetBeans using one of the executables from the *netbeans/bin* folder, either *netbeans* or *netbeans64*, depending on the operating system. You can open a project by selecting *File -> Open Project*, then choose the folder where the project is located. The IDE will look like in figure 12.24.

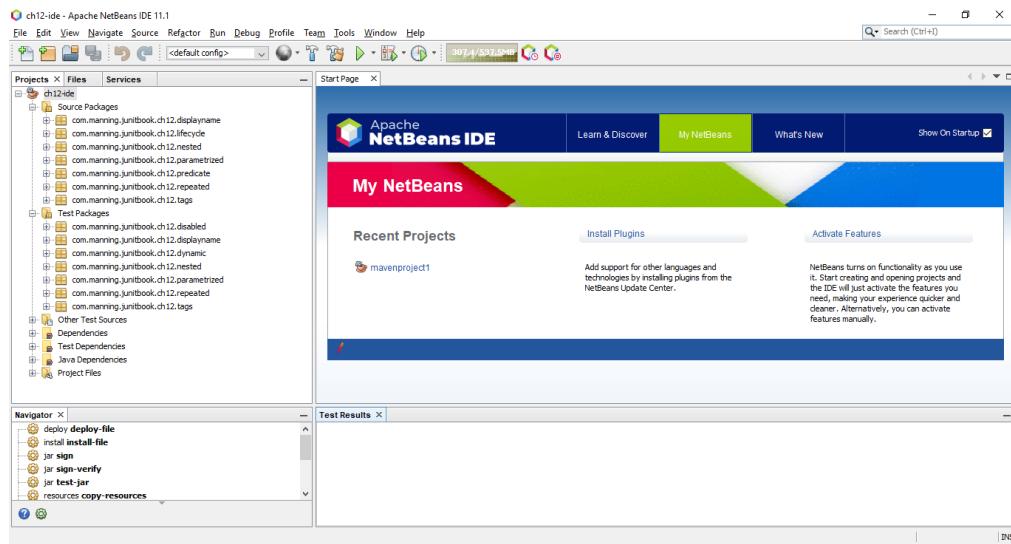


Fig 12.24 NetBeans with an open JUnit 5 project

From here, you may execute all tests by right-clicking on the project, then choosing *Test*. You will get a result like the one shown in figure 12.25. You would most likely want to periodically run all the tests while you are working on the implementation of a particular feature – you would like to make sure that the full project is working fine, and that your implementation does not affect the previously existing functionality.

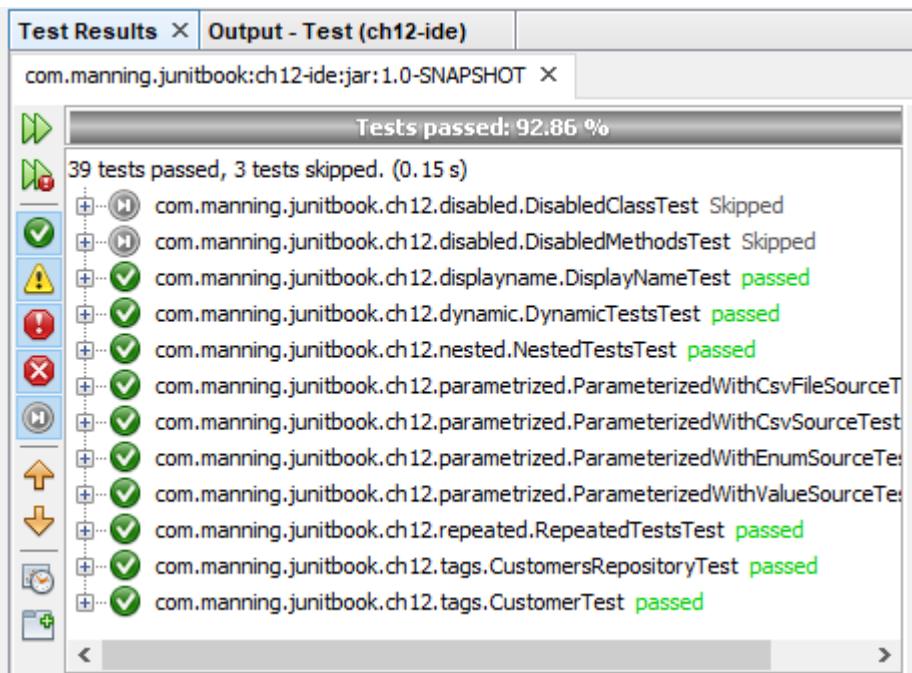


Fig 12.25 Executing all tests from inside NetBeans

Let's have a look at how to run, with the help of NetBeans, these particular test types: tests displaying their names, nested, parameterized, dynamic, repeated and dynamic tests, tests using tags. In figure 12.25 you see that the disabled tests are shown in gray, and they are marked as "skipped" after executing the whole suite. In contrast to the capabilities of IntelliJ IDEA, you cannot force the execution of a particular disabled test.

When you run the tests that are annotated with `@DisplayName`, they will be shown as in figure 12.26. Unlike IntelliJ IDEA and Eclipse, NetBeans is not able to use the information provided by this annotation and is simply showing the names of tests that have been run. Details about the functionality of this annotation are to be found in chapter 2.

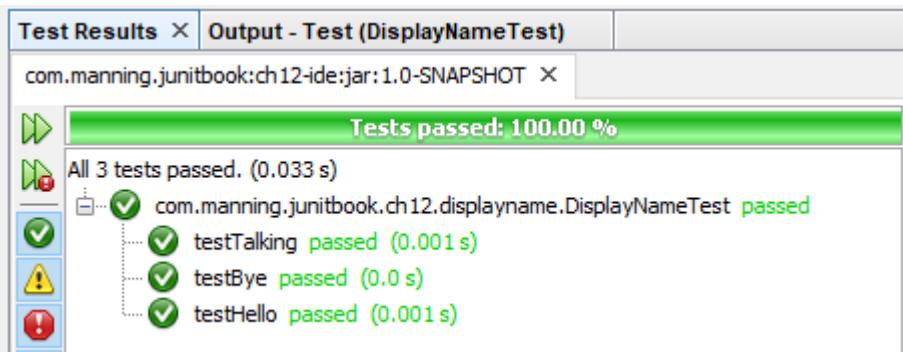


Fig 12.26 Running a test annotated with @DisplayName from IntelliJ

When you run the dynamic tests that are annotated with `@TestFactory`, they will be shown as in figure 12.27. Unlike IntelliJ IDEA and Eclipse, NetBeans is not able to use the names of the dynamically generated tests and will simply number them. Details about the functionality of this annotation are to be found in chapter 2.

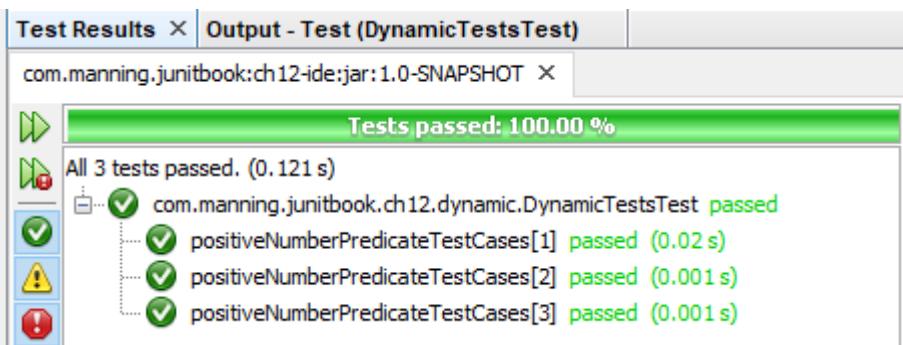


Fig 12.27 Running a dynamic test from NetBeans

When you run the nested tests, they will not be even recognized by NetBeans, as shown in figure 12.28. Details about the functionality of the `@Nested` annotation are to be found in chapter 2.

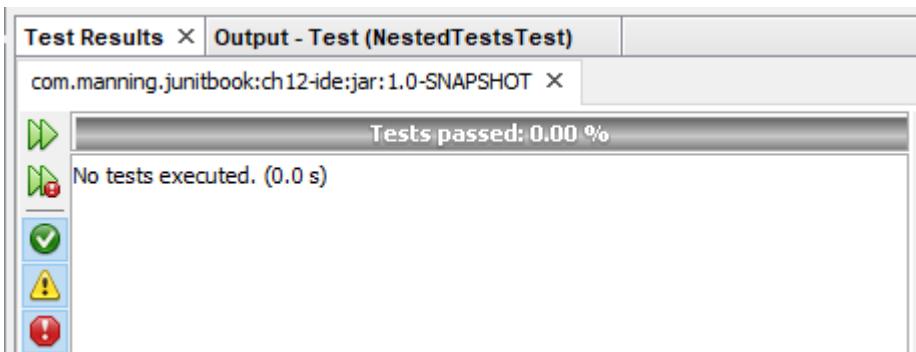


Fig 12.28 Running nested tests from NetBeans

When you run parameterized tests, they will be displayed only with numbers, with no details about the parameters that are received, unlike IntelliJ IDEA and Eclipse. This is shown in figure 12.29. Details about the functionality of the parameterized test annotations are to be found in chapter 2.

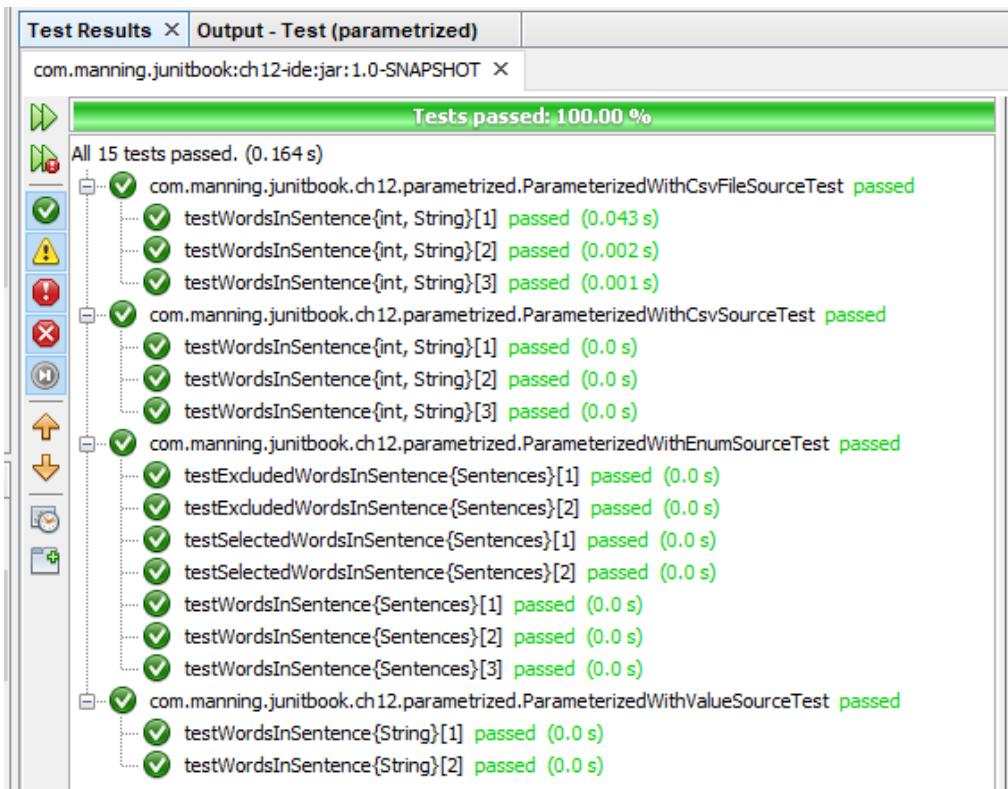


Fig 12.29 Running parameterized tests from NetBeans

When you run repeated tests, they will be shown without any of the required information about the currently repeated test, as it happens in IntelliJ IDEA and in Eclipse. You can see the results in figure 12.30. Details about the functionality of the `@RepeatedTest` annotation are to be found in chapter 2.

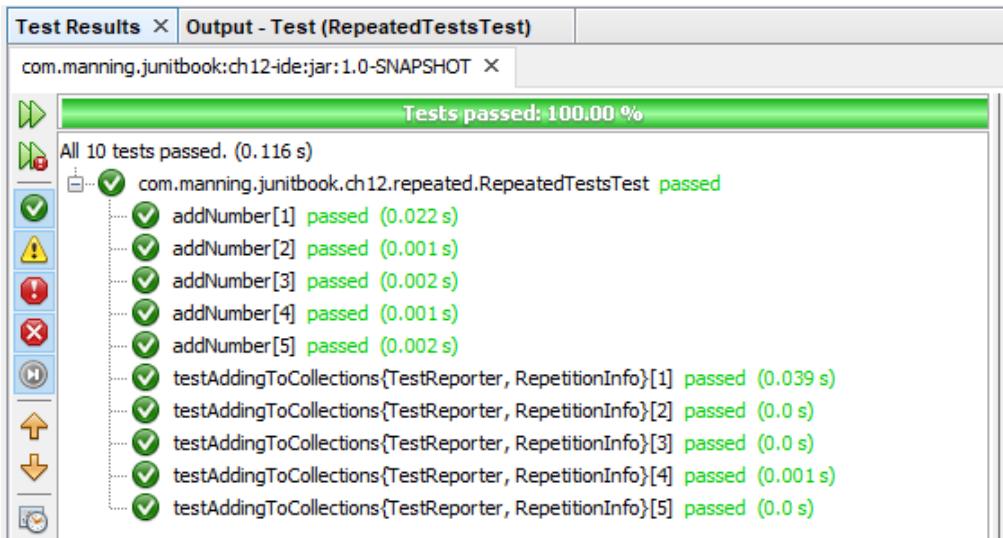


Fig 12.30 Running repeated tests from NetBeans

When running the whole tests suite from NetBeans, the tagged tests are also run. If you would like to run only some particular tagged tests, the IDE will not directly provide this option. If you are running a Maven project, you have to do changes at the level of the *pom.xml* file, with the help of the Surefire plugin. This is a plugin used during the test phase of a project to execute the unit tests of an application. It may filter some of the executed tests or generate reports. A possible configuration is shown in listing 12.1, including the tests tagged with *individual* and excluding the tests tagged with *repository*. Details about the functionality of the @Tag annotation are to be found in chapter 2.

Listing 12.1 A possible filtering configuration of the Maven Surefire plugin

```
<build>
  <plugins>
    <plugin>
      <artifactId>maven-surefire-plugin</artifactId>
      <version>2.22.2</version>
      <configuration>
        <groups>individual</groups>
        <excludedGroups>repository</excludedGroups>
      </configuration>
    </plugin>
  </plugins>
</build>
```

It is also not possible to directly run tests with code coverage from NetBeans. This is possible with the help of JaCoCo, an open-source toolkit for measuring and reporting Java code coverage. If you are running a Maven project, you have to do changes at the level of the

pom.xml file, with the help of the JaCoCo plugin. The configuration of the JaCoCo plugin is shown in listing 12.2.

Listing 12.2 The configuration of the JaCoCo plugin

```
<plugin>
  <groupId>org.jacoco</groupId>
  <artifactId>jacoco-maven-plugin</artifactId>
  <version>0.7.7.201606060606</version>
  <executions>
    <execution>
      <goals>
        <goal>prepare-agent</goal>
      </goals>
    </execution>
    <execution>
      <id>report</id>
      <phase>prepare-package</phase>
      <goals>
        <goal>report</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

After you build the project (right-click, *Build*), the menu item for Code Coverage menu item appears when you right-click again on the project (figure 12.31). You may choose *Show Report...* (figure 12.32) or you may investigate at the level of the individual classes (figure 12.33).

Using NetBeans, you may get feedback about the code coverage, but you need to use some separate plugins, as we have demonstrated with JaCoCo.

At Tested Data Systems Inc., the developers that have decided to use NetBeans for their projects are generally the ones that were using it for a long time and have moved very late to JUnit 5 (this IDE was the last one to introduce JUnit 5 support). Overall, they are not too interested in the user-friendly behavior of the new features of the testing framework.

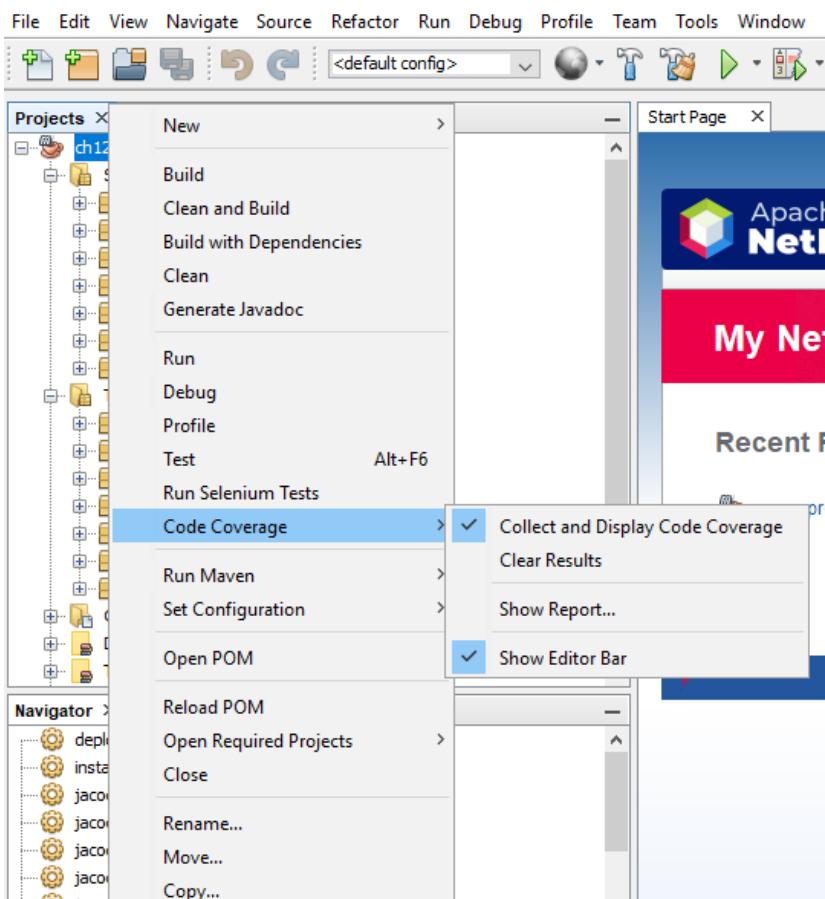


Fig 12.31 The newly added Code Coverage menu

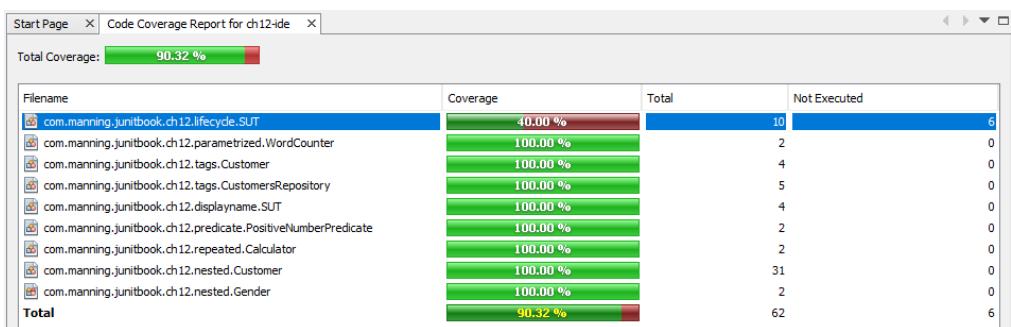


Fig 12.32 The code coverage as shown from NetBeans, with the help of the JaCoCo plugin

```

21 package com.manning.junitbook.ch12.lifecycle;
22
23
24 public class SUT {
25     private String systemName;
26
27     public SUT(String systemName) {
28         this.systemName = systemName;
29         System.out.println(systemName + " from class " + getClass().getSimpleName() + " is initializing.");
30     }
31
32     public boolean canReceiveUsualWork() {
33         System.out.println(systemName + " from class " + getClass().getSimpleName() + " can receive usual work.");
34         return true;
35     }
36
37     public boolean canReceiveAdditionalWork() {
38         System.out.println(systemName + " from class " + getClass().getSimpleName() + " cannot receive additional work.");
39         return false;
40     }
41
42     public void close() {
43         System.out.println(systemName + " from class " + getClass().getSimpleName() + " is closing.");
44     }
45
46

```

Code Coverage: 40.00 %

Test All Tests Clear Report... Disable

Fig 12.33 The code coverage at the level of individual classes

12.5 Comparing JUnit 5 usage in IntelliJ, Eclipse, and NetBeans

At the end of this chapter, we may say that we have used and compared three IDEs that may help you developing JUnit 5 projects. We summarize everything in table 12.1. We remind that we have used the latest version of each IDE at the moment of writing this chapter. It is possible that the IDE developers will focus on more integration in the future. IntelliJ IDEA has now the best integration, followed closely by Eclipse. NetBeans neglects most of the capabilities and user-friendly behavior brought by JUnit 5.

Which of the IDEs you will decide to use may be a matter of personal preference or preference inside the project. To make a good decision, you may consider some other capabilities of the IDEs, not only the integration with JUnit 5. You may do some research on sources that present each IDE in detail, starting with the official documentation. Our purpose was to provide the just-in-time information to be able to evaluate IntelliJ IDEA, Eclipse, and NetBeans from the perspective of the relationship with the framework that is the topic of our book, JUnit 5.

Table 12.1 Comparison of the JUnit 5 integration with the IDEs

Feature	IntelliJ IDEA	Eclipse	NetBeans
Force running disabled tests	Yes	No	No
User-friendly show of @DisplayName tests	Yes	Yes	No

User-friendly show of dynamic tests	Yes	Yes	No
User-friendly show of nested tests	Yes	Yes	Does not execute nested tests
User-friendly show of parameterized tests	Yes	Yes	No
User-friendly show of repeated tests	Yes	Yes	No
User-friendly execution of tagged tests	Yes	Yes	Only with the help of Surefire plugin
Running with code coverage from inside the IDE	Yes	Yes	Only with the help of JaCoCo plugin

In the next chapter, we'll investigate the capabilities of JUnit 5 reported to another important software type it is used in conjunction with: the continuous integration tools.

12.6 Summary

This chapter has covered the following:

- Introducing the IDEs, analyzing the importance of their usage together with Java and JUnit 5 to gain productivity.
- Executing JUnit 5 tests from IntelliJ IDEA and examining the capabilities of this IDE to address the features of the testing framework.
- Executing JUnit 5 tests from Eclipse and examining the capabilities of this IDE to address the features of the testing framework.
- Executing JUnit 5 tests from Eclipse and examining the capabilities of this IDE to address the features of the testing framework.
- Comparing IntelliJ IDEA, Eclipse, and NetBeans as three IDE alternatives for working on JUnit 5 projects, focusing on how each of them is able to work with the new capabilities of the testing framework.

13

Continuous integration with JUnit

5

This chapter covers

- Introducing continuous integration
- Customizing and configuring Jenkins
- Practicing continuous integration in a team of developers
- Working on tasks in a continuous integration environment

“Life is a continuous exercise in creative problem solving.”

—Michael J. Gelb

We have implemented, in chapters 10 and 11, ways to automatically execute the tests by using tools such as Maven and Gradle. Our tests were then triggered by the build. Now, it is time to go to the next level – automatically executing the build and the tests at a regular interval of time by using some other popular tools. In this chapter, we will get to know the paradigm of continuous integration and will demonstrate how to schedule your project to be built automatically in a certain period of time.

13.1 Continuous integration testing

Integration tests are usually time-consuming and, as a single developer, you may not have all the different modules built on your machine. Therefore, it makes no sense to run all the integration tests during development time. That is because, at development time, we are only focused on our module and all we want to know is that it works as a single unit. During

development time, we care mostly that, providing the right input data, the module behaves as expected and produces the expected result.

Integrating the execution of JUnit tests as part of your development cycle -

[code : run : test : code]

or

[test : code : run : test]

if you are working TDD is a very important concept in the sense that JUnit tests are unit tests – i.e. they test a single component of your project in isolation. A great deal of the projects, however, have a modular architecture, where different developers of the teamwork on different modules of the project. Each developer takes care of his own module and his own unit-tests to make sure his module is well tested. Our previously introduced company, *Tested Data Systems Inc.*, does not make any exception: for a project that is under development, the tasks are assigned to a few programmers that work independently and test their own code through JUnit 5 tests.

Different modules interact with each other, so we need to have all the different modules get assembled to see how they work together. In order for the application to be test-proven, we need other sorts of tests – integration or functional tests. We have already seen in chapter 5 that these test the interaction between different modules. At *Tested Data Systems Inc.*, the engineers will also need to test the integration of the code and of the modules they have developed independently.

Test-driven-development taught us to test early and to test often. Executing our entire unit, integration, and functional tests every time we make a small change will slow us immensely. To avoid this, we execute the unit tests at development time only – as early and as often as reasonable. What happens with the integration tests?

Integration tests should be executed independently from the development process. The best is to have them executed in a regular interval of time (say 15 minutes). This way, if something gets broken, you will hear about it in the next 15 minutes and there is a better chance for you to fix it.

DEFINITION¹: continuous integration (CI) — "Continuous Integration is a software development practice where members of a team integrate their work frequently, usually each person integrates at least daily – leading to multiple integrations per day. Each integration is verified by an automated build (including test) to

¹ This definition is taken from a marvelous article by Martin Fowler and Matthew Foemmel.

It can be found here: <http://www.martinfowler.com/articles/continuousIntegration.html>

detect integration errors as quickly as possible. Many teams find that this approach leads to significantly reduced integration problems and allows a team to develop cohesive software more rapidly."

To get the integration tests executed at a regular interval of time, we also need to have the modules of the system prepared and built. After the modules are built and the integration tests are executed, we would like to see the results of the execution as quickly as possible.

This way we get to the point that we need a software tool to do all of the following steps automatically:

1. Check out the project from the source control system
2. Build each of the modules and execute all of the unit tests to verify that the different modules work as expected in isolation
3. Execute integration tests to verify that different modules integrate with each other in an expected way
4. Publish the results from the tests executed in step 3

Now, several questions may arise at this point. First, what is the difference between a human executing all these steps and a tool doing so? The answer is: there is no difference and there shouldn't be! Apart from the fact that no one can bear such a job, if you take a close look at the first thing we do in the list above, you see that we simply checkout the project from the source control system. We do that as if we were a new member of the team and we just started with the project – with a clean checkout in an empty folder. Then, before moving on, we want to make sure that all of the modules work properly in isolation because if they do not, it doesn't make much sense to test if they integrate well with the other modules, does it? The last step in the proposed scenario is to notify the developers about the test results. The notification could be done with an email, or simply by publishing the reports from the tests on a web server.

This overall interaction can be seen in figure 13.1. The CI tool interacts with the Source Control System to get the project (1). After that, it uses the build tool that the project is using, and thus builds the project and executes different kinds of tests (2 and 3). Finally (4) the CI tool publishes the results and blows the whistle so that everybody can see it.

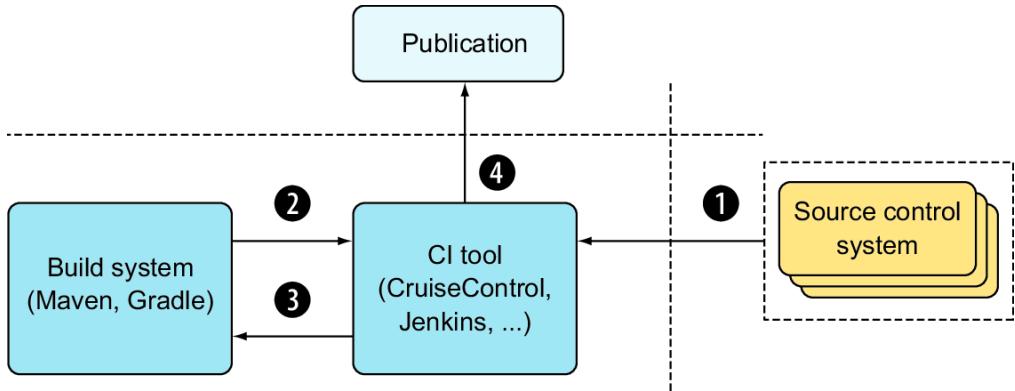


Fig 13.1 Continuous integration scheme: get the project; build it and execute tests; publish the results

The four steps we have listed above are very general and they could be improved a lot. For instance, it would be better to check and see if any changes have been made in the source-control system before we start building. Otherwise, we would waste the CPU power of the machine, knowing for sure that we will get the same results.

Now that we agree we certainly need a tool to continuously integrate our projects, let's start and take a look at one open-source solution we might want to use (it makes no sense reinventing the wheel from scratch when there are good tools already made for us).

13.2 Introducing Jenkins

Jenkins has its roots in a similar continuous integration project named Hudson. Initially developed at Sun Microsystems, Hudson was free software. Oracle bought Sun and intended to make Hudson a commercial version. The majority of the development community decided to continue the project under the name Jenkins in early 2011.

Interest in Hudson strongly decreased after this and Jenkins has replaced it. Since February 2017, Hudson is no longer under maintenance.

13.3 Jenkins customization

Jenkins is a continuous integration tool alternative that is really worth considering. The engineers at *Tested Data Systems Inc.* decided to start the analysis and investigation for using it for some of the projects developed inside the company.

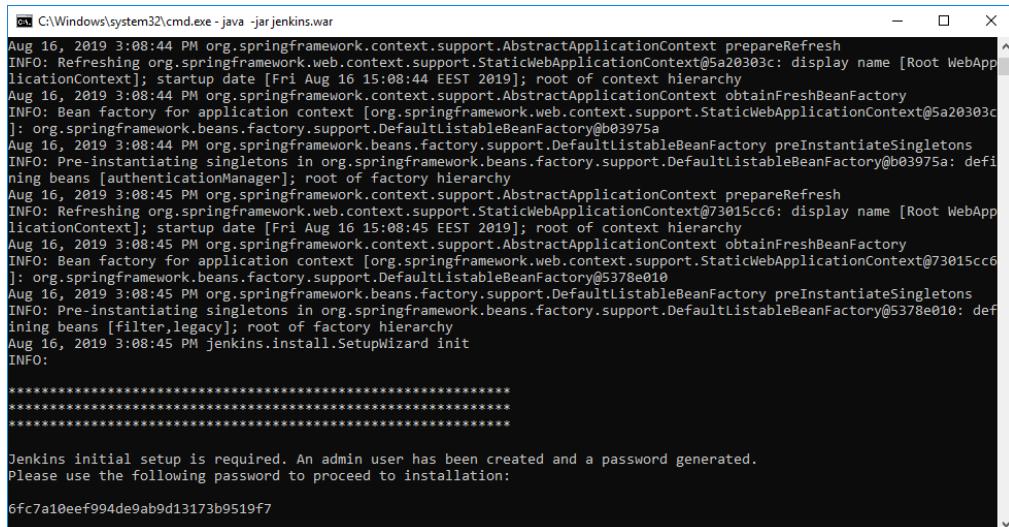
For the installation procedure itself, have a look at appendix D. After the installation is completed, a folder structure is created, from which the most important is the `jenkins.war` file (fig. 13.2).

Program Files (x86) > Jenkins >			
Name	Date modified	Type	Size
users	8/15/2019 5:22 PM	File folder	
war	8/15/2019 5:14 PM	File folder	
workflow-libs	8/15/2019 5:19 PM	File folder	
.lastStarted	8/15/2019 5:15 PM	LASTSTARTED File	0 KB
.owner	8/15/2019 9:26 PM	OWNER File	1 KB
config.xml	8/15/2019 5:23 PM	XML Document	2 KB
hudson.model.UpdateCenter.xml	8/15/2019 5:14 PM	XML Document	1 KB
hudson.plugins.git.GitTool.xml	8/15/2019 5:19 PM	XML Document	1 KB
identity.key.enc	8/15/2019 5:14 PM	Wireshark capture...	2 KB
jenkins.err.log	8/15/2019 8:49 PM	Text Document	157 KB
jenkins.exe	7/17/2019 6:08 AM	Application	363 KB
jenkins.exe.config	4/5/2015 10:05 AM	CONFIG File	1 KB
jenkins.install.InstallUtil.lastExecVersion	8/15/2019 5:23 PM	LASTEXECVERSIO...	1 KB
jenkins.install.UpgradeWizard.state	8/15/2019 5:23 PM	STATE File	1 KB
jenkins.model.JenkinsLocationConfigura...	8/15/2019 5:22 PM	XML Document	1 KB
jenkins.out.log	8/15/2019 5:14 PM	Text Document	1 KB
jenkins.pid	8/15/2019 5:13 PM	PID File	1 KB
jenkins.telemetry.Correlator.xml	8/15/2019 5:14 PM	XML Document	1 KB
jenkins.war	7/17/2019 6:08 AM	WAR File	75,566 KB
jenkins.wrapper.log	8/15/2019 9:57 PM	Text Document	3 KB
jenkins.xml	7/17/2019 6:08 AM	XML Document	3 KB
nodeMonitors.xml	8/15/2019 5:14 PM	XML Document	1 KB
secret.key	8/15/2019 5:14 PM	KEY File	1 KB
secret.key.not-so-secret	8/15/2019 5:14 PM	NOT-SO-SECRET ...	0 KB

Fig 13.2 The Jenkins installation folder with the jenkins.war file

You can start the server from the Jenkins installation folder executing the following command (fig. 13.3):

```
java -jar jenkins.war
```



```

C:\Windows\system32\cmd.exe - java -jar jenkins.war
Aug 16, 2019 3:08:44 PM org.springframework.context.support.AbstractApplicationContext prepareRefresh
INFO: Refreshing org.springframework.web.context.support.StaticWebApplicationContext@5a20303c: display name [Root WebApplicationContext]; startup date [Fri Aug 16 15:08:44 EEST 2019]; root of context hierarchy
Aug 16, 2019 3:08:44 PM org.springframework.context.support.AbstractApplicationContext obtainFreshBeanFactory
INFO: Bean factory for application context [org.springframework.web.context.support.StaticWebApplicationContext@5a20303c]: org.springframework.beans.factory.support.DefaultListableBeanFactory@b03975a
Aug 16, 2019 3:08:44 PM org.springframework.beans.factory.support.DefaultListableBeanFactory preInstantiateSingletons
INFO: Pre-instantiating singletons in org.springframework.beans.factory.support.DefaultListableBeanFactory@b03975a: defining beans [authenticationManager]; root of factory hierarchy
Aug 16, 2019 3:08:45 PM org.springframework.context.support.AbstractApplicationContext prepareRefresh
INFO: Refreshing org.springframework.web.context.support.StaticWebApplicationContext@73015cc6: display name [Root WebApplicationContext]; startup date [Fri Aug 16 15:08:45 EEST 2019]; root of context hierarchy
Aug 16, 2019 3:08:45 PM org.springframework.context.support.AbstractApplicationContext obtainFreshBeanFactory
INFO: Bean factory for application context [org.springframework.web.context.support.StaticWebApplicationContext@73015cc6]: org.springframework.beans.factory.support.DefaultListableBeanFactory@5378e010
Aug 16, 2019 3:08:45 PM org.springframework.beans.factory.support.DefaultListableBeanFactory preInstantiateSingletons
INFO: Pre-instantiating singletons in org.springframework.beans.factory.support.DefaultListableBeanFactory@5378e010: defining beans [filter,legacy]; root of factory hierarchy
Aug 16, 2019 3:08:45 PM jenkins.install.SetupWizard init
INFO:
*****
*****
*****
Jenkins initial setup is required. An admin user has been created and a password generated.
Please use the following password to proceed to installation:
6fc7a10eef994de9ab9d13173b9519f7

```

Fig 13.3 Launching the Jenkins server from the command line using the `java -jar jenkins.war` command

In order to start using the server, we need to navigate to the <http://localhost:8080/> URL. We should see something similar to figure 13.4. We unlock Jenkins using the password generated at the previous step (see fig. 13.3).

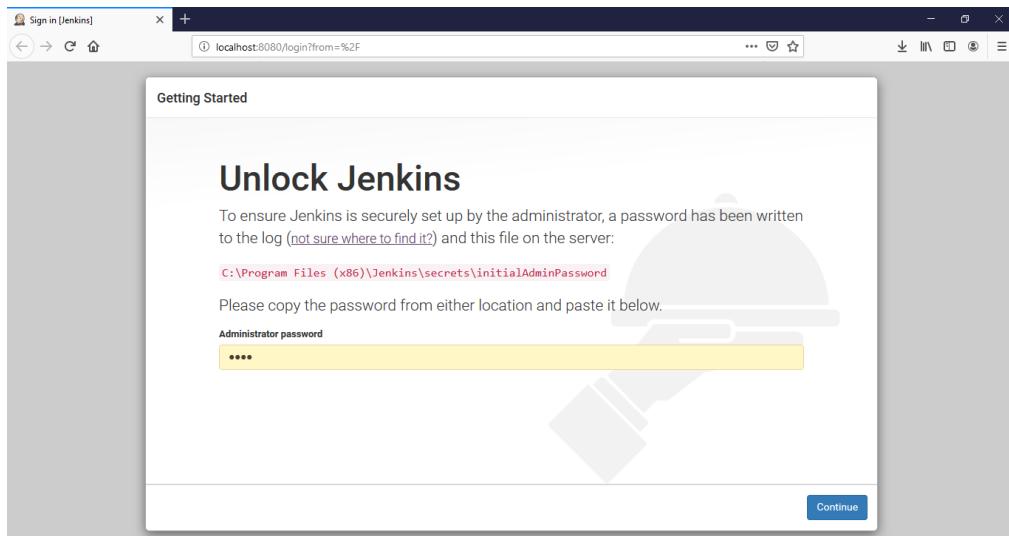


Fig 13.4 Accessing Jenkins from the web interface

After introducing the password, we will get the window from fig. 13.5. We choose to customize Jenkins by installing the most useful plugins, to get a typical configuration. Among these most useful plugins, we mention the folders plugin (for grouping the tasks), the monitoring plugin (for managing charts for CPU, HTTP response time, memory), the metrics plugin (provides health checks). We are not focusing on them for our demonstration, so we simply allow the installation of the suggested ones and move further. For details about the plugins, you may study the Jenkins documentation at <https://jenkins.io/doc/>.

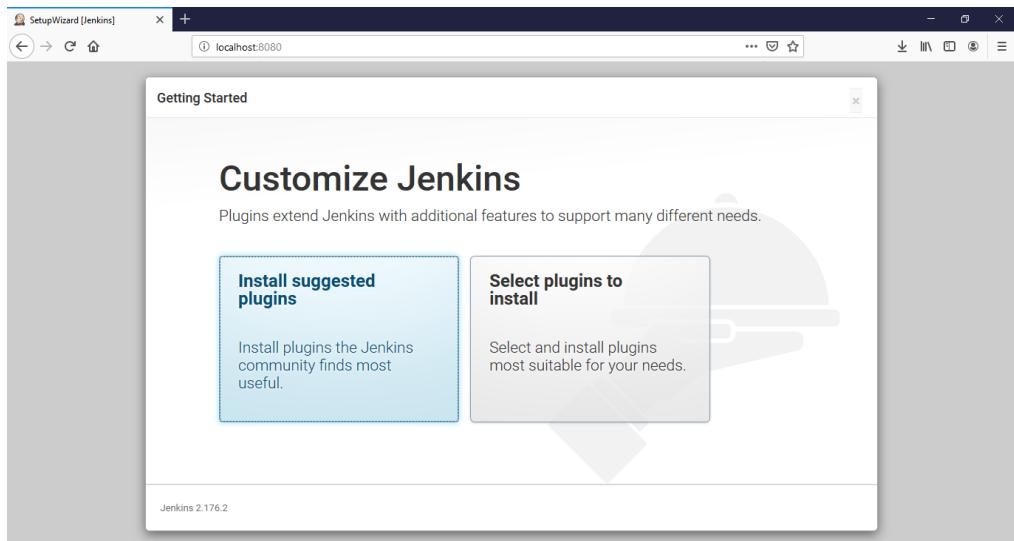


Fig 13.5 Customizing Jenkins window – choosing the plugins to be installed

After the installation of the plugins, we arrive at the window where we are required to create the first admin user by providing our own new credentials (fig. 13.6). Pressing "Save and Continue", we are directed to the window where we may start using Jenkins (fig. 13.7).

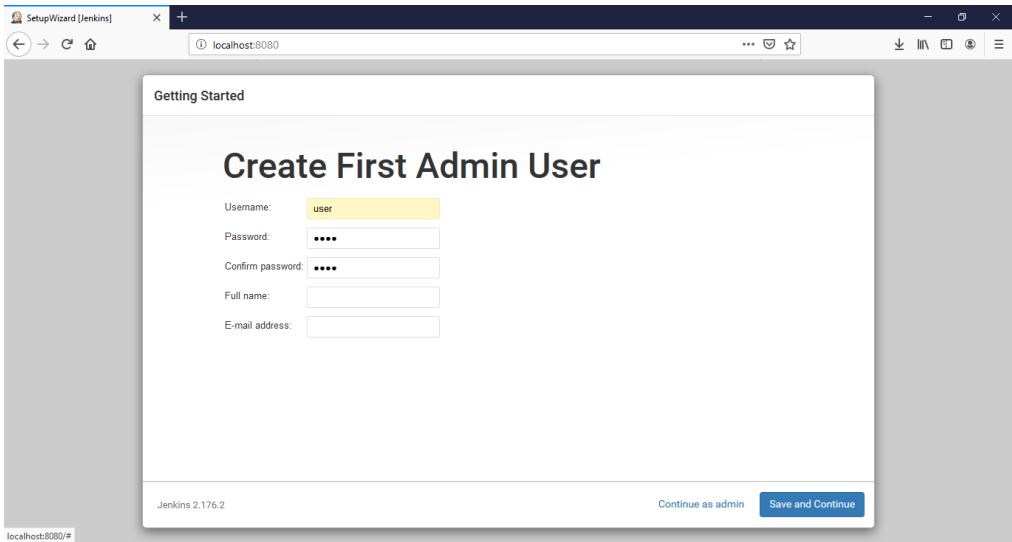


Fig 13.6 Creating the first admin user by providing our own new credentials

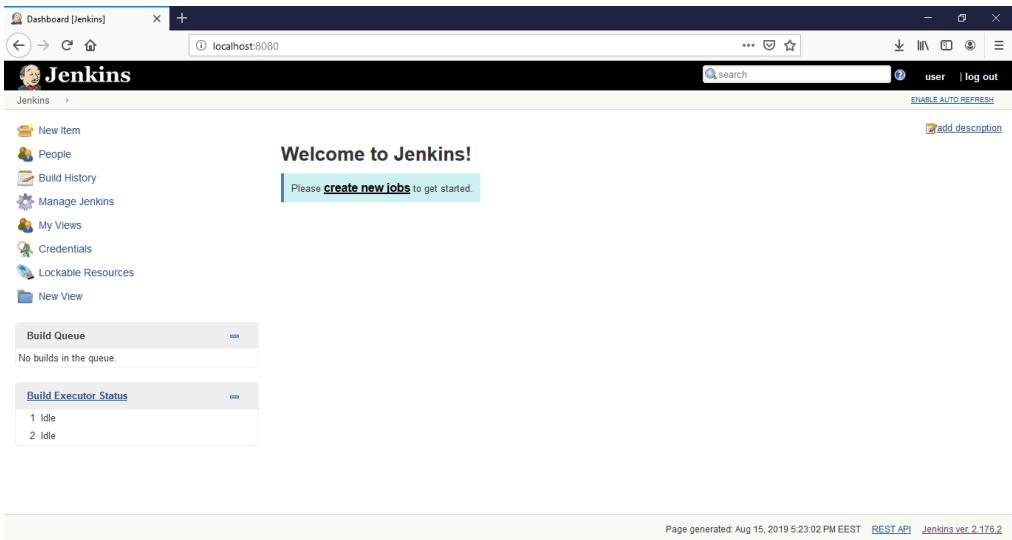


Fig 13.7 The welcome Jenkins page, after the plugins configuration and the creation of the first admin user

13.4 Practicing continuous integration in a team

At *Tested Data Systems Inc.*, one of the projects under development is a flights management application. John and Mike are developers working both on this project, but on different tasks:

©Manning Publications Co. To comment go to [liveBook](#)

John is responsible for developing the part concerning the passengers, while Mike is responsible for developing the part concerning the flights. Both are working independently and write their own code and tests.

Thus, John has implemented the `Passenger` and `PassengerTest` classes.

Listing 13.1 The Passenger class

```
public class Passenger {  
  
    private String identifier;                      #A  
    private String name;                           #A  
    private String countryCode;                     #A  
  
    public Passenger(String identifier, String name, String countryCode) { #B  
        if(!Arrays.asList(Locale.getISOCountries()).contains(countryCode)) { #B  
            throw new RuntimeException("Invalid country code");           #B  
        }                                                               #B  
        this.identifier = identifier;                         #B  
        this.name = name;                                #B  
        this.countryCode = countryCode;                   #B  
    }                                                       #B  
  
    public String getIdentifier() {                  #C  
        return identifier;                          #C  
    }                                                       #C  
  
    public String getName() {                      #C  
        return name;                            #C  
    }                                                       #C  
  
    public String getCountryCode() {                #C  
        return countryCode;                      #C  
    }                                                       #C  
  
    @Override  
    public String toString() {                    #D  
        return "Passenger " + getName() + " with identifier: " +      #D  
               getIdentifier() + " from " + getCountryCode();          #D  
    }                                                       #D  
}
```

In the `Passenger` class from listing 13.1 we see the following:

- A passenger is described through an identifier, a name and a country code (#A).
- The `Passenger` constructor checks the validity of the country code and, if this verification has passed, it sets the previously defined fields: `identifier`, `name`, `countryCode` (#B).
- The class defines getters for the three previously defined fields: `identifier`, `name`, `countryCode` (#C).
- The `toString` method is overridden to allow the information from a passenger to be nicely displayed (#D).

Listing 13.2 The PassengerTest class

```
public class PassengerTest {  
  
    @Test  
    public void testPassengerCreation() {  
        Passenger passenger = new Passenger("123-45-6789",  
                                            "John Smith", "US");  
        assertNotNull(passenger);  
    }  
  
    @Test  
    public void testInvalidCountryCode() {  
        assertThrows(RuntimeException.class,  
                    ()->{  
                        Passenger passenger = new Passenger("900-45-6789",  
                                                "John Smith", "GJ");  
                    });  
    }  
  
    @Test  
    public void testPassengerToString() {  
        Passenger passenger = new Passenger("123-45-6789",  
                                            "John Smith", "US");  
        assertEquals("Passenger John Smith with identifier:  
                    123-45-6789 from US", passenger.toString());  
    }  
}
```

In the PassengerTest class from listing 13.2, we see the following:

- A test checking the creation of a passenger with correct parameters (#A).
- A test checking that the Passenger constructor will throw an exception when the countryCode parameter is invalid (#B).
- A test checking the correct behavior of the `toString` method (#C).

On the other hand, Mike has implemented the Flight and FlightTest classes.

Listing 13.3 The Flight class

```
public class Flight {  
  
    private String flightNumber; #A  
    private int seats; #A  
    private Set<Passenger> passengers = new HashSet<Passenger>(); #A  
  
    private static String flightNumberRegex = "[A-Z]{2}\\d{3,4}$$"; #B  
    private static Pattern pattern = Pattern.compile(flightNumberRegex); #C  
  
    public Flight(String flightNumber, int seats) {  
        Matcher matcher = pattern.matcher(flightNumber); #D  
        if(!matcher.matches()) { #E  
            throw new RuntimeException("Invalid flight number"); #E  
        } #E  
        this.flightNumber = flightNumber; #F  
        this.seats = seats; #F  
    } #F
```

```

public String getFlightNumber() {
    return flightNumber;
} #G
#G
#G

public int getNumberOfPassengers () {
    return passengers.size();
} #G
#G
#G

public boolean addPassenger(Passenger passenger) {
    if(getNumberOfPassengers() >= seats) {
        throw new RuntimeException("Not enough seats for flight "
            + getFlightNumber());
    }
    return passengers.add(passenger);
} #H
#H
#H

public boolean removePassenger(Passenger passenger) {
    return passengers.remove(passenger);
} #I
#I
#I
}

```

In the `Flight` class from listing 13.3, we see the following:

- A flight is described through the flight number, the number of seats and the set of passengers (#A).
- The `flightNumberRegex` provides the regular expression that describes a correctly formed flight number (#B). A regular expression is a sequence of characters that defines a search pattern. In our case, the regular expression requires that a flight starts with two capital letters and is followed by three or four digits. This regular expression is the same for all flights, so the field will be static.
- A `Pattern` instance is created from the string regular expression (#C). The `Pattern` class belongs to the `java.util.regex` package and is the main access point of the Java regular expression API. Whenever you need to work with regular expressions in Java, you start creating such an object. This field is also the same for all flights, so it is static.
- In the constructor of the `Flight` class, we create a `Matcher` object (#D). A `Matcher` is used to search through a text for occurrences of a regular expression defined through a `Pattern`.
- If the `flightNumber` does not match the required regular expression, an exception is thrown (#E). Otherwise, the instance fields are set with the values of the parameters (#F).
- The class defines getters for the flight number and for the passengers number (#G).
- The `addPassenger` method tries to add a passenger to the current flight. If the number of passengers exceeds the number of seats, an exception is thrown. If the passenger is successfully added, it returns `true`. If the passenger already exists, it returns `false` (#H).
- The `removePassenger` method removes a passenger from the current flight. If the

passenger is successfully removed, it returns `true`. If the passenger does not exist, it returns `false` (#I).

Listing 13.4 The FlightTest class

```
public class FlightTest {  
    @Test  
    public void testFlightCreation() {  
        Flight flight = new Flight("AA123", 100);  
        assertNotNull(flight);  
    } #A  
  
    @Test  
    public void testInvalidFlightNumber() {  
        assertThrows(RuntimeException.class,  
            ()->{  
                Flight flight = new Flight("AA12", 100);  
            });  
    } #B
```

In the `FlightTest` class from listing 13.4, we see the following:

- A test checking the creation of a flight with correct parameters (#A).
- A test checking that the `Flight` constructor will throw an exception when the `flightNumber` parameter is invalid (#B).

Additionally, there is an integration test between the `Passenger` and the `Flight` classes written by Mike, as he is taking care of the `Flight` class that also works with `Passenger` objects.

Listing 13.5 The FlightWithPassengersTest class

```
public class FlightWithPassengersTest {  
  
    private Flight flight = new Flight("AA123", 1); #A  
  
    @Test  
    public void testAddRemovePassengers() throws IOException {  
        Passenger passenger = new Passenger("124-56-7890",  
            "Michael Johnson", "US"); #B  
        assertTrue(flight.addPassenger(passenger)); #C  
        assertEquals(1, flight.getNumberOfPassengers()); #C  
  
        assertTrue(flight.removePassenger(passenger)); #D  
        assertEquals(0, flight.getNumberOfPassengers()); #D  
    }  
  
    @Test  
    public void testNumberOfSeats() {  
        Passenger passenger1 = new Passenger("124-56-7890",  
            "Michael Johnson", "US"); #E  
        flight.addPassenger(passenger1); #E  
        assertEquals(1, flight.getNumberOfPassengers()); #F  
  
        Passenger passenger2 = new Passenger("127-23-7991", #G
```

```

        "John Smith", "GB"); #G
    assertThrows(RuntimeException.class, #H
        () -> flight.addPassenger(passenger2)); #H
}
}

```

In the `FlightWithPassengersTest` class from listing 13.5, we see the following:

- The creation of a flight with a correct flight number and 1 place – this is for convenient testing purposes (#A).
- A test creating a passenger (#B) to be added (#C) and removed (#D) from the flight. At each step, we are checking the fact that the operation is successful and the correct number of passengers inside the flight.
- A test creating a passenger to be added to a flight (#E), then another passenger that will exceed the number of seats In the flight (#G). We are first checking the correct number of passengers (#F), then the fact that an exception has been thrown when the number of passengers exceeds the number of seats (#H).

The code developed for this project is managed, on a continuous integration machine, by Git. What you need to know now about Git is that it is a distributed version control system, that tracks changes in source code. It can be downloaded from <https://git-scm.com/downloads> and it has been installed on the continuous integration machine. The folder `Git/cmd`, containing the `git` executable, is on the path of the operating system installed on the continuous integration machine.

For our demonstrations we'll use some basic Git commands, explaining what they are doing and why. For more comprehensive details about Git, you should address dedicated sources.

The folder where the project sources reside is a Git repository. In order this to happen, the machine administrator has run, inside this folder, the following command:

```
git init
```

This command creates a new Git repository. The folder looks like in fig. 13.8. It contains the `pom.xml` Maven file, the `src` folder containing the Java sources, and the `.git` folder, the meta-information for the Git distributed version control system.

Name	Date modified	Type	Size
.git	8/26/2019 6:04 PM	File folder	
src	8/26/2019 3:18 PM	File folder	
pom.xml	8/26/2019 4:16 PM	XML Document	2 KB

Fig 13.8 The source folder on the continuous integration machine, managed by Git

13.5 Configuring Jenkins

Configuring Jenkins is all done through the web interface. We would like to setup a project under its management. As Jenkins is running on the continuous integration machine, we go back to its web interface, accessible at <http://localhost:8080/> (fig. 13.9). No item is defined so far, so we'll press on the "New Item" option from the left side, to create a new continuous integration job.

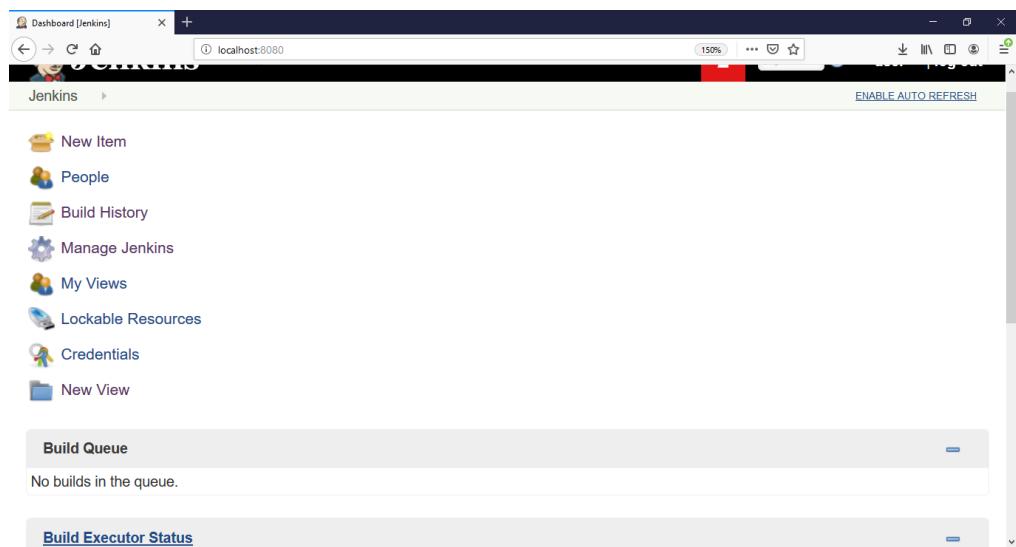


Fig 13.9 The Jenkins main page having no jobs created so far

We choose the *Freestyle project* option, we insert the item name ch13-continuous (fig. 13.10), then we press the *OK* button. On the newly displayed page we choose *Git* as *Source Code Management* option, fill in the *Repository URL* (fig. 13.11), go to the bottom part of the page, choose *Build -> Add Build Step -> Invoke top level Maven targets*, insert *clean install* (fig. 13.12) and press the *Save* button.

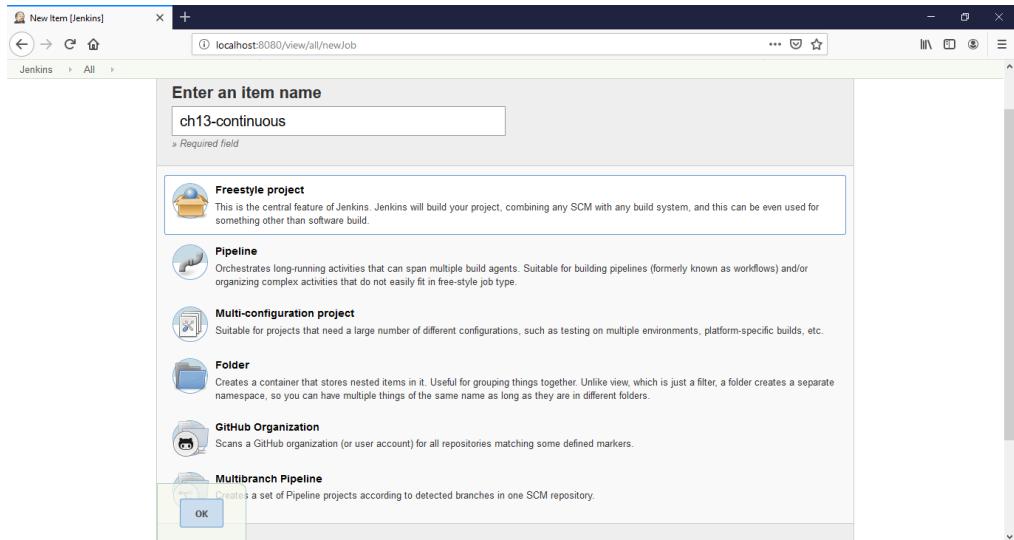


Fig 13.10 Creating a new continuous integration job inside Jenkins

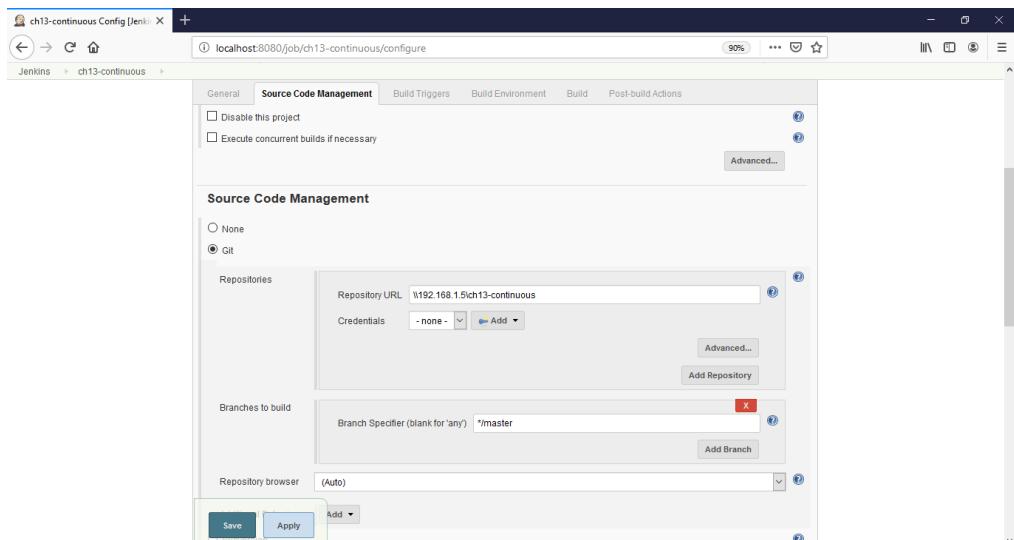


Fig 13.11 Defining the Repository URL containing the source code

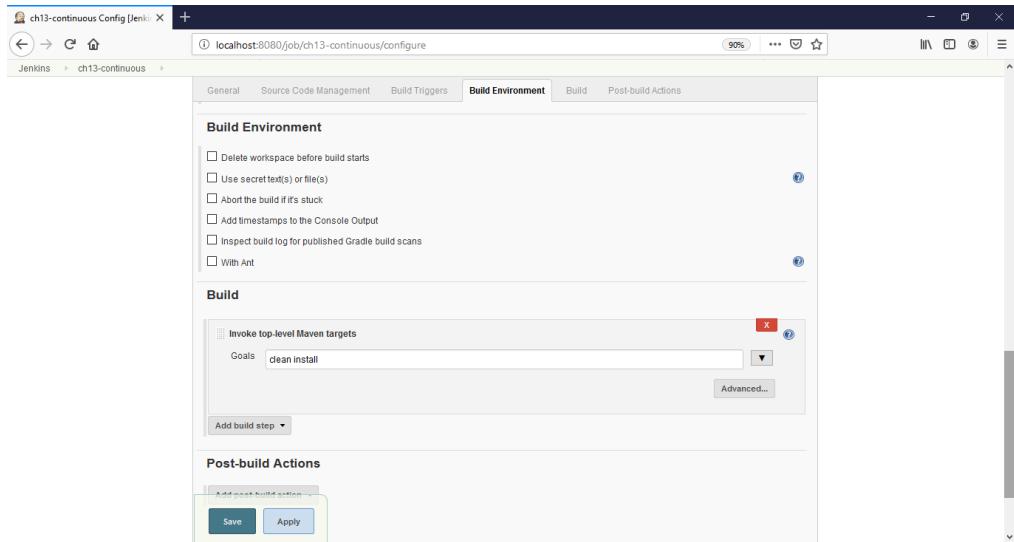


Fig 13.12 The Build configuration for the newly created continuous integration project

On the Jenkins main page, we see now the newly created continuous integration project (fig. 13.13). We will press the *Build* button from the right side of the project and wait for a little time for the project to be executed (fig. 13.14).

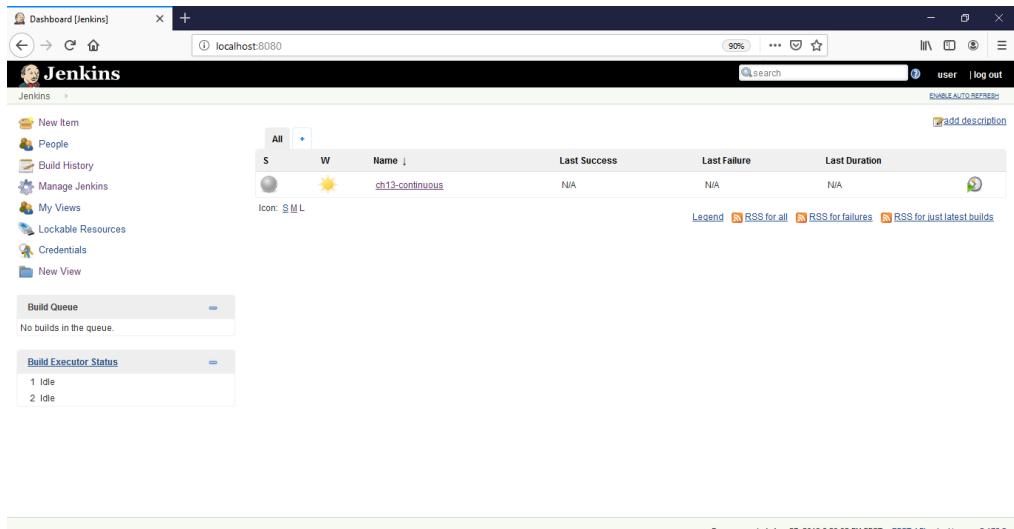


Fig 13.13 The newly created continuous integration project

S	W	Name ↓	Last Success	Last Failure	Last Duration
		ch13-continuous	4 mo 6 days - #3	4 mo 6 days - #2	12 sec

Icon: S M L Legend: RSS for all RSS for failures RSS for just latest builds

Fig 13.14 The result of the first execution of a build on the continuous integration machine, using Jenkins

So, we are sure that everything is fine now from the point of view of continuous integration. The work created by John and Mike runs fine and also integrates well.

We remind what we were explaining at the beginning of this chapter: at development time, a programmer is usually focused on his module and he wants to know that it is working as a single unit. He executes, at development time, only the unit tests. Integration tests are executed independently from the development process – and now we have a fully configured Jenkins project to take care of this.

We'll demonstrate how continuous integration will help John and Mike quickly put head to head their work and how they can easily fix the integration problems.

13.6 Working on tasks in a continuous integration environment

There is a new task coming to John for the flights management project. This task requires that the passenger is able to join a flight by himself, and not only be added to the flight. This is necessary as the new interactive system will allow the passenger to make the individual choice of the flight. Currently, we can add passengers to flights, but if we look at a `Passenger` object, we have no way to know which flight he is on.

In order to address this task, John considers introducing a `Flight` as an instance variable of the `Passenger` class. This way, a passenger will be able to individually choose and join a flight, as the new interactive system requires. Please note, there is a bidirectional reference between the `Flight` and the `Passenger` classes. The `Flight` class already contains the set of `Passengers`:

```
private Set<Passenger> passengers = new HashSet<>();
```

John adds a test into the `FlightWithPassengersTest` class and modifies the existing `testAddRemovePassengers` (listing 13.6):

Listing 13.6 The `testPassengerWithFlight` from the `FlightWithPassengersTest` class

```
@Test
public void testPassengerJoinsFlight() {
    Passenger passenger = new Passenger("123-45-6789",
                                         "John Smith", "US"); #A
    Flight flight = new Flight("AA123", 100); #B
    passenger.joinFlight(flight); #C
    assertEquals(flight, passenger.getFlight()); #D
    assertEquals(1, flight.getNumberOfPassengers()); #E
}

@Test
public void testAddRemovePassengers() throws IOException {
    Passenger passenger = new Passenger("124-56-7890",
                                         "Michael Johnson", "US");
    flight.addPassenger(passenger);
    assertEquals(1, flight.getNumberOfPassengers()); #F
    assertEquals(flight, passenger.getFlight());

    flight.removePassenger(passenger);
    assertEquals(0, flight.getNumberOfPassengers());
    assertEquals(null, passenger.getFlight()); #G
}
```

In the test from listing 13.6, John is doing the following:

- He is creating a passenger (#A) and a flight (#B).
- He makes the passenger join the flight (#C).
- He is checking that the passenger has the previously defined flight assigned to it (#D) and that the number of passengers from the flight is 1 now (#E).
- In the existing `testAddRemovePassengers`, after the flight has added a passenger, check if the flight has been set on the passenger's side (#F). After the flight has removed a passenger, there is no flight set on the passenger's side (#G).

John is also adding this piece of code to the `Passenger` class (listing 13.7):

Listing 13.7 The changes to the `Passenger` class

```
[...]
private Flight flight; #A
[...]

public Flight getFlight() {
    return flight; #B
    #B
}

public void setFlight(Flight flight) { #C
    this.flight = flight;
    #C
}

public void joinFlight(Flight flight) { #C
```

```

Flight previousFlight = this.flight;
if (null != previousFlight) {
    if(!previousFlight.removePassenger(this)) {
        throw new RuntimeException("Cannot remove passenger");
    }
}
setFlight(flight);
if(null != flight) {
    if(!flight.addPassenger(this)) {
        throw new RuntimeException("Cannot add passenger");
    }
}
}

```

In the code from listing 13.7, John is doing the following:

- He is adding a `Flight` field to the `Passenger` class (#A).
- He creates a getter (#B) and a setter (#C) for the newly added field.
- In the `joinFlight` method, he checks if a previous flight exists for the passenger and removes the passenger from it. If the removal is not successful, it throws an exception (#D). Then, it sets the flight for the passenger (#E). If the new flight is not null, adds the passenger to it. If the passenger cannot be added, it throws an exception (#F).

John will push his code to the continuous integration server located at 192.168.1.5. The local project is also under the management of a Git server. It is a clone of the code on the continuous integration server. This clone has been originally created using this Git command:

```
git clone \\192.168.1.5\ch13-continuous
```

In order to do push the code, he will execute a few Git commands:

```
git add *.java
```

The `git add` command means that you want to include updates to a particular file in the next commit. Executing the command above, we'll include all `.java` files that have been changed for the next commit. Changes are not actually recorded until we run `git commit`.

So, John will record the changes into his local repository by running the following command:

```
git commit -m "Allow the passenger to make the individual choice of a flight"
```

The changes have been committed to the local repository with an informative message explaining the task the changes belong to: "Allow the passenger to make the individual choice of a flight".

Now, there is just one more thing that John needs to do in order for the code to arrive on the continuous integration server. He will execute this command:

```
git push
```

After pushing the code to the continuous integration server, a new build is launched on this machine, and it will fail (fig. 13.15). By accessing the console of the project from Jenkins (click

on the ch13-continuous link, then on the build number in the Build History, then on Console Output), you will see that it is the modified test that fails (fig. 13.16).

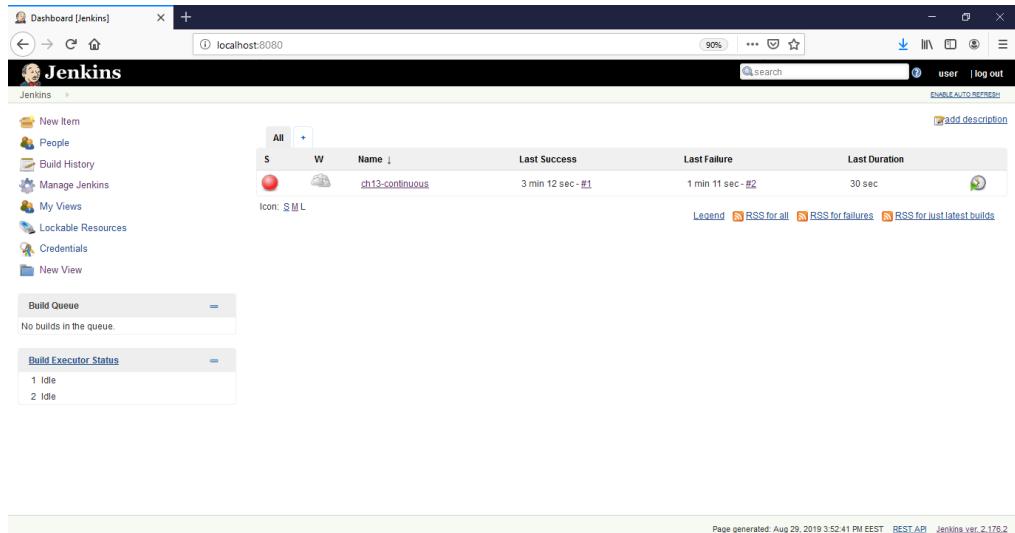


Fig 13.15 The result of the execution of the build after pushing the changes to the tests

```

Downloaded from central: https://repo.maven.apache.org/maven2/org/junit/platform/junit-platform-commons/1.3.1/junit-platform-commons-1.3.1.jar
(75 kB at 157 kB/s)
Downloaded from central: https://repo.maven.apache.org/maven2/org/junit/platform/junit-platform-launcher/1.3.1/junit-platform-launcher-1.3.1.jar (95 kB at 219 kB/s)
Downloaded from central: https://repo.maven.apache.org/maven2/org/apache/maven/surefire/surefire-junit-platform/2.22.1/surefire-junit-platform-2.22.1.jar (66 kB at 146 kB/s)
Downloaded from central: https://repo.maven.apache.org/maven2/org/junit/platform/junit-platform-engine/1.3.1/junit-platform-engine-1.3.1.jar
(13 kB at 288 kB/s)
[INFO] 
[INFO] 
[INFO] T E S T S
[INFO] -----
[INFO] Running com.manning.junitbook.ch13.flights.FlightTest
[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.022 s - in com.manning.junitbook.ch13.flights.FlightTest
[INFO] Running com.manning.junitbook.ch13.flights.passenger.FlightWithPassengerTest
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0 s <<< FAILURE! >>>
[INFO] com.manning.junitbook.ch13.flights.passenger.FlightWithPassengerTest
[INFO] testAddRemovePassengers Time elapsed: 0 s <<< FAILURE!
[ERROR] testAddRemovePassengers Error: expected: <com.manning.junitbook.ch13.flights.Flight@c46bcd4> but was: <null>
[INFO] at com.manning.junitbook.ch13.flights.passenger.FlightWithPassengerTest.testAddRemovePassengers(FlightWithPassengerTest.java:29)
[INFO] 
[INFO] Running com.manning.junitbook.ch13.passengers.PassengerTest
[INFO] Tests run: 3, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0 s - in com.manning.junitbook.ch13.passengers.PassengerTest
[INFO] 
[INFO] Results:
[INFO] 
[INFO] Failures:
[INFO]   FlightWithPassengerTest.testAddRemovePassengers:29 expected: <com.manning.junitbook.ch13.flights.Flight@c46bcd4> but was: <null>
[INFO] 
[INFO] Tests run: 8, Failures: 1, Errors: 0, Skipped: 0
[INFO] 
[INFO] -----
[INFO] BUILD FAILURE
[INFO] -----
[INFO] Total time: 34.761 s
[INFO] Finished at: 2019-08-29T15:52:09+03:00
[INFO] 
[ERROR] Failed to execute goal org.apache.maven.plugins:maven-surefire-plugin:2.22.1:test (default-test) on project ch13: There are test

```

Fig 13.16 The console of the Jenkins project showing the failure after running the build

The error message says:

```
[INFO] Running com.manning.junitbook.ch13.flights.passengers.FlightWithPassengersTest
[ERROR] Tests run: 3, Failures: 1, Errors: 0, Skipped: 0, Time elapsed: 0 s <<< FAILURE! - in
      com.manning.junitbook.ch13.flights.passengers.FlightWithPassengersTest
[ERROR] testAddRemovePassenger  Time elapsed: 0 s  <<< FAILURE!
org.opentest4j.AssertionFailedError: expected:
<com.manning.junitbook.ch13.flights.Flight@c46bcd4> but was: <null>
  at
    com.manning.junitbook.ch13.flights.passengers.FlightWithPassengersTest.testAddRemovePassenger(FlightWithPassengersTest.java:29)
```

After the build fails and the console output is investigated, the programmers will realize that it is a problem of integration between Flight and Passenger. We have to make sure that the relationship between Passenger and Flight is bidirectional: if the Passenger has a reference to a Flight, the Flight will also have a reference to the Passenger, and vice-versa.

Mike is the developer working on the Flight, so he will have to take care of this. Mike's local project is also under the management of Git. It is a clone of the code on the continuous integration server. This clone has been originally created using this Git command:

```
git clone \\192.168.1.5\ch13-continuous
```

In order to do get the code updated from John, Mike will execute the following Git command:

```
git pull
```

This way, Mike will have the last updates on his machine. Fixing the issue, Mike will modify the existing Flight class, more exactly the addPassenger and removePassenger methods (listing 13.8):

Listing 13.8 The modified Flight class

```
public boolean addPassenger(Passenger passenger) {
    if(getNumberOfPassengers() >= seats) {
        throw new RuntimeException("Not enough seats for flight "
            + getFlightNumber());
    }
    passenger.setFlight(this);                                     #A
    return passengers.add(passenger);
}

public boolean removePassenger(Passenger passenger) {
    passenger.setFlight(null);                                     #B
    return passengers.remove(passenger);
}
```

Mike has added two lines of code to do the following:

- When a passenger is added to a flight, the flight is also set on the passenger's side (#A).
- When a passenger is removed from a flight, the flight is also removed from the passenger's side (#B).

Mike will send his changes to the continuous integration server by executing the following Git commands:

```
git add *.java  
git commit -m "Adding integration code for a passenger join/unjoin"  
git push
```

After pushing the code to the continuous integration server, a new build is launched on this machine, and it will succeed (fig. 13.17).

The screenshot shows the Jenkins dashboard at localhost:8080. The main area displays a table of builds. One build, named 'ch13-continuous', is listed with the following details:

S	W	Name	Last Success	Last Failure	Last Duration
Icon: S	Icon: W	ch13-continuous	1 min 31 sec - #3	2 min 36 sec - #2	12 sec

Below the table, there are links for RSS feeds: 'RSS for all', 'RSS for failures', and 'RSS for just latest builds'. On the left sidebar, under 'Build History', there is a link to 'ch13-continuous'. At the bottom of the page, a footer note says 'Page generated: Aug 29, 2019 4:17:03 PM EEST [REST API](#) Jenkins ver. 2.176.2'.

Fig 13.17 The Jenkins build after having introduced the necessary integration code into the Flight class

So, we see the benefits of continuous integration: the integration problems will be quickly signaled and the developers will be able to immediately fix the issues. JUnit 5 and Jenkins are simply cooperating very effectively!

The next chapter will start the fourth part of the book, dedicated to the work with modern frameworks and JUnit 5. We'll meet, for the beginning, the JUnit 5 extension model.

13.7 Summary

This chapter has covered the following:

- The concept of continuous integration has been introduced as a software development practice where members of a team integrate their work frequently.
- The benefits that continuous integration is providing for developing team Java applications have been examined: each integration is verified by an automated build and will detect integration errors as quickly as possible; the integration problems are

heavily reduced; in case they exist, the developers will be able to immediately fix the issues.

- Jenkins has been introduced as a continuous integration tool to be used inside a collaborative project. Jenkins builds use JUnit 5 for execution.
- The collaborative work in a team practicing continuous integration has been demonstrated: working on implementing the current tasks, using Jenkins as a continuous integrations server and Git as a version control system.

14

JUnit 5 extension model

This chapter covers

- Introducing the JUnit 5 extension model
- Creating JUnit 5 extensions
- Implementing JUnit 5 tests using available extension points
- Developing an application with tests extended by JUnit 5 extensions

The wheel . . . is an extension of the foot. The book . . . is an extension of the eye. . . . Clothing, an extension of the skin. . . . Electric circuitry, an extension of the central nervous system.

—Marshall McLuhan

In chapter 4, we demonstrated ways to extend the execution of the tests. We set the JUnit 4 rules and JUnit 5 extensions face to face. We analyzed how to make the migration from the old JUnit 4 rules to the new extensions model developed by JUnit 5. We emphasized a few well-known extensions such as `MockitoExtension` and `SpringExtension`. In chapter 8, we implemented tests using mock objects and `MockitoExtension`, and we'll implement more tests using `SpringExtension` in chapter 16.

In this chapter, we demonstrate the systematic creation of custom extensions and their applicability to the creation of JUnit 5 tests.

14.1 Introducing the JUnit 5 extension model

Although JUnit 4 provides extension points through runners and rules (chapter 3), the JUnit 5 extension model consists of a single concept: the `Extension API`. Extension itself is just a *marker interface* (or *tag* or *token interface*)—an interface with no fields or methods inside. It is

used to mark the fact that the class implementing an interface of this category has some special behavior. Among the best-known, Java marker interfaces are `Serializable` and `Cloneable`.

JUnit 5 can extend the behavior of classes or methods of test and these extensions can be reused by many tests.

A JUnit 5 extension is connected to the occurrence of some particular event during the execution of a test. This kind of event is called an extension point. At the moment when such a point in the lifecycle of a test has been reached, the JUnit engine automatically calls the registered extension.

The list of available extension points is made of:

- *Conditional test execution*—Controls whether a test should be run
- *Life-cycle callbacks*— Reacts to events in the life cycle of a test
- *Parameter resolution*—Resolves at run time the parameter received by a test
- *Exception handling*—Defines the behavior of a test when it encounters certain types of exceptions
- *Test instance postprocessing*—Executed after an instance of a test is created

Please note that the extensions are largely used inside frameworks and build tools. They may also be used for application programming, but not to the same extent. The creation and usage of extensions follow common principles, our demonstration will use examples appropriate for regular applications development.

14.2 Creating the first JUnit 5 extension

Tested Data Systems is developing a flights management application. Harry is working on this project, developing and testing the passengers part. At this moment, the `Passenger` and `PassengerTest` classes look like listings 14.1 and 14.2, respectively.

Listing 14.1 The Passenger class

```
public class Passenger {  
  
    private String identifier; #A  
    private String name; #A  
  
    public Passenger(String identifier, String name) { #B  
        this.identifier = identifier; #B  
        this.name = name; #B  
    } #B  
  
    public String getIdentifier() { #C  
        return identifier; #C  
    } #C  
  
    public String getName() { #C  
        return name; #C  
    } #C
```

```

@Override
public String toString() {
    return "Passenger " + getName() + " with identifier: " +
           getIdentifier();
}
}

```

In this listing:

- A passenger is described through an identifier a name (#A).
- The Passenger constructor sets the identifier and name fields (#B).
- The class defines getters for the identifier and name (#C).
- The `toString` method is overridden to allow the information from a passenger to be nicely displayed, showing the name and the identifier (#D).

Listing 14.2 The PassengerTest class

```

public class PassengerTest {

    @Test
    void testPassenger() throws IOException {
        Passenger passenger = new Passenger("123-456-789", "John Smith");
        assertEquals("Passenger John Smith with identifier: 123-456-789",
                    passenger.toString());
    }
}

```

The `PassengerTest` class in listing 14.2 has a single test that checks the behavior of the `toString` method.

Harry's next task involves the conditional execution of tests depending on the context. There are three types of contexts: regular, low, and peak, depending on the number of passengers from a certain period. The task requires the tests to be executed only in the regular and low contexts. The tests are not executed during the peak periods, as an overloaded system would really cause problems for the company.

To fulfill this task, he will create a JUnit extension that controls whether a test should be run. Then, he extends the tests with the help of this extension. Such an extension is defined by implementing the `ExecutionCondition` interface. Harry will create an `ExecutionContextExtension` class that implements the `ExecutionCondition` interface and overrides the `evaluateExecutionCondition()` method (listing 14.3). The method verifies whether a property representing the current context name equals "regular" or "low" and otherwise disables the test.

Listing 14.3 The ExecutionContextExtension class

```

public class ExecutionContextExtension implements ExecutionCondition { #A

    @Override
    public ConditionEvaluationResult #B
        evaluateExecutionCondition(ExecutionContext context) { #B

```

```

Properties properties = new Properties();                      #C
String executionContext = "";                                #C

try {
    properties.load(ExecutionContextExtension.class
                    .getClassLoader()           #C
                    .getResourceAsStream("context.properties")); #C
    executionContext = properties.getProperty("context");      #C
    if (!"regular".equalsIgnoreCase(executionContext) &&
        !"low".equalsIgnoreCase(executionContext)) {          #D
        return ConditionEvaluationResult.disabled(          #D
            "Test disabled outside regular and low contexts"); #D
    }
} catch (IOException e) {
    throw new RuntimeException(e);
}
return ConditionEvaluationResult.enabled("Test enabled on the "+ #E
                                         executionContext + " context"); #E
}
}

```

in this listing:

- We create a conditional test execution extension by implementing the `ExecutionCondition` interface (#A).
- We override the `evaluateExecutionCondition` method, which returns a `ConditionEvaluationResult` to determine whether a test is enabled (#B).
- We create a `Properties` object that loads the properties from the resource `context.properties` file. We keep the value of the `context` property (#C).
- If the `context` property differs from "regular" or "low", the returned `ConditionEvaluationResult` means that the test is disabled (#D).
- Otherwise, the returned `ConditionEvaluationResult` means that the test is enabled (#E).

The context is configured through the resources/context.properties configuration file:

```
context =regular
```

For the current business logic, the "regular" value means that the tests will be executed in the current context.

The last thing to do is annotate the existing `PassengerTest` with the new extension:

```
@ExtendWith({ExecutionContextExtension.class})
public class PassengerTest {
[...]
```

Because the test executes in the current "regular" context configuration, the test will run just as it previously did. During the peak periods, the context is setup differently (`context=peak`). By executing the test during the peak period, we get the result shown in figure 14.1; the test is disabled, and it gives a reason.

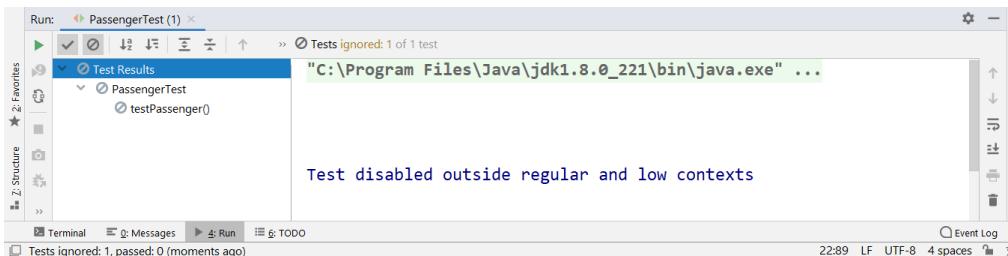


Figure 14.1 The result of `PassengerTest` during the peak period, as extended with `ExecutionContextExtension`

We may instruct the JVM (Java Virtual Machine) to bypass the effects of conditional execution, however, deactivating it by setting the `junit.jupiter.conditions.deactivate` configuration key to a pattern that matches the condition. From the Run -> Edit Configurations menu, we may set `junit.jupiter.conditions.deactivate=*`, for example, and the result will be the deactivation of all conditions (figure 14.2). The result of the execution will not be influenced by any of the conditions, so all tests will run.

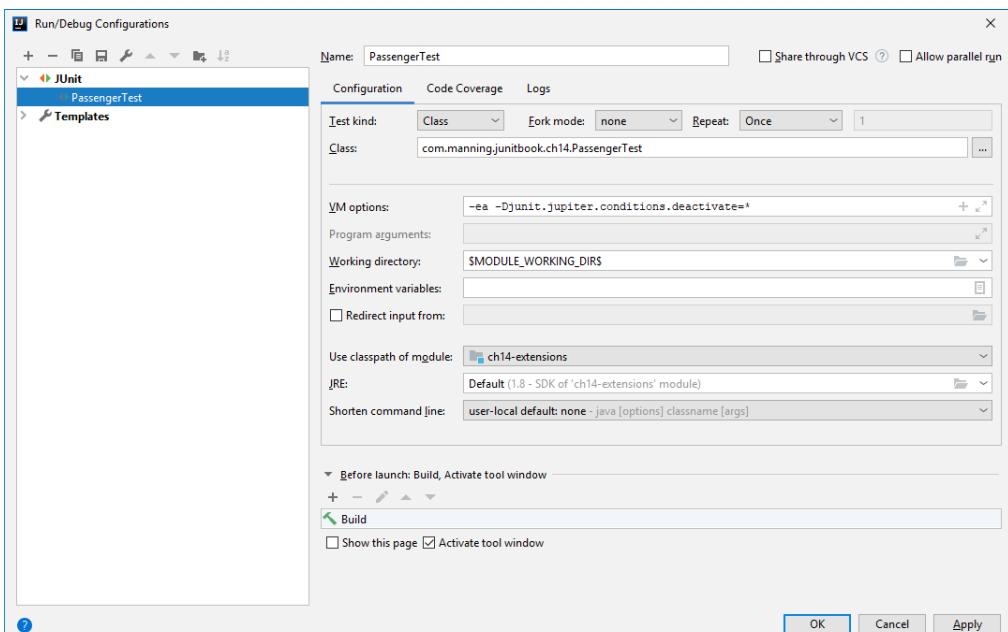


Figure 14.2 Deactivating the conditional execution by setting the `junit.jupiter.conditions.deactivate` configuration key

Harry carried out the conditional execution task by implementing a conditional test execution extension.

14.3 Writing JUnit 5 tests using the available extension points

Harry is responsible for implementing the passenger business logic and testing it, including the persistence of the passengers to a database. This section follows his activity in implementing the tasks with the help of JUnit 5 extensions.

14.3.1 Persisting the passengers to a database

Harry's next task is saving the passengers in a test database. Before the whole test suite is executed, the database must be reinitialized, and a connection to it must be open. At the end of the execution of the suite, the connection to the database must be closed. Before executing a test, we have to set up the database in a known state so we can be sure that its content will be tested correctly. Harry decides to use the H2 database, JDBC, and JUnit 5 extensions.

H2 is a relational database management system developed in Java that permits the creation of an in-memory database. It can also be embedded in Java applications when testing purposes require this.

JDBC (Java Database Connectivity) is a Java API that defines how a client accesses a database. JDBC is part of the Java Standard Edition platform.

To solve the task, the first thing Harry needs to do is add the H2 dependency to the pom.xml file, as shown in listing 14.4.

Listing 14.4 The H2 dependency added to the pom.xml file

```
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <version>1.4.199</version>
</dependency>
```

To manage the connection to the database, Harry implements the ConnectionManager class (listing 14.5).

Listing 14.5 The ConnectionManager class

```
public class ConnectionManager {
    private static Connection connection; #A

    public static Connection getConnection() { #A
        return connection; #A
    } #A

    public static Connection openConnection() { #B
        try {
            Class.forName("org.h2.Driver"); // this is driver for H2 #B
            connection = DriverManager.getConnection("jdbc:h2:~/book", #C
                "sa", // Login #C
                "" // password #C
        }
    }
}
```

```

        );
    return connection;
} catch(ClassNotFoundException | SQLException e) {
    throw new RuntimeException(e);
}

public static void closeConnection() {
    if (null != connection) {
        try {
            connection.close();
        } catch(SQLException e) {
            throw new RuntimeException(e);
        }
    }
}

```

In this listing:

- We declare a `java.sql.Connection` field and a getter to return it (#A).
- In the `openConnection` method, we load the `org.h2.Driver` class, the driver for H2 (#B), and we create a connection to the database with the URL `jdbc:h2:~/passenger` having the default credentials "sa" and " " (#C).
- In the `closeConnection` method, we try to close the previously opened connection (#D).

To manage the database tables, Harry implements the `TablesManager` class (listing 14.6).

Listing 14.6 The TablesManager class

```

public class TablesManager {

    public static void createTable(Connection connection) {          #A
        String sql =          #A
            "CREATE TABLE IF NOT EXISTS PASSENGERS (ID VARCHAR(50), " +          #A
            "NAME VARCHAR(50));";          #A

        executeStatement(connection, sql);          #A
    }          #A

    public static void dropTable(Connection connection) {          #B
        String sql = "DROP TABLE IF EXISTS PASSENGERS;";          #B

        executeStatement(connection, sql);          #B
    }          #B

    private static void executeStatement(Connection connection,          #C
                                         String sql)          #C
    {          #C
        try(PreparedStatement statement =          #C
             connection.prepareStatement(sql))          #C
        {          #C
            statement.executeUpdate();          #C
        } catch (SQLException e) {          #C

```

```

        throw new RuntimeException(e);          #C
    }
}

}

```

in this listing:

- The `createTable` method creates the `PASSENGERS` table in the database, containing and `ID` and a `NAME`, both `VARCHAR(50)` (#A).
- The `dropTable` method drops the `PASSENGERS` table from the database (#B).
- The `utility executeStatement` method executes any `SQL` command against the database (#C).

To manage the execution of the queries against the database, Harry implements the `PassengerDao` interface (listing 14.7) and the `PassengerDaoImpl` class (listing 14.8). A *DAO* (data access object) provides an interface to a database and maps the application calls to specific database operations without exposing the details of the persistence layer.

Listing 14.7 The PassengerDao interface

```

public interface PassengerDao {
    public void insert(Passenger passenger);          #A
    public void update(String id, String name);        #B
    public void delete(Passenger passenger);           #C
    public Passenger getById(String id);              #D
}

```

In this listing, the `insert` (#A), `update` (#B), `delete` (#C), and `getById` (#D) methods declare the operations against the database to be implemented.

Listing 14.8 The PassengerDaoImpl class

```

public class PassengerDaoImpl implements PassengerDao {

    private Connection connection;                  #A

    public PassengerDaoImpl(Connection connection) { #A
        this.connection = connection;                #A
    }

    @Override
    public void insert(Passenger passenger) {        #B
        String sql = "INSERT INTO PASSENGERS (ID, NAME) VALUES (?, ?)";

        try (PreparedStatement statement = connection.prepareStatement(sql)){ #C
            statement.setString(1, passenger.getIdentifier());               #C
            statement.setString(2, passenger.getName());                      #C
            statement.executeUpdate();                                       #D
        } catch (SQLException e) {
            throw new RuntimeException(e);
        }
    }

    @Override

```

```

public void update(String id, String name) {
    String sql = "UPDATE PASSENGERS SET NAME = ? WHERE ID = ?";      #E

    try (PreparedStatement statement = connection.prepareStatement(sql)){
        statement.setString(1, name);                                     #F
        statement.setString(2, id);                                      #F
        statement.executeUpdate();                                       #G
    } catch (SQLException e) {
        throw new RuntimeException(e);
    }
}

@Override
public void delete(Passenger passenger) {
    String sql = "DELETE FROM PASSENGERS WHERE ID = ?";            #H

    try (PreparedStatement statement = connection.prepareStatement(sql)){
        statement.setString(1, passenger.getIdentifier());             #I
        statement.executeUpdate();                                      #J
    } catch (SQLException e) {
        throw new RuntimeException(e);
    }
}

@Override
public Passenger getById(String id) {
    String sql = "SELECT * FROM PASSENGERS WHERE ID = ?";          #K
    Passenger passenger = null;

    try (PreparedStatement statement = connection.prepareStatement(sql)){
        statement.setString(1, id);                                     #L
        ResultSet resultSet = statement.executeQuery();                #M

        if (resultSet.next()) {
            passenger = new Passenger(resultSet.getString(1),           #N
                                         resultSet.getString(2));           #N
        }
    } catch (SQLException e) {
        throw new RuntimeException(e);
    }

    return passenger;                                              #O
}
}

```

In this listing:

- The `Connection` field is kept inside the class and provided as an argument of the constructor (#A).
- The `insert` method declares the SQL query to be executed (#B), sets the identifier and the name of the passenger as parameters of the query (#C), and executes the query (#D).
- The `update` method declares the SQL query to be executed (#E), sets the identifier and the name of the passenger as parameters of the query (#F), and executes the query (#G).

- The `delete` method declares the SQL query to be executed (#H), sets the identifier of the passenger as a parameter of the query (#I), and executes the query (#J).
- The `getById` method declares the SQL query to be executed (#K), sets the identifier of the passenger as a parameter of the query (#L), executes the query (#M), and creates a new `Passenger` object with the parameters returned from the database (#N). Then the new object is returned by the method (#O).

Now Harry needs to implement the JUnit 5 extension to do the following:

- Before the execution of the whole test suite, reinitialize the database and open a connection to it
- At the end of the execution of the suite, close the connection to the database
- Before executing a test, make sure that the database is in a known state so that the developer can be sure that its content is tested correctly

Looking at the requirements, which ask for actions based on the life cycle of the test suite, it is natural for Harry to choose to implement the life-cycle callbacks. In order to implement the extensions concerning the test lifecycle, Harry must implement the following interfaces:

- `BeforeEachCallback` and `AfterEachCallback`— executed before and after the execution of all test methods respectively
- `BeforeAllCallback` and `AfterAllCallback`— executed before and after the execution of all test methods respectively

Harry implements the `DatabaseOperationsExtension` shown in listing 14.9.

Listing 14.9 The DatabaseOperationsExtension class

```
public class DatabaseOperationsExtension implements #A
    BeforeAllCallback, AfterAllCallback, BeforeEachCallback, #A
    AfterEachCallback { #A

    private Connection connection; #B
    private Savepoint savepoint; #B

    @Override
    public void beforeAll(ExtensionContext context) {
        connection = ConnectionManager.openConnection(); #C
        TablesManager.dropTable(connection); #C
        TablesManager.createTable(connection); #C
    }

    @Override
    public void afterAll(ExtensionContext context) {
        ConnectionManager.closeConnection(); #D
    }

    @Override
    public void beforeEach(ExtensionContext context)
        throws SQLException {
        connection.setAutoCommit(false); #E
        savepoint = connection.setSavepoint("savepoint"); #F
    }
}
```

```

    }

    @Override
    public void afterEach(ExtensionContext context)
        throws SQLException {
        connection.rollback(savepoint); #G
    }

}

```

In this listing:

- The `DatabaseOperationsExtension` class implements four life-cycle interfaces: `BeforeAllCallback`, `AfterAllCallback`, `BeforeEachCallback`, and `AfterEachCallback` (#A).
- The class declares a `Connection` field to connect to the database and a `Savepoint` field to track the state of the database before the execution of a test and to restore it after the test (#B).
- The `beforeAll` method, inherited from the `BeforeAllCallback` interface, is executed before the whole suite. It opens a connection to the database, drops the existing table, and re-creates it (#C).
- The `afterAll` method, inherited from the `AfterAllCallback` interface, is executed after the whole suite. It closes the connection to the database (#D).
- The `beforeEach` method, inherited from the `BeforeEachCallback` interface, is executed before each test. It disables auto-commit mode, so the database changes resulting from the execution of the test should not be committed (#E). Then the method saves the state of the database before the execution of the test so that after the test, the developer can roll back to it (#F).
- The `afterEach` method, inherited from the `AfterEachCallback` interface, is executed after each test. It rolls back to the state of the database that was saved before the execution of the test (#G).

Harry updates the `PassengerTest` class and introduces tests that verify the newly introduced database functionalities (listing 14.10).

Listing 14.10 The updated PassengerTest class

```

@ExtendWith({ExecutionContextExtension.class,
            DatabaseOperationsExtension.class })
public class PassengerTest {

    private PassengerDao passengerDao;

    public PassengerTest(PassengerDao passengerDao) { #B
        this.passengerDao = passengerDao;
    }

    @Test
    void testPassenger(){
        Passenger passenger = new Passenger("123-456-789", "John Smith");
    }
}

```

```

        assertEquals("Passenger John Smith with identifier: 123-456-789",
                     passenger.toString());
    }

    @Test
    void testInsertPassenger() {
        Passenger passenger = new Passenger("123-456-789",
                                             "John Smith");                      #C
        passengerDao.insert(passenger);           #D
        assertEquals("John Smith",                #E
                     passengerDao.getById("123-456-789").getName());      #E
    }

    @Test
    void testUpdatePassenger() {
        Passenger passenger = new Passenger("123-456-789",
                                             "John Smith");                      #F
        passengerDao.insert(passenger);           #G
        passengerDao.update("123-456-789", "Michael Smith");          #H
        assertEquals("Michael Smith",             #I
                     passengerDao.getById("123-456-789").getName());      #I
    }

    @Test
    void testDeletePassenger() {
        Passenger passenger = new Passenger("123-456-789",
                                             "John Smith");                      #J
        passengerDao.insert(passenger);           #K
        passengerDao.delete(passenger);          #L
        assertNull(passengerDao.getById("123-456-789"));                 #M
    }
}

```

In this listing:

- The test is extended by the `DatabaseOperationsExtension (#A)`.
- The constructor of the `PassengerTest` class receives a `PassengerDao` as argument (`#B`). This `PassengerDao` will be needed for executing the tests against the database.
- In the `testInsertPassenger` method, we create a passenger (`#C`), insert it into the database with `passengerDao` (`#D`), and check whether it is found in the database (`#E`).
- In the `testUpdatePassenger` method, we create a passenger (`#F`), inserting it into the database with `passengerDao` (`#G`), update the information about it (`#H`), and check whether the update information is found in the database (`#I`).
- In the `testDeletePassenger` method, we create a passenger (`#J`), insert it into the database with `passengerDao` (`#K`), delete it (`#L`), and check whether the passenger is found in the database (`#M`).

If we run the tests at this time, we get the results shown in figure 14.3.

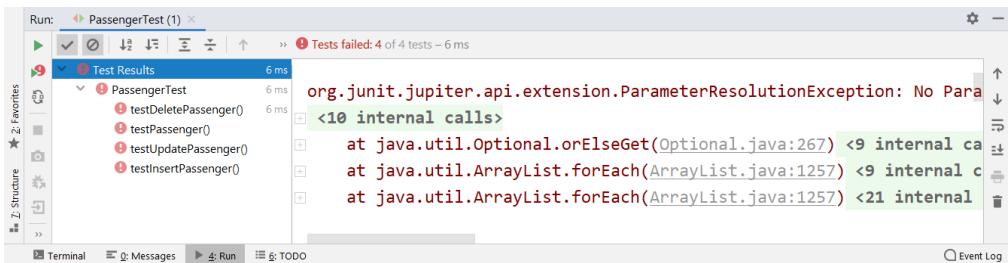


Figure 14.3 The result of the updated `PassengerTest` after it is extended with `DatabaseOperationsExtension`

The tests fail with this message:

```
org.junit.jupiter.api.extension.ParameterResolutionException:  
No ParameterResolver registered for parameter [com.manning.junitbook.ch14.jdbc.PassengerDao  
arg0] in constructor
```

This message is due to the fact that the constructor of the `PassengerTest` class is receiving a parameter of type `PassengerDao`, whereas this parameter is not provided by any `ParameterResolver`. To complete the task, Harry has to implement a parameter resolution extension, as shown in listing 14.11.

Listing 14.11 The `DatabaseAccessObjectParameterResolver` class

```
public class DatabaseAccessObjectParameterResolver implements #A  
    ParameterResolver{ #A  
  
    @Override  
    public boolean supportsParameter(ParameterContext parameterContext,  
                                    ExtensionContext extensionContext)  
        throws ParameterResolutionException {  
            return parameterContext.getParameter()  
                .getType()  
                .equals(PassengerDao.class); #B  
        }  
  
    @Override  
    public Object resolveParameter(ParameterContext parameterContext,  
                                 ExtensionContext extensionContext)  
        throws ParameterResolutionException {  
            return new PassengerDaoImpl(ConnectionManager.getConnection()); #C  
        }  
}
```

In this listing:

- The class implements the `ParameterResolver` interface (#A).
- The `supportsParameter` method returns true if the parameter is of type `PassengerDao`, which is the missing parameter of the constructor of the

PassengerTest class (#B), so the parameter resolver supports only a PassengerDao object.

- The resolveParameter method returns a newly initialized PassengerDaoImpl that receives as a parameter of the constructor the connection provided by the ConnectionManager (#C). This parameter will be injected into the test constructor at run time.

Additionally, Harry extends the PassengerTest class with the DatabaseAccessObjectParameterResolver. The first lines of the PassengerTest class look like listing 14.12.

Listing 14.12 The updated PassengerTest class, extended with DatabaseAccessObjectParameterResolver

```
@ExtendWith({ExecutionContextExtension.class, DatabaseOperationsExtension.class,
             DatabaseAccessObjectParameterResolver.class})
public class PassengerTest {
    [...]
```

The result of the execution of PassengerTest is shown in figure 14.4. All tests are green; the interaction with the database works properly.



Figure 14.4 The result of the updated PassengerTest after it is extended with DatabaseOperationsExtension and DatabaseAccessObjectParameterResolver

Harry successfully implemented the additional behavior of the tests executed against a database with the help of the life-cycle callback and parameter resolution extensions.

14.3.2 Checking the unicity of the passengers

Next, Harry must prevent a passenger from being inserted into the database more than once. To implement this requirement, he decides to create his own custom exception, together with an exception handling extension. He wants to introduce this custom exception because it is more expressive than the general SQLException.

First, Harry creates the custom exception, as shown in listing 14.13.

Listing 14.13 The PassengerExistException class

```
public class PassengerExistsException extends Exception {
```

```

private Passenger passenger; #A

public PassengerExistsException(Passenger passenger, String message) {
    super(message); #B
    this.passenger = passenger; #C
}
}

```

In this listing:

- We keep the existing passenger as a field of the exception (#A).
- We invoke the constructor of the superclass, retaining the message parameter (#B), then we set the passenger (#C).

Next, Harry changes the `PassengerDao` interface and the `PassengerDaoImpl` class so that the insert method

```
public void insert(Passenger passenger) throws PassengerExistsException;
```

throws `PassengerExistsException`, as shown in listing 14.14.

Listing 14.14 The modified insert method from `PassengerDaoImpl`

```

public void insert(Passenger passenger) throws PassengerExistsException {
    String sql = "INSERT INTO PASSENGERS (ID, NAME) VALUES (?, ?)";

    if (null != getById(passenger.getIdentifier()) ) { #A
        throw new PassengerExistsException #A
            (passenger, passenger.toString()); #A
    } #A

    try (PreparedStatement statement = connection.prepareStatement(sql)){ #A
        statement.setString(1, passenger.getIdentifier());
        statement.setString(2, passenger.getName());
        statement.executeUpdate();
    } catch (SQLException e) { #A
        throw new RuntimeException(e);
    }
}

```

In the `insert` method from `PassengerDaoImpl`, we check the existence of the passenger. If it exists, we throw `PassengerExistsException` (#A).

Harry wants to introduce a test that tries to insert the same passenger twice. As he expects this test to throw an exception, he implements an exception-handling extension to log this one, as shown in listing 14.15.

Listing 14.15 The `LogPassengerExistsExceptionExtension` class

```

public class LogPassengerExistsExceptionExtension implements #A
    TestExecutionExceptionHandler { #A
        private Logger logger = Logger.getLogger(this.getClass().getName()); #B

        @Override
        public void handleTestExecutionException(ExtensionContext context, #C
            Throwable throwable) throws Throwable { #C
}

```

```

        if (throwable instanceof PassengerExistsException) {          #D
            logger.severe("Passenger exists:" + throwable.getMessage()); #D
            return;           #D
        }
        throw throwable;           #E
    }
}

```

In this listing:

- The class implements the `TestExecutionExceptionHandler` interface (#A).
- We declare a logger of the class (#B) and override the `handleTestExecutionException` method inherited from the `TestExecutionExceptionHandler` interface (#C).
- We check whether the thrown exception is an instance of `PassengerExistsException`, in which case we simply log it and return from the method (#D); otherwise, we re-throw the exception so it can be handled elsewhere (#E).

The updated `PassengerTest` class is shown in listing 14.16.

Listing 14.16 The updated PassengerTest class

```

@ExtendWith({ExecutionContextExtension.class, DatabaseOperationsExtension.class,
            DatabaseAccessObjectParameterResolver.class,
            LogPassengerExistsExceptionExtension.class})
public class PassengerTest {
    [...]
}

```

The result of `PassengerTest` is shown in figure 14.5. All tests are green; the `PassengerExistsException` is caught and logged by the new extension.

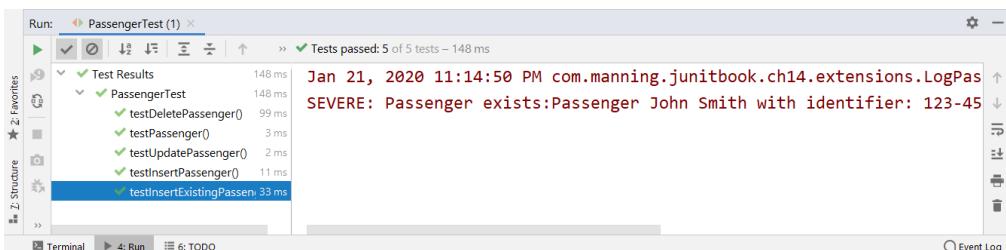


Figure 14.5 The result of the updated `PassengerTest` after it is extended with `LogPassengerExistsExceptionExtension`

Table 14.2 summarizes the extension points and interfaces to be implemented.

Table 14.2 Extension points and corresponding interfaces

Extension point	Interfaces to implement
Conditional test execution	ExecutionCondition
Life-cycle callbacks	BeforeAllCallback, AfterAllCallback, BeforeEachCallback, AfterEachCallback
Parameter resolution	ParameterResolver
Exception handling	TestExecutionExceptionHandler
Test instance postprocessing	TestInstancePostProcessor

Chapter 15 is dedicated to presentation layer testing and finding bugs in the graphical user interface (GUI) of an application.

14.4 Summary

This chapter has covered the following:

- Introducing the JUnit 5 extension model and the available extension points: conditional test execution, life-cycle callbacks, parameter resolution, exception handling, test instance post-processing.
- Demonstrating the development tasks that required the creation of JUnit 5 extensions: a context extension for conditional test execution; a passenger database setup extension for life-cycle callbacks; a parameter resolver for parameter resolution; a logging exception extension for exception handling.
- Implementing JUnit 5 tests required by the development scenario using the extensions enumerated above: the insertion, update and deletion of passengers into a database; the unicity of the passengers that are inserted into the database.

15

Presentation Layer Testing

This chapter covers:

- Introducing presentation layer testing
- Operating with HtmlUnit
- Developing HtmlUnit tests
- Operating with Selenium
- Developing Selenium tests
- Comparing HtmlUnit and Selenium

If debugging is the process of removing software bugs, then programming must be the process of putting them in.

- Edsger Dijkstra

Simply stated, presentation layer testing is finding bugs in the graphical user interface (GUI) of an application. Finding errors here is as important as finding errors in other application tiers. Bad user experience can lose a customer or discourage a web surfer from visiting your site again. Furthermore, bugs in the user-interface may cause other parts of the application to malfunction.

Due to its nature and interaction with a person, GUI testing presents unique challenges and requires its own set of tools and techniques. This chapter will cover the testing of web application user interfaces.

We address here what can be objective, that is programmatically (by writing explicit Java code), asserted about the GUI. Outside the scope of this discussion are whether the choice of subjective elements like fonts, colors and layout cause an application to be pleasant, difficult or impossible to use.

Testing the interaction with websites may be challenging from the point of view of stability. If the content of the websites will change in time, if you encounter problems with your Internet connection, this kind of test may occasionally or permanently fail. The tests presented in this chapter have been designed to have high stability for a long period of time, by accessing known websites with less probability of change in the near future.

What we can test is:

- The content of web pages to any level of detail (we could include spelling)
- The application structure or navigation (following links to their expected destination for example)
- The ability to verify user-stories with acceptance tests¹.

A user story is an informal, natural language description of one or more features of a software system. An acceptance test is a test conducted to determine if the requirements of a specification are met. We can also verify that the site works with the required browsers and operating systems.

15.1 Choosing a Testing Framework

We will look at two free open source tools to implement presentation layer tests within JUnit 5: HtmlUnit and Selenium.

HtmlUnit is a 100% Java headless browser framework that runs in the same virtual machine as your JUnit 5 tests. A headless browser is a browser without a graphical user interface. We use HtmlUnit when our application is independent of the operating system features and browser-specific implementations of JavaScript, DOM, CSS, etc.

Selenium drives various web browsers programmatically and checks the results from executing the JUnit 5 tests. We use Selenium when we require validation of specific browsers and operating systems, especially if the application takes advantage of or depends on a browser-specific implementation of JavaScript, DOM, CSS, etc. Let us start with HtmlUnit.

15.2 Introducing HtmlUnit

HtmlUnit is an open-source Java headless browser framework. It allows tests to imitate programmatically the user of a browser-based web application. HtmlUnit JUnit 5 tests do not display a user interface. In the remainder of this HtmlUnit section, when we talk about "testing with a web browser", it is with the understanding that we are really "testing by emulating a specific web browser".

¹ Extreme programming acceptance tests <http://www.extremeprogramming.org/rules/functionaltests.html>

15.2.1 A live example

We'll first introduce the ManagedWebClient base class that we are going to use as the base class for the HtmlUnit tests using the JUnit 5 annotations.

Listing 15.1 The ManagedWebClient base class

```
import com.gargoylesoftware.htmlunit.WebClient;
[...]

public abstract class ManagedWebClient {
    protected WebClient webClient; #A

    @BeforeEach
    public void setUp() {
        webClient = new WebClient();
    } #B

    @AfterEach
    public void tearDown() {
        webClient.close();
    } #C
}
```

In this listing, we are creating the abstract ManagerWebClient class as the basis for the HtmlUnit test classes and we are doing the following:

- We are defining a protected WebClient field to be inherited by the subclasses (#A). The com.gargoylesoftware.htmlunit.WebClient class is the main starting point of an HtmlUnit test.
- Before each test, we are initializing a new WebClient object that will be used for the execution of the test (#B). It simulates a web browser and will be used to execute all the tests.
- After each test, we make sure that the simulated browser is closed when the test is executed with a WebClient instance (#C).

Let us jump in with the example in listing 15.2 if you can connect to the Internet, you can test. We will go to the HtmlUnit web site and test the home page. We will also navigate to the Javadoc, and make sure a class appears on top of the list of documented classes.

Listing 15.2 Our first HtmlUnit example

```
public class HtmlUnitPageTest extends ManagedWebClient {

    @Test
    public void homepage() throws IOException {
        HtmlPage page = webClient.getPage("http://htmlunit.sourceforge.net"); #A
        assertEquals("HtmlUnit - Welcome to HtmlUnit", page.getTitleText()); #A

        String pageAsXml = page.asXml(); #B
        assertTrue(pageAsXml.contains("<div class=\"container-fluid\">")); #B

        String pageAsText = page.asText(); #C
    }
}
```

```

        assertTrue(pageAsText.contains(                      #C
            "Support for the HTTP and HTTPS protocols"));    #C
    }

    @Test
    public void testClassNav() throws IOException {          #D
        HtmlPage mainPage = webClient.getPage(               #D
            "http://htmlunit.sourceforge.net/apidocs/index.html");   #D
        HtmlPage packagePage = (HtmlPage)                      #E
            mainPage.getFrameByName("packageFrame").getEnclosedPage(); #E
        HtmlListItem htmlListItem = (HtmlListItem)           #F
            packagePage.getElementsByTagName("li").item(0);      #F
        assertEquals("AboutURLConnection", htmlListItem.getTextContent()); #G
    }
}

```

Let us work through the example:

- In the first test, we start by accessing the HtmlUnit web-site home page and we are checking that the title of the page is the expected one (#A).
- We are getting the home page of the HtmlUnit website as XML and we are checking that it contains a particular tag (#B).
- We are getting the home page of the HtmlUnit web-site as text and we are checking that it contains a particular string (#C).
- Into the second test, we are accessing a URL from the HtmlUnit web-site (#D).
- We are getting the page from inside the packageFrame frame (#E).
- Inside the page obtained at (#E), we are looking for the first element tagged as "li" (#F).
- We are checking that the text of the element obtained at (#F) is "AboutURLConnection" (#G).

This example covers the basics: getting a web page, navigating the HTML object model, and asserting results.

15.3 Writing HtmlUnit tests

When we write an HtmlUnit test, we write code that simulates the action of a user sitting in front of a web browser: we get a web page, enter data, read the text, and click on buttons and links. Instead of manually manipulating the browser, we programmatically control an emulated browser. At each step, we can query the HTML object model and assert that values are what we expect. The framework will throw exceptions if it encounters a problem, which allows our test cases to avoid checking for these errors, reducing clutter.

15.3.1 HTML Assertions

We know that JUnit 5 provides a class called `Assertions` to allow tests to fail when they detect an error condition. `Assertions` are the bread and butter of any unit test.

HtmlUnit may work with JUnit 5, but it also provides a class in the same spirit called WebAssert, which contains standard assertions for HTML like `assertTitleEquals`, `assertTextPresent` or `notNull`.

15.3.2 Testing for a specific web browser

HtmlUnit, as of version 2.36, supports the following browsers:

Table 15.1 HTMLUnit supported browsers.

Web Browser and Version	HtmlUnit BrowserVersion Constant
Internet Explorer 11	BrowserVersion.INTERNET_EXPLORER
Firefox 5.2 (deprecated)	BrowserVersion.FIREFOX_52
Firefox 6.0	BrowserVersion.FIREFOX_60
Latest Chrome	BrowserVersion.CHROME
The best-supported browser at the moment	BrowserVersion.BEST_SUPPORTED

By default, WebClient emulates `BrowserVersion.BEST_SUPPORTED` which, at the time of writing this chapter, is Google Chrome, but may change in the future, depending on the evolution of each particular browser. In order to specify which browser to emulate, you provide the WebClient constructor with a `BrowserVersion`. For example, for Firefox 6.0, use:

```
WebClient webClient = new WebClient(BrowserVersion.FIREFOX_60);
```

15.3.3 Testing more than one web browser

Tested Data Systems is an outsourcing company having many customers. Naturally, each customer will use his preferred browser for accessing the pages of the application that has been developed for him. Consequently, the engineers at Tested Data Systems would like to test their applications with more than one browser version. John is in charge of writing tests for this purpose. He will define a test matrix to include all HtmlUnit supported web browsers.

Listing 15.3 uses the JUnit 5 parameterized tests to drive the same test with all browsers in the test matrix. The JUnit 5 parameterized tests will use Firefox 6.0, Internet Explorer 11, latest Chrome (79 at the moment of writing this chapter) and the best-supported browser (again Chrome 79 at the moment of writing this chapter, but may change in the future).

Listing 15.3 Testing for all HtmlUnit supported browsers

```
public class JavadocPageAllBrowserTest {  
  
    private static Collection<BrowserVersion[]> getBrowserVersions() { #A  
        return Arrays.asList(new BrowserVersion[][] { #A  
            { BrowserVersion.FIREFOX_60 }, #A  
            { BrowserVersion.INTERNET_EXPLORER }, #A  
            { BrowserVersion.CHROME }, #A
```

```

        { BrowserVersion.BEST_SUPPORTED } });
#A

}

@ParameterizedTest
@MethodSource("getBrowserVersions")
public void testClassNav(BrowserVersion browserVersion)
    throws IOException
{
    WebClient webClient = new WebClient(browserVersion);
#E

    HtmlPage mainPage = (HtmlPage)webClient
        .getPage(
            "http://htmlunit.sourceforge.net/apidocs/index.html");
    WebAssert.notNull("Missing main page", mainPage);
#G

    HtmlPage packagePage = (HtmlPage) mainPage
        .getFrameByName("packageFrame").getEnclosedPage();
    WebAssert.notNull("Missing package page", packagePage);
#I

    HtmlListItem htmlListItem = (HtmlListItem) packagePage
        .getElementsByTagName("li").item(0);
    assertEquals("AboutURLConnection", htmlListItem.getTextContent());
#J
#K
}
}

```

In listing 15.3 we are doing the following:

- We are creating a `private static` method that will provide the collection of all web browsers supported by `HtmlUnit` (#A).
- We are creating the `testClassNav` method that will receive as parameter the `BrowserVersion` (#D). This is a parameterized test as indicated by the `@ParameterizedTest` annotation (#B). The `BrowserVersion` parameters will be injected by the `getBrowserVersion` method, as indicated by the `@MethodSource` annotation (#C).
- We are creating a `WebClient` receiving as constructor argument the injected `BrowserVersion` (#E) and we are accessing a page on the Internet (#F). We are checking that the page we try to access exists (#G).
- Inside the page that we have previously accessed, we are looking for a frame with the name `packageFrame` (#H) and we are checking that the frame exists (#I).
- Inside the frame with the name `packageFrame` we are looking for a list of items – it is the list of all `HtmlUnit` classes. We are checking if the first class on the list is named `AboutURLConnection` (#J).

15.3.4 Creating stand-alone tests

A developer may not always want to use the actual URLs of the pages as input for the tests. The reason is that external pages may be subject to change without notice. A minor change in the website can make the tests break. Next, we will show you how to embed and run HTML in the unit test code itself.

The framework allows you to plug in a mock² HTTP connection into a web client. In listing 15.4, we create a JUnit 5 test that sets up a mock connection (an instance of the class com.gargoylesoftware.htmlunit.MockWebConnection) with a default HTML String response. The JUnit 5 test can then get this default page by using any URL value. We will test the title of the HTML response obtained by a web client with a mocked connection, that avoids using an actual URL that may change without notice.

Listing 15.4 Configuring a stand-alone test

```
public class InLineHtmlFixtureTest extends ManagedWebClient {  
  
    @Test  
    public void testInLineHtmlFixture() throws IOException {  
        final String expectedTitle = "Hello 1!";  
        String html = "<html><head><title>" +  
            expectedTitle +  
            "</title></head></html>";  
        MockWebConnection connection = new MockWebConnection();  
        connection.setDefaultResponse(html);  
        webClient.setWebConnection(connection);  
        HtmlPage page = webClient.getPage("http://page");  
        WebAssert.assertTitleEquals(page, expectedTitle);  
    }  
}
```

In listing 15.4 we are doing the following:

- We create the expected HTML page title (#A) and the expected HTML response (#B).
- Then we create a MockWebConnection (#C) and set the HTML response as the default response for the mock connection (#D). We then set the web client connection to our mock connection (#E).
- We are now ready to go, and we get the test page (#F). Any URL will be fine here since we set up our HTML response as the default response. We finally check that the page title matches our HTML response (#G).

To configure a JUnit 5 test with multiple pages, we call one of the `MockWebConnection.setResponse` methods for each page. The code in listing 15.5 sets up three web pages in a mock connection. We will test the titles of the HTML responses obtained by a web client with a mocked connection, that avoids using an actual URL that may change without notice.

Listing 15.5 Configuring a test with multiple page fixtures

```
@Test  
public void testInLineHtmlFixtures() throws IOException {
```

² See Chapter 8: Testing with mock objects

```

final URL page1Url = new URL("http://Page1/"); #A
final URL page2Url = new URL("http://Page2/"); #A
final URL page3Url = new URL("http://Page3/"); #A

MockWebConnection connection = new MockWebConnection(); #B
connection.setResponse(page1Url,
    "<html><head><title>Hello 1!</title></head></html>");
connection.setResponse(page2Url,
    "<html><head><title>Hello 2!</title></head></html>"); #C
connection.setResponse(page3Url,
    "<html><head><title>Hello 3!</title></head></html>"); #C
webClient.setWebConnection(connection); #D

HtmlPage page1 = webClient.getPage(page1Url);
WebAssert.assertTitleEquals(page1, "Hello 1!"); #E
#E

HtmlPage page2 = webClient.getPage(page2Url);
WebAssert.assertTitleEquals(page1, "Hello 2!"); #E
#E

HtmlPage page3 = webClient.getPage(page3Url);
WebAssert.assertTitleEquals(page1, "Hello 3!"); #E
#E
}

```

In listing 15.5 we are doing the following:

- We create three pages (#A), a mock connection (#B) and set three responses for each of the URL pages (#C).
- We set the web client connection to our mock connection (#D).
- We finally test getting each page and verifying each page title (#E).

Common Pitfall

Do not forget the trailing / in the URL; “`http://Page1/`” will work but “`http://Page1`” will not be found in the mock connection and therefore throw an `IllegalStateException` with the message “No response specified that can handle URL”.

15.3.5 Testing forms

HTML form support is built into the `HtmlPage` API where form elements can be accessed with `getForms` (returns `List<HtmlForm>`) to get all form elements and `getFormByName` to get the first `HtmlForm` with a given name. You can call one of the `HtmlForm` `getInput` methods to get HTML input elements, and then simulate user input with `setValueAttribute`.

The following example will focus on the `HtmlUnit` mechanics of driving a form. First, we create a simple page to display a form with an input field and submit button. We include form validation via JavaScript alerts in listing 15.6.

Listing 15.6 Example form page

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
```

```

<script>
function validate_form(form) {
    if (form.in_text.value=="") {
        alert("Please enter a value.");
        form.in_text.focus();
        return false;
    }
}
</script>
<title>Form</title></head>
<body>
<form name="validated_form" action="submit.html" onsubmit="return validate_form(this);"
      method="post">
    Value:
    <input type="text" name="in_text" id="in_text" size="30"/>
    <input type="submit" value="Submit" id="submit"/>
</form>
</body>
</html>

```

This form looks like the following when you click the button without input.

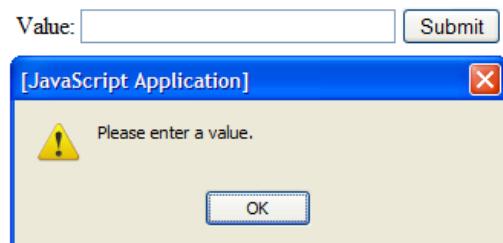


Figure 15.1 Sample form with one input and one submit button and the alert triggered when pressing the button

We test normal user interaction with the form in listing 15.7.

Listing 15.7 Testing a form

```

public class FormTest extends ManagedWebClient {

    @Test
    public void testForm() throws IOException {
        HtmlPage page =
            webClient.getPage("file:src/main/webapp/formtest.html");          #A
        HtmlForm form = page.getFormByName("validated_form");                  #B
        HtmlTextInput input = form.getInputByName("in_text");                  #C
        input.setValueAttribute("typing...");                                    #D
        HtmlSubmitInput submitButton = form.getInputByName("submit");         #E
        HtmlPage resultPage = submitButton.click();                           #E
        WebAssert.assertTitleEquals(resultPage, "Result");                   #F
    }
}

```

In listing 15.7 we are doing the following:

- We use the web client inherited from the parent `ManagedWebClient` class to get the page containing the form (#A), and then we get the form (#B).
- We get the input text field from the form (#C), emulate the user typing in a value (#D), then get and click the submit button (#E).
- We get a page back from clicking the button and we make sure it is the expected page (#F).

If at any step, the framework does not find an object, the API throws an exception and the test automatically fails. This allows you to focus on the test and let the framework handle failing your test if the page or form is not as expected. The section Testing JavaScript Alerts will complete this example.

15.3.6 Testing JavaScript

`HtmlUnit` processes JavaScript automatically. Even when, for example, HTML is generated with `Document.write()`, you follow the usual pattern: call `getPage`, find an element, click on it, and check the result.

You can toggle JavaScript support on and off in a web client by calling:

```
webClient.getOptions().setJavaScriptEnabled(true);
```

or

```
webClient.getOptions().setJavaScriptEnabled(false);
```

`HtmlUnit` enables JavaScript support by default. You can also set how long a script is allowed to run before being terminated by calling:

```
webClient.setJavaScriptTimeout(timeout);
```

To deal with JavaScript alert and confirm calls, you can provide the framework with callbacks routines. We will explore these ones next.

TESTING JAVASCRIPT ALERTS

Our tests can check which JavaScript alerts have taken place. We will re-use our example for testing forms in listing 15.7, which includes JavaScript validation code to alert the user of empty input values.

The test in listing 15.8 loads our form page and checks calling the alert when the form detects an error condition. In another step, we will enhance our existing test from listing 15.7 to ensure that the normal operation of the form does not raise any alerts. Our test will install an alert handler that gathers all alerts and will check the result after the page has been loaded. The class `CollectingAlertHandler` saves alert messages for later inspection.

Listing 15.8 Asserting expected alerts

```
public class FormTest extends ManagedWebClient {  
[...]
```

```

@Test
public void testFormAlert() throws IOException {
    CollectingAlertHandler alertHandler =           #A
        new CollectingAlertHandler();
    webClient.setAlertHandler(alertHandler);          #B
    HtmlPage page = webClient.getPage(
        "file:src/main/webapp/formtest.html");       #C
    HtmlForm form = page.getFormByName("validated_form"); #D
    HtmlSubmitInput submitButton = form.getInputByName("submit"); #E
    HtmlPage resultPage = submitButton.click();         #F
    WebAssert.assertTitleEquals(resultPage, page.getTitleText()); #G
    WebAssert.assertTextPresent(resultPage, page.asText()); #H

    List<String> collectedAlerts =           #I
        alertHandler.getCollectedAlerts();           #I
    List<String> expectedAlerts = Collections.singletonList(      #J
        "Please enter a value."); #J
    assertEquals(expectedAlerts, collectedAlerts); #K
}
}

```

Into listing 15.8 we do the following:

- We start by creating the alert handler (#A), which we install in the web client inherited from the parent class (#B).
- We get the form page (#C), get the form object (#D), get the submit button (#E) and click on it (#F). This invokes the JavaScript, which calls alert.
- Clicking the button returns a page object, which we use to check that the page has not changed by comparing current and previous page titles (#G). We also check that the page has not changed by comparing current and previous page objects (#H).
- Finally, we get the list of alert messages that were raised (#I), create a list of expected alert messages (#J), and compare the expected and actual lists (#K).

Next, listing 15.9 rewrites the original form test from listing 15.9 to make sure that normal operation raises no alerts.

Listing 15.9 Asserting no alerts under normal operation

```

public class FormTest extends ManagedWebClient {
[...]

    @Test
    public void testFormNoAlert() throws IOException {
        CollectingAlertHandler alertHandler = new CollectingAlertHandler(); #A
        webClient.setAlertHandler(alertHandler);                         #A
        HtmlPage page = webClient.getPage(
            "file:src/main/webapp/formtest.html");
        HtmlForm form = page.getFormByName("validated_form");
        HtmlTextInput input = form.getInputByName("in_text");
        input.setValueAttribute("typing...");                                #B
        HtmlSubmitInput submitButton = form.getInputByName("submit");
        HtmlPage resultPage = submitButton.click();
        WebAssert.assertTitleEquals(resultPage, "Result");
        assertTrue(alertHandler.getCollectedAlerts().isEmpty(), #C

```

```

        "No alerts expected");
    }
}

```

Compared with the test from listing 15.8, we do the following:

- We install a `CollectingAlertHandler` in the web client (#A).
- We simulate a user entering a value (#B) and, at the end of the test, we check that the alert handler list of messages is empty (#C).

To customize the alert behavior, we need to implement our own `AlertHandler`. Setting the `AlertHandler` as shown in listing 15.10 will cause our test to fail when a script raises the first alert.

Listing 15.10 Custom alert handler

```
webClient.setAlertHandler((page, message) ->
    fail("JavaScript alert: " + message));
```

In listing 15.11, we apply the same principles to test JavaScript confirm calls by installing a confirm handler in the web client with `setConfirmHandler`.

Listing 15.11 Asserting the expected confirmation messages

```
public class WindowConfirmTest extends ManagedWebClient {

    @Test
    public void testWindowConfirm() throws FailingHttpStatusCodeException,
        IOException {
        String html = "<html><head><title>Hello</title></head>
                     <body onload='confirm(\"Confirm Message\")'>
                     </body></html>";
        URL testUrl = new URL("http://Page1/");
        MockWebConnection mockConnection = new MockWebConnection();
        final List<String> confirmMessages = new ArrayList<String>();

        webClient.setConfirmHandler((page, message) -> {
            confirmMessages.add(message);
            return true;
        });
        mockConnection.setResponse(testUrl, html);
        webClient.setWebConnection(mockConnection);

        HtmlPage firstPage = webClient.getPage(testUrl);
        WebAssert.assertTitleEquals(firstPage, "Hello");
        assertEquals(new String[] { "Confirm Message" },
                    confirmMessages.toArray());
    }
}
```

In listing 15.11 we do the following:

- We start by creating the HTML page including the confirm message (#A) and the test URL to be accessed (#B).

- We create a mock connection (#C) and we initialize an empty confirm messages list (#D).
- We setup the `webClient` inherited from the super-class with a confirm handler defined as a lambda expression that will simply add the message to the list (#E). We set the response of the connection when accessing the test URL (#F) and then set the web client connection to our mock connection (#G).
- We get the page (#H), then check its title (#I) and the array of confirming messages (#J).

In listing 15.12 we are doing making some variation of the previous program, which will introduce a function into the JavaScript code of the emulated web-site and a collecting alert handler.

Listing 15.12 Asserting the expected confirmation messages from a JavaScript function

```
public class WindowConfirmTest extends ManagedWebClient {

    @Test
    public void testWindowConfirmAndAlert() throws
        FailingHttpStatusCodeException, IOException {
        String html = "<html><head><title>Hello</title>
                      <script>function go(){
                        alert(confirm('Confirm Message'))
                      }</script>\n" +
                      "</head><body onload='go()'></body></html>"; #A
        URL testUrl = new URL("http://Page1/");
        MockWebConnection mockConnection = new MockWebConnection();
        final List<String> confirmMessages = new ArrayList<String>(); #B
        webClient.setAlertHandler(new CollectingAlertHandler());
        webClient.setConfirmHandler((page, message) -> {
            confirmMessages.add(message);
            return true;
        });
        mockConnection.setResponse(testUrl, html);
        webClient.setWebConnection(mockConnection);

        HtmlPage firstPage = webClient.getPage(testUrl);
        WebAssert.assertTitleEquals(firstPage, "Hello");
        assertArrayEquals(new String[] { "Confirm Message" }, #C
                         confirmMessages.toArray());
        assertArrayEquals(new String[] { "true" }, #D
                         ((CollectingAlertHandler)
                          webClient.getAlertHandler())
                          .getCollectedAlerts().toArray()); #D
    }
}
```

Compared to listing 15.11, in listing 15.12 we do the following:

- We are creating the HTML page including the confirm message provided by a JavaScript function (#A).
- We are also setting an alert handler for the `webClient` inherited from the super-class (#B). A `CollectingAlertHandler` is a simple alert handler that keeps track of

alerts in a list.

- Besides the list of confirmation messages (#C), we are also checking the collected alerts (#D).

15.4 Introducing Selenium

Selenium³ is a free open-source tool suite used to test web applications. Selenium's strength lies in its ability to run tests against a real browser on a specific operating system, this is unlike HtmlUnit, which emulates the browser in the same VM (Virtual Machine) as your tests. Selenium lets you write tests in a few programming languages, including Java with JUnit 5.

Selenium WebDriver is the name of the key interface against which the test should be written. WebDriver is implemented by many current browsers, including Chrome, Firefox, Internet Explorer.

Selenium WebDriver makes direct calls to the browser using each browser native support for automation. How these direct calls are made, and the features they support depends on the browser you are using. Each and every browser has different logic for performing actions on the browser. Figure 15.2 shows the various components of Selenium WebDriver Architecture.

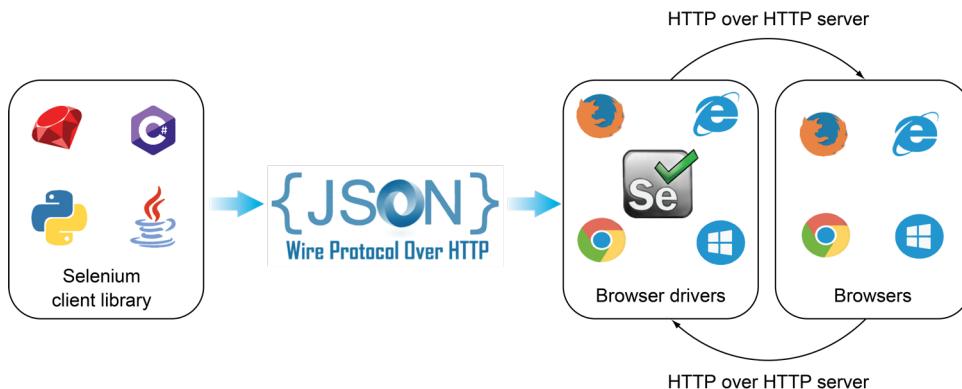


Figure 15.2 The Selenium WebDriver architecture, including the Selenium Client Library, the browser drivers, the browsers themselves and the HTTP communication

Selenium WebDriver includes the following four components:

³ Selenium site: <http://seleniumhq.org/>

1. Selenium Client Libraries - Selenium supports multiple libraries for programming languages as Java, C#, PHP, Python, Ruby, etc.
2. JSON Wire Protocol Over HTTP - JSON (JavaScript Object Notation) is used to transfer data between servers and clients on the web. JSON Wire Protocol is a REST API that transfers the information to the HTTP server. Each WebDriver (such as FirefoxDriver, ChromeDriver, InternetExplorerDriver, etc.) has its own HTTP server.
3. Browser Drivers - Each browser contains a separate browser driver. Browser drivers communicate with the respective browser without revealing the internal logic of the browser functionality. When a browser driver has received any command, then that command will be executed on the respective browser and the response will go back in the form of an HTTP response.
4. Browsers - Selenium supports multiple browsers such as Firefox, Chrome, Internet Explorer, Safari, etc.

15.5 Writing Selenium Tests

We can now explore writing individual tests with Selenium. We will look at how to test for more than one browser, how to navigate the object model and work through some example JUnit 5 tests.

The first thing to do in order to setup a Selenium Java project is to include the Maven dependency into the project, as shown in listing 15.13.

Listing 15.13 The Selenium dependency into the pom.xml configuration

```
<dependency>
    <groupId>org.seleniumhq.selenium</groupId>
    <artifactId>selenium-java</artifactId>
    <version>3.141.59</version>
</dependency>
```

Selenium, as of version 3, supports the following browsers:

Table 15.2 Browsers supported by Selenium

Web Browser	Browser driver class
Google Chrome	ChromeDriver
Internet Explorer	InternetExplorerDriver
Safari	SafariDriver
Opera	OperaDriver
Firefox	FirefoxDriver
Edge	EdgeDriver

In order to be able to test for a specific browser, we will need to download the Selenium drivers for that specific browser and to include the access path to it on the operating system path. The links to download each driver can be found by visiting <https://www.seleniumhq.org/download/>, navigating to the "Browsers" section and choosing the browser you are interested about.

For our demonstration purposes, we are going to use three of the most popular browsers: Google Chrome, Internet Explorer and Mozilla Firefox. So, we have downloaded the Selenium drivers for them and copied them into a dedicated folder, as shown in fig. 15.3.

Local Disk (C:) > Work > Manning > drivers				
Name	Date modified	Type	Size	
chromedriver.exe	8/19/2019 6:32 AM	Application	8,498 KB	
IEDriverServer.exe	8/2/2018 7:40 AM	Application	3,228 KB	
geckodriver.exe	1/29/2019 12:49 AM	Application	14,140 KB	

Figure 15.3 The folder containing the Selenium drivers

Please note that you will need exactly the driver corresponding to the browser installed on your computer. For example, if your Google Chrome version is 79, you will be able to use only version 79 of the driver. Versions as 77, 78 or 80 will not work.

We need to include the folder with the Selenium drivers on the path of the operating system. On Windows, the most used operating system, you can do this by accessing This PC -> Properties -> Advanced system settings -> Environment variables -> Path -> Edit (fig. 15.4). For doing the same on other operating systems, consult their documentation.

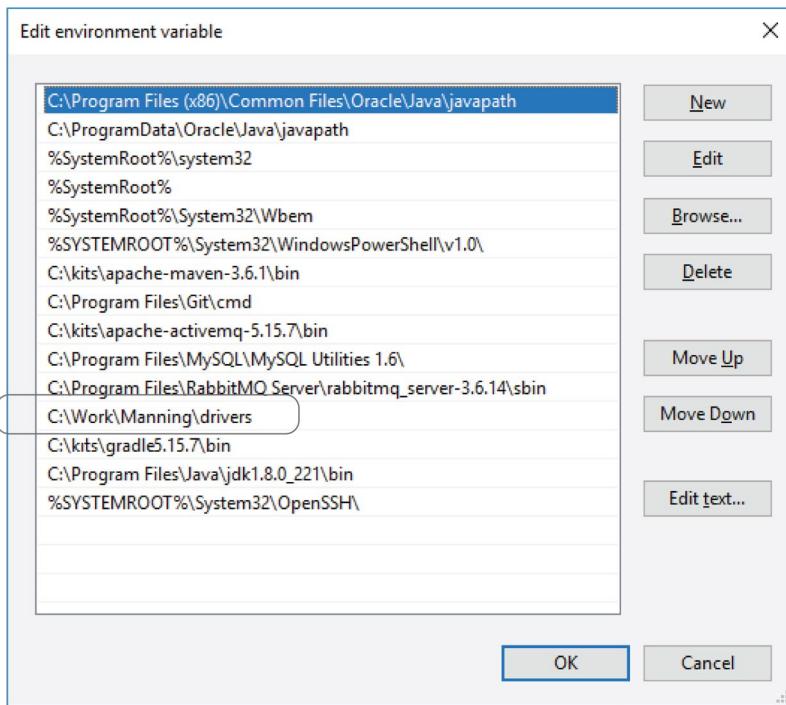


Figure 15.4 The path to the Selenium drivers folder added on the operating system path

We have now setup the environment, we are ready to proceed to write the Selenium tests.

15.5.1 Testing for a specific web browser

Our tests will use specific browsers for the beginning. We'll put face to face two tests accessing the Manning home page and the Google home page and verifying that their titles are the expected ones. In listings 15.14 and 15.15 we'll use Chrome and Firefox into two separate test classes in order to do this and JUnit 5 annotated methods.

Listing 15.14 Accessing the Manning and the Google homepages with Chrome

```
public class ChromeSeleniumTest {  
  
    private WebDriver driver; #A  
  
    @BeforeEach  
    void setUp() {  
        driver = new ChromeDriver(); #B  
    }  
  
    @Test  
    void testChromeManning() {  
        driver.get("https://www.manning.com/"); #C  
    }  
}
```

```

        assertThat(driver.getTitle(), is("Manning | Home")); #C
    }

    @Test
    void testChromeGoogle() {
        driver.get("https://www.google.com");
        assertThat(driver.getTitle(), is("Google")); #D
    }

    @AfterEach
    void tearDown() {
        driver.quit(); #E
    }
}

```

Listing 15.15 Accessing the Manning and the Google homepages with Firefox

```

public class FirefoxSeleniumTest {

    private WebDriver driver; #A'

    @BeforeEach
    void setUp() {
        driver = new FirefoxDriver(); #B'
    }

    @Test
    void testFirefoxManning() {
        driver.get("https://www.manning.com/");
        assertThat(driver.getTitle(), is("Manning | Home")); #C'
        #C'
    }

    @Test
    void testFirefoxGoogle() {
        driver.get("https://www.google.com");
        assertThat(driver.getTitle(), is("Google")); #D'
        #D'
    }

    @AfterEach
    void tearDown() {
        driver.quit(); #E'
    }
}

```

In the listings above, we are doing the following:

- We are declaring a web driver (#A - #A') and we are initializing it as a `ChromeDriver` (#B) and as a `FirefoxDriver` (#B') respectively.
- The first test is accessing the Manning home page and is checking that the title of the page is "Manning | Home" (#C - #C'). We do this using Hamcrest matchers (see chapter 2).
- The second test is accessing the Google home page and is checking that the title of the page is "Google" (#D - #D'). We also do this using Hamcrest matchers.
- After executing each JUnit 5 test, we make sure that we call the `quit` method of the

web driver, which will close all the open browser windows and the driver instance becomes garbage collectible, meaning available to be removed from the memory (#E - #E').

If we run each of the tests above, we will see that the execution will open the Chrome and the Firefox browsers respectively, access the Manning and the Google websites and validate their titles.

15.5.2 Testing navigation using a web browser

Our next test will access the Wikipedia website, then find an element on the webpage and click on it. In listing 15.16 we'll use JUnit 5 annotations and Firefox in order to do this.

Listing 15.16 Finding an element on a page with Firefox

```
public class WikipediaAccessTest {  
    private WebDriver driver; #A  
  
    @BeforeEach  
    void setUp() {  
        driver = new FirefoxDriver(); #B  
    }  
  
    @Test  
    void testWikipediaAccess() {  
        driver.get("https://en.wikipedia.org/"); #C  
        assertThat(driver.getTitle(), #C  
                  is("Wikipedia, the free encyclopedia")); #C  
  
        WebElement contents = driver.findElement(By.linkText("Contents")); #D  
        assertTrue(contents.isDisplayed()); #D  
  
        contents.click(); #E  
        assertThat(driver.getTitle(), #E  
                  is("Wikipedia:Contents - Wikipedia ")); #E  
    }  
  
    @AfterEach  
    void tearDown() {  
        driver.quit(); #F  
    }  
}
```

In the listings above, we are doing the following:

- We are declaring a web driver as RemoteWebDriver (#A). RemoteWebDriver is a class that implements the WebDriver interface and is extended by browser classes as FirefoxDriver, ChromeDriver or InternetExplorerDriver. We are declaring the web driver as a RemoteWebDriver in order to be able to call the method that is finding elements on a webpage and that is not declared by the WebDriver interface.
- We are initializing the web driver as a FirefoxDriver (#B).
- We are accessing the Wikipedia home page and checking that the title of the page is

“Wikipedia, the free encyclopedia” (#C).

- We are looking for the “Contents” element and checking that it is displayed (#D).
- We are clicking on the “Contents” element and checking that the newly displayed page has the title “Wikipedia:Contents - Wikipedia” (#E).
- After executing the test, we make sure that we call the `quit` method of the web driver, which will close all the open browser windows and the driver instance becomes garbage collectible, meaning available to be removed from the memory (#F).

Exercise

The reader can easily change this program in order to use another web driver, as `ChromeDriver` or `InternetExplorerDriver`.

15.5.3 Testing more than one web browser

We would like to emphasize the advantages of the new JUnit 5 features and put them in practice into our Selenium tests. We will run the same test class with more than one browser. In listing 15.17, we rework our previous example accessing the Manning and Google homepages as JUnit 5 parameterized tests running with different browsers.

Listing 15.17 Accessing the Manning and Google homepages with different browsers

```
public class MultiBrowserSeleniumTest {  
  
    public static Collection<WebDriver> getBrowserVersions() { #A  
        return Arrays.asList(new WebDriver[] {new FirefoxDriver(), new #A  
            ChromeDriver(), new InternetExplorerDriver()}); #A  
    } #A  
  
    @ParameterizedTest #B  
    @MethodSource("getBrowserVersions") #B  
    void testManningAccess(WebDriver driver) { #B  
        driver.get("https://www.manning.com/"); #C  
        assertThat(driver.getTitle(), is("Manning | Home")); #C  
        driver.quit(); #D  
    }  
  
    @ParameterizedTest #B  
    @MethodSource("getBrowserVersions") #B  
    void testGoogleAccess(WebDriver driver) { #B  
        driver.get("https://www.google.com"); #E  
        assertThat(driver.getTitle(), is("Google")); #E  
        driver.quit(); #F  
    }  
}
```

In the listing above, we are doing the following:

- We are creating the `getBrowserVersions` method that will serve as a method source for injecting the argument of the parameterized tests (#A). The method is

returning a collection of web drivers (`FirefoxDriver`, `ChromeDriver`, and `InternetExplorerDriver`) that will be used one by one to be injected into the parameterized tests.

- We define two parameterized tests, for which the source that is providing the test arguments is the `getBrowserVersions` method (#B). This means that each of the parameterized methods will be executed a number of times equal to the size of the collection returned by the `getBrowserVersions` method – and it will use different browsers each time when it will run.
- The first test is accessing the Manning home page and is checking that the title of the page is “Manning | Home” (#C). We do this using Hamcrest matchers (see chapter 2). The JUnit 5 parameterized test will be executed three times, once for each of the web drivers provided by the `getBrowserVersions` method. After executing each test, we make sure that we call the `quit` method of the web driver, which will close all the open browser windows and the driver instance becomes garbage collectible, meaning available to be removed from the memory (#D).
- The second test is accessing the Google home page and is checking that the title of the page is “Google” (#E). The test will also be executed three times, once for each of the web drivers provided by the `getBrowserVersions` method. We also do this using Hamcrest matchers. After executing each test, we also call the `quit` method of the web driver (#F).

If we run the test class above, we will see that the execution will open the Firefox, Chrome and Internet Explorer browsers for each test, access the Manning and the Google websites and validate their titles. We will have two tests executed through three browsers each – this is a total of six test executions.

15.5.4 Testing Google search and navigation using different web browsers

We will show how to test a Google search, clicking on one of the links that we obtain from the search engine and navigate inside one of the results. In listing 15.18, we search for “en.wikipedia.org.” using Google, we jump to the first result and then to one element inside it.

Listing 15.18 Testing Google search and navigating inside the Wikipedia website

```
public class GoogleSearchTest {  
  
    public static Collection<RemoteWebDriver> getBrowserVersions() { #A  
        return Arrays.asList(new RemoteWebDriver[] {new FirefoxDriver(), #A  
            new ChromeDriver(), new InternetExplorerDriver()}); #A  
    } #A  
  
    @ParameterizedTest  
    @MethodSource("getBrowserVersions") #B  
    void testGoogleSearch(RemoteWebDriver driver) { #B  
        driver.get("http://www.google.com"); #C  
        WebElement element = driver.findElement(By.name("q")); #C  
        element.sendKeys("en.wikipedia.org"); #D  
        driver.findElement(By.name("q")).sendKeys(Keys.ENTER); #D  
    }  
}
```

```

WebElement myDynamicElement = (new WebDriverWait(driver, 10))          #E
    .until(ExpectedConditions
        .presenceOfElementLocated(By.id("resultStats")));
#E
#E

List<WebElement> findElements =
    driver.findElements(By.xpath("//*[@id='rso']//a/h3"));           #F
#F

findElements.get(0).click();                                              #G
#G

assertEquals("https://en.wikipedia.org/wiki/Main_Page",                 #G
            driver.getCurrentUrl());                                     #G
assertThat(driver.getTitle(),                                           #G
           is("Wikipedia, the free encyclopedia"));                   #G

WebElement contents = driver.findElementByLinkText("Contents");          #H
assertTrue(contents.isDisplayed());                                       #H

contents.click();                                                       #I
assertThat(driver.getTitle(),                                           #I
           is("Wikipedia:Contents - Wikipedia"));                      #I

driver.quit();                                                        #J
#J
}
}

```

In the listing above, we are doing the following:

- We are creating the `getBrowserVersions` method that will serve as a method source for injecting the argument of the parameterized tests (#A). The method is returning a collection of `RemoteWebDrivers` (`FirefoxDriver`, `ChromeDriver`, and `InternetExplorerDriver`) that will be used one by one to be injected into the parameterized tests. We need `RemoteWebDrivers` in order to be able to call the method that is finding elements on a webpage and that is not declared by the `WebDriver` interface.
- We define one JUnit 5 parameterized test, for which the source that is providing the test arguments is the `getBrowserVersions` method (#B). This means that the parameterized method will be executed a number of times equal to the size of the collection returned by the `getBrowserVersions` method.
- We are accessing the www.google.com website and look for the element with the "q" name (#C). This is the name of the input edit box where we need to insert the text to search for and can be obtained by right-clicking on it and choosing "Inspect" or "Inspect element" (depending on the browser). We will get a result as in fig. 15.5.
- We are inserting the "en.wikipedia.org" text to search for using Google and we are pressing Enter (#D).
- We wait until the Google page shows the result, but no longer than 10 seconds (#E).
- We get all elements returned by the Google search using an XPath (#F). XPath is a query language for finding elements in XML. What you need to know at this time is only that the `//*[@id='rso']//a/h3` XPath will provide you the list of all elements returned by the Google search. For more comprehensive documentation about working

with XPath in Selenium, you may study the content of the <https://www.guru99.com/xpath-selenium.html> website.

- We click on the first element on the list and we check the URL and the title of the newly accessed page (#G).
- Inside this new page, we are looking for the element with the “Contents” text and check that it is displayed (#H). Then, we click on this element and we are checking that the title of the newly accessed page is "Wikipedia:Contents - Wikipedia" (#I).
- After executing the test, we call the `quit` method of the web driver, which will close all the open browser windows and the driver instance becomes garbage collectible, meaning available to be removed from the memory (#J).

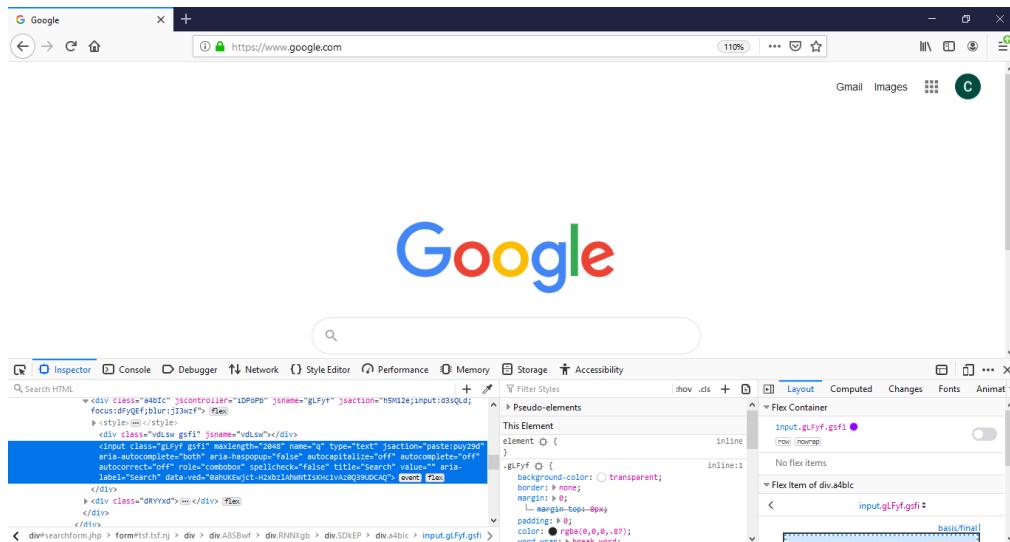


Figure 15.5 Inspecting the name of the input edit box on the Google homepage with the help of Firefox

If we run the test above, we will see that the execution will open the Firefox, Chrome and Internet Explorer browsers. We will have one test executed through three browsers – this is a total of three test executions.

15.5.5 Testing the authentication scenario to a website

Tested Data Systems is an outsourcing company having many customers that need to interact with their applications using a website where they need to authenticate. Authentication is critical for such an application, as it is verifying the identity of the user and allowing the access. Consequently, John needs to test authentication by writing tests to follow both successful and unsuccessful scenarios. John will choose Selenium as he would also like to visually follow the interaction with the web site.

For this purpose, we will use as test web site <https://the-internet.herokuapp.com/>. This one provides many functionalities that may be automatically tested, among which the form authentication that we need for our scenario. The class from listing 15.19 represents the interaction with the homepage of the <https://the-internet.herokuapp.com/> website (fig. 15.6).

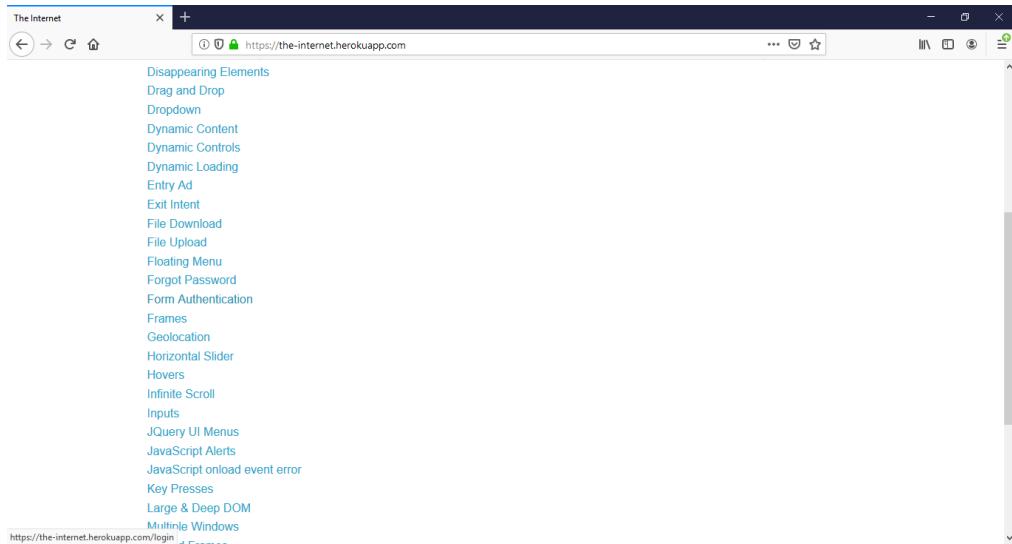


Figure 15.6 The <https://the-internet.herokuapp.com/> homepage

Listing 15.19 The class describing the homepage of the tested website

```
public class Homepage {  
    private WebDriver webDriver; #A  
  
    public Homepage(WebDriver webDriver) {  
        this.webDriver = webDriver; #B  
    }  
  
    public LoginPage openFormAuthentication() {  
        webDriver.get("https://the-internet.herokuapp.com/"); #C  
        webDriver.findElement(By.cssSelector("[href=\"/login\"]")) #D  
            .click();  
        return new LoginPage(webDriver); #E  
    }  
}
```

In the listing above, we are doing the following:

- The `Homepage` class contains a `private WebDriver` field that will be used to interact with the website (#A). It is initialized by the constructor of the class (#B).
- Into the `openFormAuthentication` class, we are accessing the <https://the-internet.herokuapp.com/> website (#C). We are looking for the element having the

`login` hyperlink (`href`) and we click on it (#D). We return a new `LoginPage` (to be described into the next listing), receiving the web driver as an argument of the constructor.

The class from listing 15.20 represents the interaction with the login page from <https://the-internet.herokuapp.com/login/> (fig. 15.7).

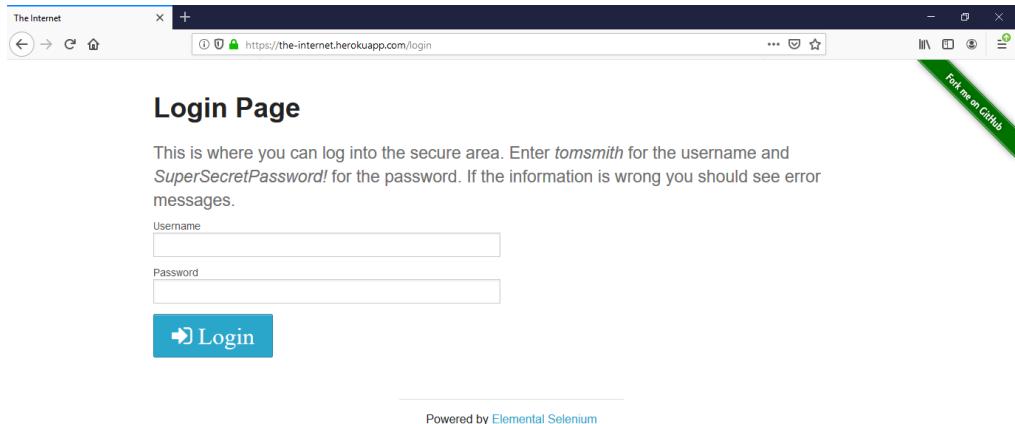


Figure 15.7 The login page <https://the-internet.herokuapp.com/login/>

Listing 15.20 The class describing the login page of the tested website

```
public class LoginPage {  
  
    private WebDriver webDriver; #A  
  
    public LoginPage(WebDriver webDriver) {  
        this.webDriver = webDriver; #B  
    }  
  
    public LoginPage loginWith(String username, String password) {  
        webDriver.findElement(By.id("username")).sendKeys(username); #C  
        webDriver.findElement(By.id("password")).sendKeys(password); #C  
        webDriver.findElement(By.cssSelector("#login button")).click(); #D  
  
        return this; #E  
    }  
  
    public void thenLoginSuccessful() {  
        assertTrue(webDriver #F  
            .findElement(By.cssSelector("#flash.success")) #F  
            .isDisplayed()); #F  
        assertTrue(webDriver #F  
    }
```

```

        .findElement(By.cssSelector("[href=\"/logout\"]"))
        .isDisplayed()); #F
    }

    public void thenLoginUnsuccessful() {
        assertTrue(webDriver.findElement(By.id("username")).isDisplayed()); #G
        assertTrue(webDriver.findElement(By.id("password")).isDisplayed()); #G
    }
}

```

In the listing above, we are doing the following:

- The LoginPage class contains a private WebDriver field that will be used to interact with the website (#A). It is initialized by the constructor of the class (#B).
- The loginWith method will find the elements with the ids username and password and write into their content the string arguments username and password from the method (#C). Then, we will find the login button using a CSS selector and click on it (#D). The method will return the same object, as we intend to execute the thenLoginSuccessful and thenLoginUnsuccessful methods on it (#E).
- We check the successful login by verifying that the elements with CSS selector #flash.success (the green area from fig. 15.8) and with the hyperlink (href) logout (the logout button from fig. 15.8) are displayed on the page (#F). Remember that we can take the names of an element by right-clicking on it and choosing "Inspect" or "Inspect element" (depending on the browser).
- We check the unsuccessful login by verifying that the elements having ids username and password are displayed on the page (#G). This is because, after an unsuccessful login attempt, we remain on the same page (fig. 15.9).

CSS SELECTOR — A CSS selector is the combination of an element selector and a selector value that identifies the web element within a web page. What you need to know at this time is only that the #login button CSS selector will find this element on the webpage. For a more comprehensive documentation about CSS selectors, you may study the content of the https://www.w3schools.com/cssref/css_selectors.asp website.

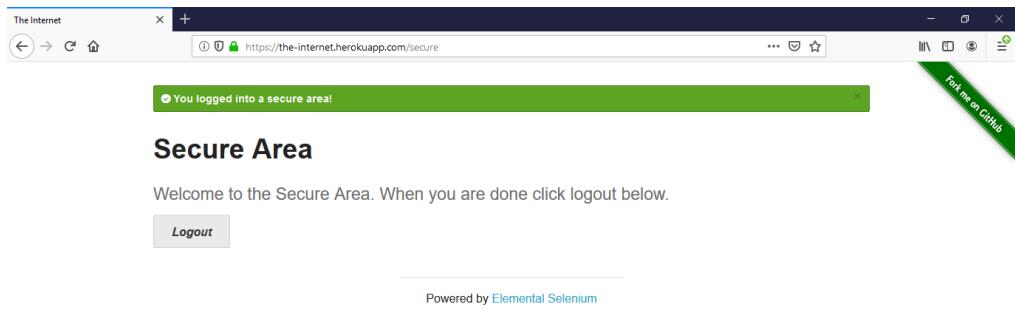


Figure 15.8 The successful login page <https://the-internet.herokuapp.com/secure/>

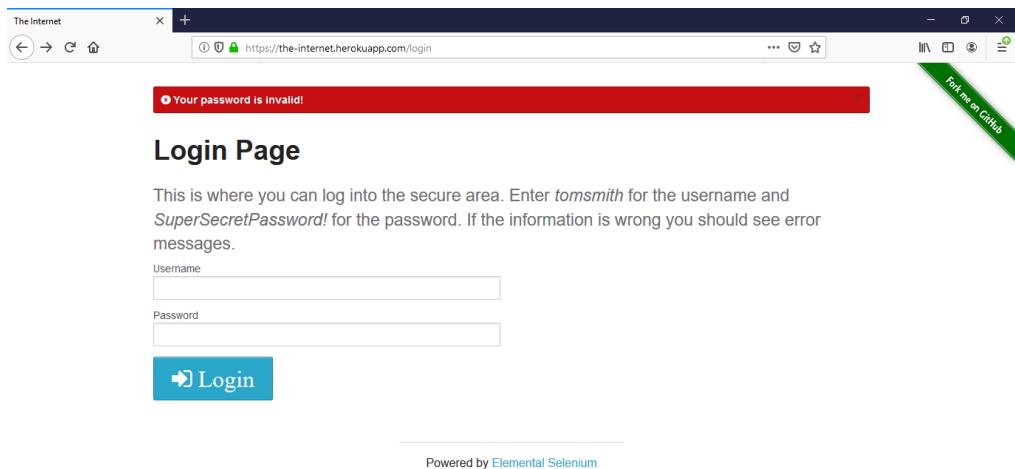


Figure 15.9 The unsuccessful login page <https://the-internet.herokuapp.com/secure/>

John will use the `Homepage` and `LoginPage` classes that he has created to run the successful login and unsuccessful login testing scenarios, as shown in listing 15.21. Tom Smith is a test user that accesses a web interface for the flights management application that we have

already introduced. The credentials that allow the login are username `tomsmith` and password `SuperSecretPassword!`. John will work with this test user (we'll never use real credentials in a test as it is a breach of security). John will create a test with the correct credentials of the test user (never to be changed, as the test will otherwise fail) and another test with some incorrect ones.

Listing 15.21 The successful and unsuccessful login tests

```
public class LoginTest {  
  
    private Homepage homepage; #A  
  
    public static Collection<WebDriver> getBrowserVersions() { #B  
        return Arrays.asList(new WebDriver[] {new FirefoxDriver(), #B  
                                         new ChromeDriver(), new InternetExplorerDriver()}); #B  
    } #B  
  
    @ParameterizedTest #C  
    @MethodSource("getBrowserVersions") #C  
    public void loginWithValidCredentials(WebDriver webDriver) { #C  
        homepage = new Homepage(webDriver); #D  
        homepage #E  
            .openFormAuthentication() #E  
            .loginWith("tomsmith", "SuperSecretPassword!"); #E  
            .thenLoginSuccessful(); #E  
        webDriver.quit(); #G  
    } #G  
  
    @ParameterizedTest #C  
    @MethodSource("getBrowserVersions") #C  
    public void loginWithInvalidCredentials(WebDriver webDriver) { #C  
        homepage = new Homepage(webDriver); #D  
        homepage #F  
            .openFormAuthentication() #F  
            .loginWith("tomsmith", "SuperSecretPassword"); #F  
            .thenLoginUnsuccessful(); #F  
        webDriver.quit(); #G  
    } #G  
}
```

In the listing above, we are doing the following:

- We are defining a `Homepage` field that we are going to initialize during the execution of the tests (#A).
- We are creating the `getBrowserVersions` method that will serve as a method source for injecting the argument of the parameterized tests (#B). The method is returning a collection of `WebDrivers` (`FirefoxDriver`, `ChromeDriver`, and `InternetExplorerDriver`) that will be used one by one to be injected into the parameterized tests.
- We define two JUnit 5 parameterized tests, for which the source that is providing the test arguments is the `getBrowserVersions` method (#C). This means that the parameterized method will be executed a number of times equal to the size of the

collection returned by the `getBrowserVersions` method.

- We start each parameterized test by initializing a `Homepage` variable by passing to its constructor the web driver as argument (#D).
- The first test will open the form authentication, login with valid username and password and check that the login is successful (#E).
- The second test will open the form authentication, login with invalid username and password and check that the login is unsuccessful (#F).
- After executing each test, we call the `quit` method of the web driver, which will close all the open browser windows and the driver instance becomes garbage collectible (#G).

This way, John has implemented the two scenarios of the successful and unsuccessful login to the web site. More than this, John has created, through the `Homepage` and `LoginPage` classes, the basics of a testing library that other developers that will join the project will be able to use. And, if we see how the two tests from listing 15.21 look like, we notice that the calls to the testing methods simply flow one after the other. The scenario of the test is easy to understand by any newcomer. This will really speed up the development of Selenium tests at Tested Data Systems!

15.6 HtmlUnit vs. Selenium

Here is a recap of the similarities and differences you will find between HtmlUnit and Selenium.

The similarities are that both are free and open-source and both their last versions at the time of writing this chapter (HtmlUnit 2.36.0 and Selenium 3.141.59) require Java 8 as the minimum platform to run.

The major difference between the two is that HtmlUnit emulates a specific web browser while Selenium drives a real web browser process.

The HtmlUnit pros are that, being a headless web browser, the execution of the tests is faster. It also provides its own domain-specific set of assertions.

The Selenium pros are that the API is simpler and drives native browsers, which guarantees that the behavior of the tests is as close as possible to the interaction with the real user. Selenium also supports multiple languages and provides a “visual effect” for executing a larger scenario – a reason for which it has been preferred for testing the authentication functionality.

Finally:

Use HtmlUnit when...

Use HtmlUnit when your application is independent of operating system features and browser-specific implementations not accounted for by HtmlUnit.

Use Selenium when...

Use Selenium when you require validation of specific browsers and operating systems, especially if the application takes advantage of or depends on a browser-specific implementation.

In the next chapter, we will start building and testing applications using one of the most popular Java frameworks from today: the Spring framework.

15.7 Summary

This chapter has covered :

- Examining the presentation layer testing
- Operating with HtmlUnit, like a headless browser open-source tool
- Operating with Selenium, an open-source tool that drives various browsers programmatically
- Testing HTML Assertions (`assertTitleEquals`, `assertTextPresent`, `notNull`) and the functionality of different web browsers (Mozilla Firefox, Google Chrome, Internet Explorer) and creating standalone tests using the HtmlUnit tool.
- Testing forms, web navigation and frames, and websites developed with JavaScript using the HtmlUnit tool.
- Testing the operation of searching on Google, selecting the Manning site link and navigating on it using the Selenium tool and different web browsers (Mozilla Firefox, Google Chrome, Internet Explorer).
- Creating and testing an authentication scenario to a website using the Selenium tool.

16

Testing Spring applications

This chapter covers:

- Introducing the Spring Framework
- Introducing Dependency Injection
- Executing the first steps to using and testing a Spring application
- Demonstrating the usage of the `SpringExtension` for JUnit Jupiter
- Developing a new feature into a Spring application and testing it with JUnit 5

Dependency Injection" is a 25-dollar term for a 5-cent concept.

- James Shore

Spring is a lightweight but at the same time flexible and universal framework used for creating Java applications. It is a framework with open-source code. It is not dedicated to any particular layer - this means that it can be used at the level of any layer of a Java application to develop it. Spring includes several separate frameworks. This chapter will focus on the basis of Spring - the Dependency Injection (or Inversion of Control) pattern and how to test core Spring applications with the help of JUnit 5.

16.1 Introducing the Spring Framework

A software framework is an abstraction in which software with generic functionality can be changed by code that the user writes. This enables the creation of particular software. A framework supports the development of software applications by providing a working paradigm.

Rod Johnson created Spring in 2003. Spring took its rise from the book "Expert One-on-One Java J2EE Design and Development". The basic idea behind Spring is to simplify the traditional approach to designing enterprise applications.

The key difference between a library and a framework is "Inversion of Control". When you call a method from a library, you are in control. But with a framework, the control is inverted: the framework calls you (fig. 16.1).

A library is a collection of classes or functions. A library will enable code reuse: a developer can use the code that already created by other developers. Usually, a library is dedicated to a domain-specific area. For example, there are libraries of computer graphics, which can let the developer quickly building three-dimensional scenes and displaying them on the computer screen.

A framework inverts control. The programmer must follow the paradigm provided by the framework and fill out the own code. A framework defines a skeleton where the programmer will insert the features to fill out the skeleton. Our code will be under the control of the framework and the framework will call it. This way, the programmers will focus on implementing the business logic rather than on the design.

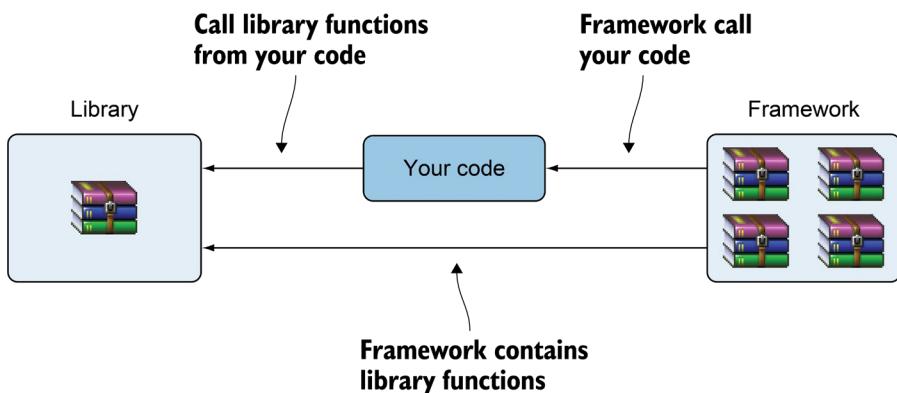


Figure 16.1 Your code is calling the library; the framework calls your code

The Spring Framework provides a comprehensive infrastructure for developing Java applications. It handles the infrastructure so that we can focus on our application and enables us to build applications from "plain old Java objects" (POJOs).

16.2 Introducing Dependency Injection

A Java application typically consists of objects that collaborate to form the application properly. The objects in an application have dependencies on each other. Java cannot organize the building blocks of an application, leaving the task to developers and architects.

We can use design patterns (Factory, Builder, Proxy, Decorator, etc.) to compose classes and objects, but this burden is on the side of the developer.

Spring implements various design patterns by itself. The Spring Framework Dependency Injection (Inversion of Control) pattern provides a means of composing disparate components into a fully working application.

We'll start from the traditional approach, where dependencies need to be managed by the developer, at the level of the code. In the flights management application developed at Tested Data System, the passengers that are tracked belong to a country. In fig. 16.2, a Passenger object is dependent on a Country object. The developer will initialize this dependency directly into the code (listing 16.1 and 16.2).

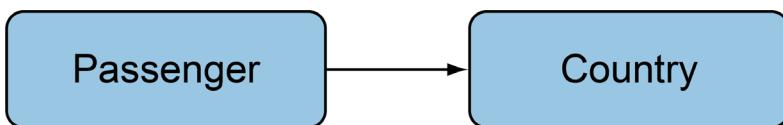


Figure 16.2 The Passenger belongs to a Country, the Passenger object is dependent on a Country object

Listing 16.1 The Country class

```
public class Country {  
    private String name; #A  
    private String codeName; #B  
  
    public Country(String name, String codeName) { #C  
        this.name = name; #C  
        this.codeName = codeName; #C  
    } #C  
  
    public String getName() { #A  
        return name; #A  
    } #A  
  
    public String getCodeName() { #B  
        return codeName; #B  
    } #B  
}
```

In listing 16.1 we are creating the Country class that includes the following:

- A name field together with a getter for it (#A) and a codeName field together with a getter for it (#B).
- A constructor that initializes the name and codeName fields (#C).

Listing 16.2 The Passenger class

```
public class Passenger {  
    private String name; #A  
    private Country country; #B  
  
    public Passenger(String name) { #C
```

```

        this.name = name;                                #C
        this.country = new Country("USA", "US");         #C
    }

    public String getName() {                          #A
        return name;                                  #A
    }

    public Country getCountry() {                     #B
        return country;                             #B
    }

}

```

In listing 16.2 we are creating the `Passenger` class that includes the following:

- A `name` field together with a getter for it (#A) and a `country` field together with a getter for it (#B).
- A constructor that initializes the `name` and `country` fields (#C). We see that the `country` is effectively initialized into the `Passenger` constructor, meaning that there is a tight coupling between the `country` and the passenger.

With this approach, we are in the general situation from fig. 16.3.

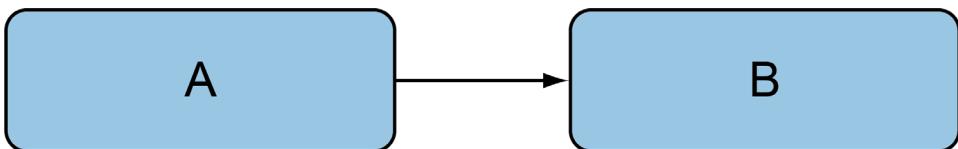


Figure 16.3 The direct general dependency between objects A and B (Passenger and Country in the particular previous example)

This approach comes with a series of shortcomings:

- Class A directly depends on class B.
- It is impossible to test A separately from B.
- The lifetime of object B depends on A – it is impossible to use an object of type B in other places (although class B can be reused).
- It is not possible to replace B with another implementation.

These shortcomings favor the new approach of Dependency Injection.

Through the Dependency Injection approach (also known as Inversion of Control), objects are inserted into a container that injects the dependencies when it creates the object. This process is fundamentally the inverse, hence the name Inversion of Control (IoC). Martin Fowler suggested the name of Dependency Injection (DI) because it better reflects the essence of the pattern. The basic idea is to eliminate the dependency of application components from certain implementation and to delegate to the container the rights to control classes instantiation.

Moving to this new approach for the previous example with Passenger and Country, we will eliminate the direct dependency between the objects and we'll rewrite the Passenger class listing 16.3).

Listing 16.3 The Passenger class eliminating the tight coupling with Country

```
public class Passenger {  
    private String name;  
    private Country country;  
  
    public Passenger(String name) { #A  
        this.name = name; #A  
    } #A  
  
    public String getName() {  
        return name;  
    }  
  
    public Country getCountry() {  
        return country;  
    }  
  
    public void setCountry(Country country) { #B  
        this.country = country; #B  
    } #B  
}
```

The change from listing 16.3 is at the level of its constructor - it will no longer create the dependent country by itself (#A). The country may be set through the `setCountry` method (#B). As shown in fig. 16.4, direct dependency has been removed.



Figure 16.4 The Passenger and Country classes with no direct dependency

We'll delegate the management of the dependencies to the container that will be instructed about how to do this through the configuration information shown in listing 16.4. We have inserted all information into the `application-context.xml` file that will be used by the Spring Framework for creating and managing the objects and their dependencies.

XML is still the traditional way of configuring Spring applications. It has the advantage of being non-intrusive – this means that your code will be unaware of the fact that it is used by a framework and is not mixed with any external dependencies. More than this, when the configuration changes, the code does not need to be recompiled. This idea may be beneficial for testers who have less technical knowledge.

To be comprehensive, in this chapter we'll also show alternatives of configuring Spring, and we start with this XML one – easier to understand and to follow, at least at the beginning.

The objects under the control of the Spring Framework container are generally called beans. According to the definition of *beans* in the Spring Framework documentation:

BEAN

In Spring, a bean is an object from the backbone of the application, managed by a Spring IoC container.

Listing 16.4 The application-context.xml configuration information

```
<bean id="passenger" class="com.manning.junitbook.spring.Passenger">          #A
    <constructor-arg name="name" value="John Smith"/>                      #B
    <property name="country" ref="country"/>                                #C
</bean>                                                               #D

<bean id="country" class="com.manning.junitbook.spring.Country">          #E
    <constructor-arg name="name" value="USA"/>                         #F
    <constructor-arg name="codeName" value="US"/>                         #F
</bean>                                                               #G
```

In the listing above, we see the following:

- We are declaring the passenger bean belonging to the class com.manning.junitbook.spring.Passenger. We initialize it by passing to it "John Smith" as the argument of the constructor (#B) and we are setting the country passing the reference to the country bean to the setCountry setter (#C). Then, we close the bean definition (#D).
- We are declaring the country bean belonging to the class com.manning.junitbook.spring.Country. We initialize it by passing to it "USA" and "US" as arguments of the constructor (#F). Then, we close the bean definition (#G).

In order to access the beans created by the container, we will do something as in listing 16.5.

Listing 16.5 Accessing the beans defined in the application-context.xml file

```
ClassPathXmlApplicationContext context =           #A
    new ClassPathXmlApplicationContext(          #A
        "classpath:application-context.xml");   #A
Passenger passenger = (Passenger) context.getBean("passenger");      #B
Country country = (Country) context.getBean("country");                #C
```

In the listing above, we see the following:

- We are creating a context variable of type ClassPathXmlApplicationContext and we are initializing it to point to the application-context.xml file from the classpath (#A). The class ClassPathXmlApplicationContext belongs to the Spring Framework and we'll show a little later how to introduce the dependency that it

belongs to into the Maven pom.xml file.

- We are requesting the passenger bean from the container (#B), then we are requesting the country bean (#C). From now on, we can use the passenger and country variables in our program.

As advantages of this approach, we can enumerate the following:

- When we request an object of any type, the container will return it.
- Passenger and Country are not dependent and do not depend on any outer libraries - they are Plain Old Java Objects (POJOs).
- The application-context.xml file documents the system and objects dependencies.
- The container controls the lifetime of the created objects.
- We facilitate the reuse of the classes and components.
- The code is cleaner (classes do not initiate auxiliary objects).
- The unit testing is simplified; the classes are simpler and not cluttered with dependencies.
- It is very easy to make changes to object dependencies in the system. We simply need to change the application-context.xml file, but we do not need to recompile the Java source files.
- It is especially recommended to insert to the DI (IoC) container the objects for which the implementation may change.

Thus, we have made the step between the traditional approach, where the dependencies are to be found inside the code, to the Dependency Injection (Inversion of Control) approach, where objects do not know anything about each other (fig. 16.5).

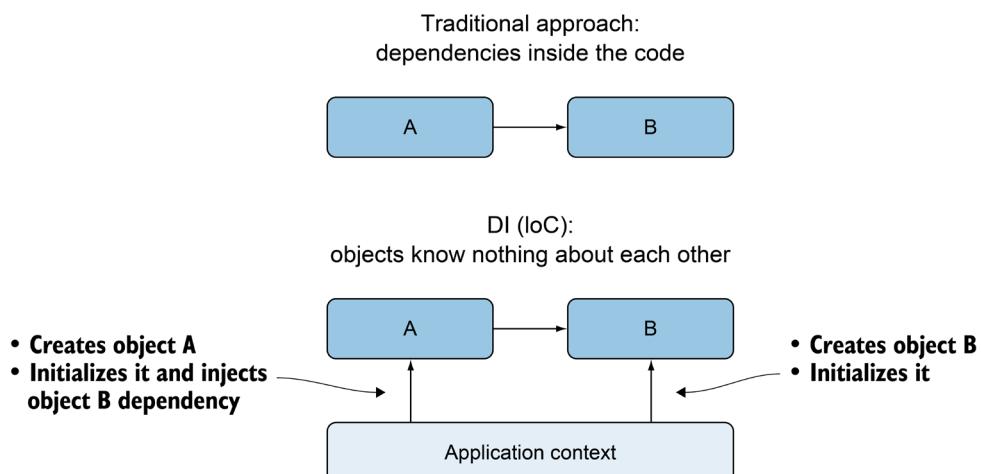


Figure 16.5 The step from the traditional approach to the Dependency Injection (Inversion of Control)

In general, the work of Spring container can be represented as in fig. 16.6. When instantiating and initiating the container, the application classes are combined with metadata (container configuration) and at the output, we get a fully configured and ready-to-work application.

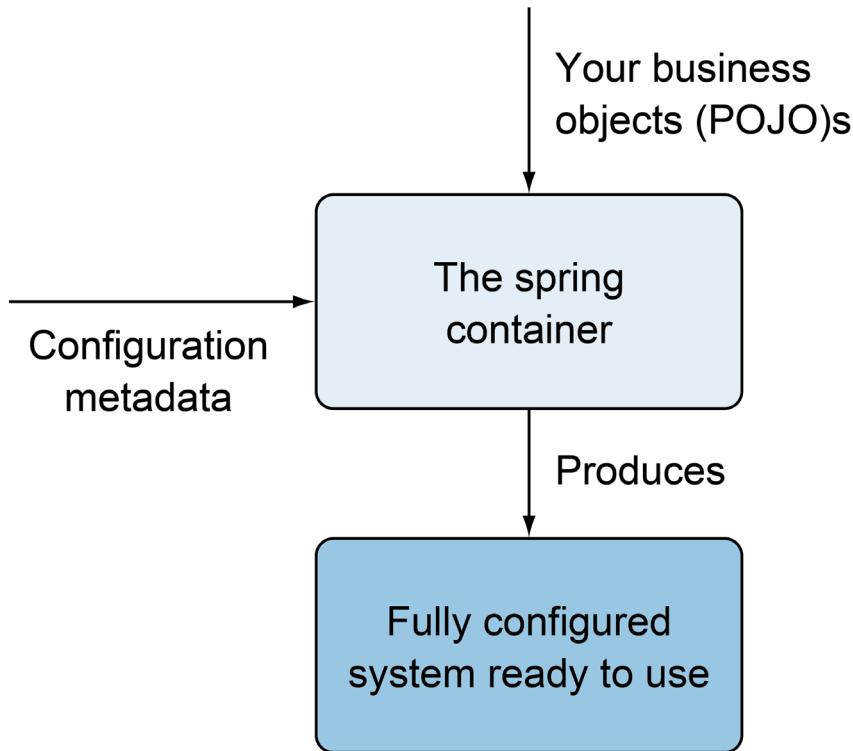


Figure 16.6 The functionality of the Spring container puts head to head the POJOs and the configuration metadata

16.3 The first steps to using and testing a Spring application

Due to the advantages that we have presented, the Spring Framework has already been introduced by the Tested Data Systems company into some of its projects. Among these is the flights management application that we have already talked about. The first introduction of Spring has happened before the introduction of JUnit 5, so we'll first follow how this has taken place.

16.3.1 Creating the Spring context programmatically

A few years ago, John was responsible for first moving the flight management application to Spring 4 and to test it using JUnit 4. It is very useful to see how John accomplished this task

at that time. It is very possible that you will join an application using Spring 4 and/or JUnit 4 and need to continue its development or to migrate it to Spring 5 and JUnit 5. Therefore, we will take a close look at what such an application looks like.

The first thing that John has to do to work with Spring 4 and JUnit 4 is to introduce the needed dependencies into the Maven pom.xml file (listing 16.6).

Listing 16.6 The Spring 4 and JUnit 4 dependencies into the pom.xml configuration

```
<dependency> #A
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>4.2.5.RELEASE</version>
</dependency> #A

<dependency> #B
    <groupId>org.springframework</groupId>
    <artifactId>spring-test</artifactId>
    <version>4.2.5.RELEASE</version>
</dependency> #B

<dependency> #C
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.12</version>
</dependency> #C
```

In the listing above we are doing the following:

- We are adding the `spring-context` Maven dependency (#A). This is necessary for using classes as `ClassPathXmlApplicationContext` that are needed to load the application context or the `@Autowired` annotation that we'll introduce later.
- We are adding the `spring-test` Maven dependency (#B). This is needed for using the `SpringJUnit4ClassRunner` runner (we'll demonstrate its usage).
- We are also adding the JUnit 4.12 dependency (#C). Remember, the first introduction of Spring into the flights management application was done a few years ago, before JUnit 5 existed.

The first test that John wrote at that time was one that loads a passenger from the Spring container and verifies its correctness (listing 16.7).

Listing 16.7 The first unit test for a Spring application

```
public class SimpleAppTest { #A

    private static final String APPLICATION_CONTEXT_XML_FILE_NAME = #B
        "classpath:application-context.xml"; #B

    private ClassPathXmlApplicationContext context; #C

    private Passenger expectedPassenger; #D

    @Before
    public void setUp() {
```

```

        context = new ClassPathXmlApplicationContext(
            APPLICATION_CONTEXT_XML_FILE_NAME);           #E
        expectedPassenger = getExpectedPassenger();      #E
    }

    @Test
    public void testInitPassenger() {
        Passenger passenger = (Passenger) context.getBean("passenger");   #G
        assertEquals(expectedPassenger, passenger);                      #H
    }
}

```

In the listing above we are doing the following:

- We are creating a `SimpleAppTest` class (#A). It defines the string to access the `application-context.xml` file on the classpath (#B), the Spring context (#C) and the `expectedPassenger` (#D), to be constructed programmatically and to be compared with the one extracted from the Spring context.
- Before the execution of each test, we are creating the context based on the string to access the `application-context.xml` file on the classpath (#E) and we are also creating the `expectedPassenger` programmatically (#F).
- Inside the test, we are getting the `passenger` bean from the context (#G) and we are comparing it with the one constructed programmatically (#H). The configuration of the `passenger` bean from the Spring context is the one from listing 16.4.

In order to make an accurate comparison between the `passenger` extracted from the container and the one constructed programmatically, John has overridden the `equals` and `hashCode` methods from the `Passenger` and `Country` classes (listings 16.8 and 16.9).

Listing 16.8 The overridden equals and hashCode methods from the Passenger class

```

public class Passenger {
    [...]
    @Override
    public boolean equals(Object o) {                                #A
        if (this == o) return true;                                  #A
        if (o == null || getClass() != o.getClass()) return false;  #A
        Passenger passenger = (Passenger) o;                         #A
        return name.equals(passenger.name) &&                      #A
               Objects.equals(country, passenger.country);          #A
    }

    @Override
    public int hashCode() {                                         #B
        return Objects.hash(name, country);                         #B
    }
}

```

In the listing above, we are doing the following:

- We are constructing the `Passenger` methods of `equals` (#A) and `hashCode` (#B) based on the `name` and `country` fields.

Listing 16.9 The overridden equals and hashCode methods from the Country class

```
public class Country {  
    [...]  
    @Override  
    public boolean equals(Object o) { #A  
        if (this == o) return true; #A  
        if (o == null || getClass() != o.getClass()) return false; #A  
  
        Country country = (Country) o; #A  
  
        if (codeName != null ? #A  
            !codeName.equals(country.codeName) : #A  
            country.codeName != null) return false; #A  
        if (name != null ? #A  
            !name.equals(country.name) : #A  
            country.name != null) return false; #A  
  
        return true; #A  
    } #A  
  
    @Override  
    public int hashCode() { #B  
        int result = 0; #B  
        result = 31 * result + (name != null ? name.hashCode() : 0); #B  
        result = 31 * result + (codeName != null ? #B  
            codeName.hashCode() : 0); #B  
        return result; #B  
    } #B  
}
```

In the listing above, we are doing the following:

- We are constructing the `Country` methods of `equals` (#A) and `hashCode` (#B) based on the `name` and `codeName` fields.

The expected passenger is constructed programmatically through the `getExpectedPassenger` method from the `PassengerUtil` class (listing 16.10).

Listing 16.10 The PassengerUtil class

```
public class PassengerUtil {  
  
    public static Passenger getExpectedPassenger() {  
        Passenger passenger = new Passenger("John Smith"); #A  
  
        Country country = new Country("USA", "US"); #B  
        passenger.setCountry(country); #C  
  
        return passenger; #D  
    } #D  
}
```

In the listings above, we are doing the following:

- Inside the `getExpectedPassenger` method, we are first creating the passenger "John Smith" (#A), we are creating the country "USA" (#B) and setting it as the

country of the passenger (#C). We are returning this passenger at the end of the method (#D).

Running the newly created `SimpleAppTest` will be successful, as shown in fig. 16.7. John has verified that the bean extracted from the Spring container is equal to the ones constructed programmatically.

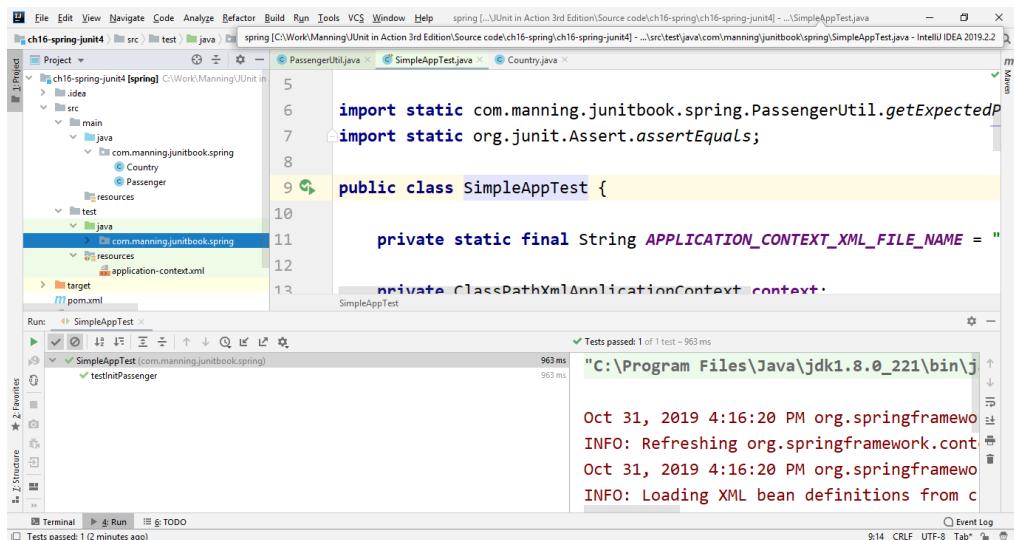


Figure 16.7 The result of running `SimpleAppTest`

16.3.2 Using the Spring TestContext framework

The Spring TestContext Framework offers unit and integration testing support independent of the testing framework in use, whether JUnit 3.x, JUnit 4.x, TestNG and so on. The TestContext framework uses convention over configuration: it provides default behavior that can be overridden through configuration.

For JUnit 4.5+, the TestContext framework also provides a custom runner.

John decided to refactor the initial test in order to use the capabilities of the TestContext framework. The result is shown in listing 16.11.

Listing 16.11 The `SpringAppTest` class

```
@RunWith(SpringJUnit4ClassRunner.class) #A
@ContextConfiguration("classpath:application-context.xml") #B
public class SpringAppTest {
    @Autowired #C
    private Passenger passenger; #C
    private Passenger expectedPassenger;
```

```

@Before
public void setUp() {
    expectedPassenger = getExpectedPassenger();
}

@Test
public void testInitPassenger() {
    assertEquals(expectedPassenger, passenger);
}

}

```

In the listing above, we are doing the following:

- We are running the test with the help of `SpringJUnit4ClassRunner` (#A). We are also annotating the test class to look for the context configuration into the `application-context.xml` file from the classpath (#B). These annotations belong to the `spring-test` dependency and serve to replace the programmatic creation of the context, as we have seen in listing 16.7. Through these annotations, we take care of the creation of the context from the `application-context.xml` file from the classpath and injecting the defined beans into our test.
- We are declaring a field of type `Passenger` and we are annotating it as `@Autowired` (#C). Spring will automatically look into the container and try to auto-wire the declared `passenger` field to a unique injected bean of the same type, declared into the container. It is important that there is one single bean of type `Passenger` inside the container, otherwise, there will be ambiguity and the test will fail to throw `UnsatisfiedDependencyException`.

The `SpringAppTest` is shorter than the `SimpleAppTest` one, as we have pushed even more control on the Spring Framework, by using the `@RunWith` and `@ContextConfiguration` annotations that help to automatically initialize the context.

The result of running the newly created `SimpleAppTest` and `SpringAppTest` will be successful, as shown in fig. 16.8. John has verified in two ways that the bean extracted from the Spring container is equal to the one constructed programmatically.

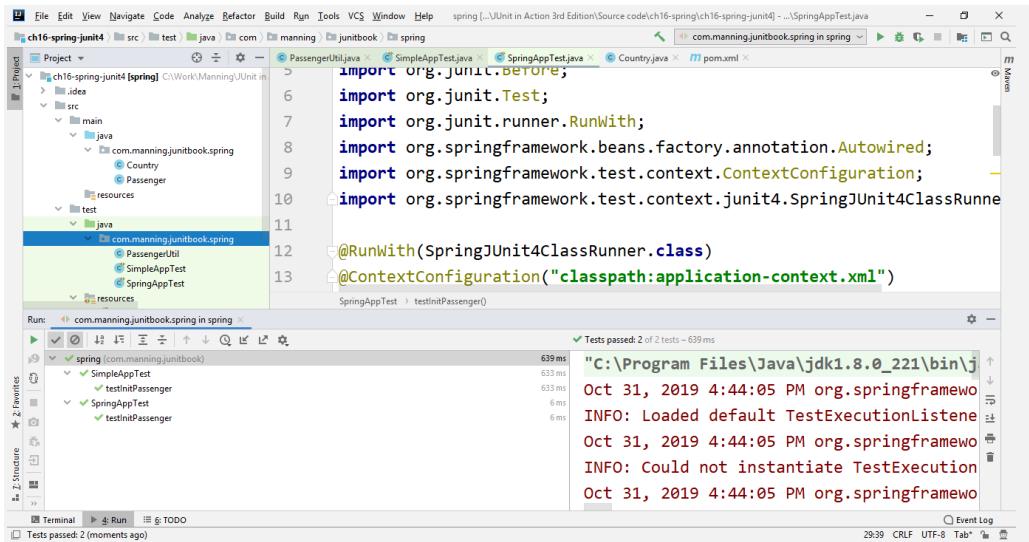


Figure 16.8 SimpleAppTest and SpringAppTest are successfully executed as Spring JUnit 4 tests

16.4 Using the SpringExtension for JUnit Jupiter

SpringExtension, introduced in Spring 5, is used to integrate Spring TestContext with JUnit 5 Jupiter Test. SpringExtension is used together with the JUnit 5 Jupiter @ExtendWith annotation.

Continuing his work on developing and testing the Spring flights management application, John will first migrate to JUnit 5, following the steps we presented in chapter 4. Then, he will continue to add new features to the application that will now be tested with JUnit 5.

The first thing that John must do is to replace the Spring 4 and JUnit 4 dependency from the pom.xml file with the Spring 5 and JUnit 5 ones.

Listing 16.12 The pom.xml file with the Spring 5 and JUnit 5 dependencies

```
<dependency> #A
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>5.2.0.RELEASE</version>
</dependency> #A
<dependency> #B
    <groupId>org.springframework</groupId>
    <artifactId>spring-test</artifactId>
    <version>5.2.0.RELEASE</version>
</dependency> #B
<dependency> #C
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-api</artifactId>
    <version>5.6.0</version>
    <scope>test</scope>

```

```

</dependency> #C
<dependency> #D
    <groupId>org.junit.jupiter</groupId> #D
    <artifactId>junit-jupiter-engine</artifactId> #D
    <version>5.6.0</version> #D
    <scope>test</scope> #D
</dependency> #D

```

In the listing above, we are doing the following:

- We introduce the `spring-context` (#A) and `spring-test` (#B) dependencies, version 5.2.0.RELEASE in both cases. `spring-context` is necessary to use the `@Autowired` annotation, while `spring-test` is necessary to use the `SpringExtension` and the `@ContextConfiguration` annotation. We have replaced the same dependencies but from version 4.2.5.RELEASE.
- We introduce the `junit-jupiter-api` (#C) and `junit-jupiter-engine` (#D) dependencies that we need to test our application with JUnit 5. We are replacing the previously used JUnit 4.12 dependency, as we have done for migration processes between JUnit 4 and JUnit 5 (see chapter 4).

John has then migrated the code from using Spring 4 and JUnit 4 to using Spring 5 and JUnit 5. The result is shown in listing 16.13.

Listing 16.13 The SpringAppTest class

```

@ExtendWith(SpringExtension.class) #A
@ContextConfiguration("classpath:application-context.xml") #B
public class SpringAppTest {

    @Autowired #C
    private Passenger passenger; #C
    private Passenger expectedPassenger;

    @BeforeEach #D
    public void setUp() {
        expectedPassenger = getExpectedPassenger();
    }

    @Test
    public void testInitPassenger() {
        assertEquals(expectedPassenger, passenger);
        System.out.println(passenger);
    }
}

```

In the listing above, we are doing the following:

- We are extending the test with the help of `SpringExtension` (#A). We are also annotating the test class to look for the context configuration into the `application-context.xml` file from the `classpath` (#B). These annotations belong to the `spring-test` dependency (version 5.2.0.RELEASE this time) and serve to replace the work with runners of JUnit 4. Through these annotations, we take care of the creation of the

context from the `application-context.xml` file from the classpath and injecting the defined beans into our test. For more details about JUnit 5 extensions, please refer to chapter 14.

- We are keeping the field of type `Passenger` and annotating it with `@Autowired` (#C). Spring will automatically look into the container and try to auto-wire the declared `passenger` field to a unique injected bean of the same type, declared into the container. It is important that there is one single bean of type `Passenger` inside the container, otherwise, there will be ambiguity and the test will fail to throw `UnsatisfiedDependencyException`.
- We have replaced the JUnit 4 `@Before` annotation with the JUnit 5 `@BeforeEach` one. This is no particularity of Spring applications, but it relates to the usual migration from JUnit 4 to JUnit 5 (see chapter 4).

The result of running the JUnit 5 `SpringAppTest` will be successful, as shown in fig. 16.9. John has verified the correct migration from working with Spring 4 and JUnit 4 to Spring 5 and JUnit 5. Now, he is ready to continue his work and add new features to the Spring flights management application.

```

File Edit View Navigate Code Analyze Refactor Build Run Tools VCS Window Help
spring (...) -> SpringAppTest.java
Project src com Manning JUnit in Action 3rd Edition Source code ch16-spring/ch16-spring-junit5
src main java com.manning.junitbook.spring Country Passenger
resources test java com.manning.junitbook.spring PassengerUtil SpringAppTest
resources application-context.xml
target SpringAppTest
Run: SpringAppTest
Test Results: 1 test passed: 1 of 1 test - 100 ms
Nov 01, 2019 2:59:19 PM org.springframework.context.support.DefaultApplicationContext
INFO: Loaded default TestExecutionListener
Nov 01, 2019 2:59:19 PM org.springframework.context.support.DefaultApplicationContext
INFO: Using TestExecutionListeners: [org.junit.jupiter.engine.execution.JupiterTestExecutionListener]

```

Figure 16.9 `SpringAppTest` is successfully executed as Spring JUnit 5 test

16.5 Adding a new feature and testing it with JUnit 5

We follow John in his developer activity of adding a new feature in the current Spring 5 flights management application and testing it with JUnit 5. In addition to what we have already presented, we will demonstrate some essential capabilities of the Spring Framework (creating

beans through annotations, classes implementing essential interfaces, implementing the observer pattern with the help of Spring 5 defined events and listeners). For a gentle introduction, we have started with an XML-based configuration. Be aware that the Spring Framework is a large topic and we are mostly discussing the things related to testing these applications with JUnit 5. For a comprehensive book on this topic, also showing in detail the configuration possibilities, please refer to the Manning "Spring in Action" by Craig Walls (<https://www.manning.com/books/spring-in-action-fifth-edition>).

The feature that John received requires tracking the registration of passengers through the registration manager of the flight management system. To be properly organized, the customer requires that whenever a new passenger is registered, the registrations manager must answer with a confirmation. This functionality follows the idea of the Observer pattern (fig. 16.10), which we are going to examine and implement.

OBSERVER pattern

The Observer pattern is a design pattern in which a subject maintains a list of dependents (observers or listeners). The subject will notify the observers about events on its side, events about which the observers are interested.

Such an observer may be attached to the side of the subject. Once attached, the observer will receive notifications about the events that are of interest to it. The events are announced by the subject and, as a consequence, the observer will execute an update of its state.

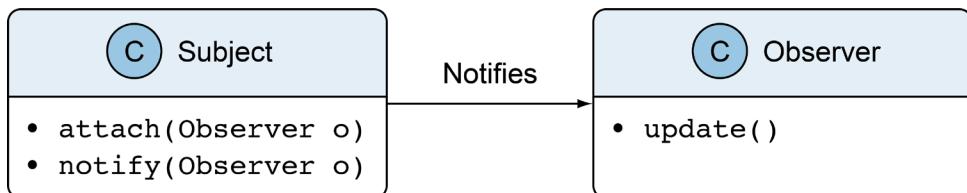


Figure 16.10 The subject notifies the observer about events of interest to it

In our case, we identify the following components of the system:

- The registration manager is the Subject. It is where the event takes place and what needs to inform the Observers.
- The registration itself is the event that is generated on the registration manager side. This event must be broadcasted to the Observers (listeners) that are interested.
- The registration listener will be the Observer that will receive the registration event and that will confirm the registration by setting the passenger as registered inside the system and by printing a message.

The functionality of this concrete registration system is shown in figure 16.11.

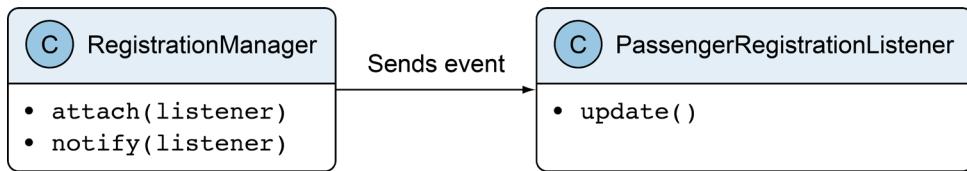


Figure 16.11 The registration manager informs the listener about registration events

John will start working on this feature by changing the `Person` class to be aware of the registration status (listing 16.14).

Listing 16.14 The modified Person class

```

public class Passenger {
    [...]
    private boolean isRegistered; #A

    [...]
    public boolean isRegistered() {
        return isRegistered;
    }

    public void setIsRegistered(boolean isRegistered) {
        this.isRegistered = isRegistered;
    }

    @Override
    public String toString() {
        return "Passenger{" +
            "name='" + name + '\'' +
            ", country='" + country +
            ", registered=" + isRegistered +
            '}';
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Passenger passenger = (Passenger) o;
        return isRegistered == passenger.isRegistered && #C
            Objects.equals(name, passenger.name) &&
            Objects.equals(country, passenger.country);
    }

    @Override
    public int hashCode() {
        return Objects.hash(name, country, isRegistered); #D
    }
}

```

In the listing above, we are doing the following:

- We have added the `isRegistered` field to the `Passenger` class, together with the

getter and setter for it (#A).

- We have modified the `toString` (#B), `equals` (#C) and `hashCode` (#D) methods to also take into consideration this newly introduced `isRegistered` field.

John has also modified the `application-context.xml` file (listing 16.15). Declaring data beans into the XML configuration file is suitable for testing purposes, as it is our case now. The XML is easy to change, the code does not need to be recompiled and you may have different configurations in different environments, without touching the code.

Listing 16. 15 The modified application-context.xml

```
<bean id="passenger" class="com.manning.junitbook.spring.Passenger">
    <constructor-arg name="name" value="John Smith"/>
    <property name="country" ref="country"/>
    <property name="isRegistered" value="false"/> #A
</bean>

<context:component-scan base-package="com.manning.junitbook.spring" /> $B
```

In the listing above, we are modifying the following to the previous version:

- We have added the `isRegistered` field to the initialization of the `Passenger` bean (#A).
- We have inserted the directive to require Spring to also scan the package `com.manning.junitbook.spring` to find components (#B). This will mean that, besides the beans defined inside the XML, there will be other ones defined inside the code from the indicated base package, through annotations.

John is creating the `PassengerRegistrationEvent` class, to define the custom event that is happening inside the registration system (listing 16.16).

Listing 16. 16 The PassengerRegistrationEvent class

```
public class PassengerRegistrationEvent extends ApplicationEvent { #A

    private Passenger passenger; #B

    public PassengerRegistrationEvent(Passenger passenger) {
        super(passenger); #C
        this.passenger = passenger; #C
    }

    public Passenger getPassenger() {
        return passenger; #B
    }

    public void setPassenger(Passenger passenger) {
        this.passenger = passenger; #B
    } #B
}
```

In the listing above, we are doing the following:

- We are defining the PassengerRegistrationEvent class that extends ApplicationEvent. ApplicationEvent is the Spring abstract class to be extended by all application events.
- We are keeping a reference to the Passenger object whose registration has generated the event, together with a getter and a setter on it (#B).
- Into the PassengerRegistrationEvent we are calling the constructor of the super-class ApplicationEvent that receives as argument the source of the event - the passenger itself (#C).

We have mentioned that there are alternative ways to create Spring beans and we have already used the XML way, which is more suitable for data beans that may change for different executions and for different environments. We'll now show how to create beans using annotations, which is more suitable for functional beans that generally do not change.

John is creating the RegistrationManager class, to serve as events generator (listing 16.17).

Listing 16. 17 The RegistrationManager class

```
@Service #A
public class RegistrationManager implements ApplicationContextAware { #B
    private ApplicationContext applicationContext; #C

    public ApplicationContext getApplicationContext() { #C
        return applicationContext; #C
    } #C

    @Override #D
    public void setApplicationContext(ApplicationContext applicationContext) throws BeansException { #D
        this.applicationContext = applicationContext; #D
    } #D

}
```

In the listing above, we are doing the following:

- We are annotating the class with the @Service annotation (#A). This means that Spring will automatically create a bean of the type of this class. Remember that, into the application-context.xml file, we have inserted the component-scan directive, with the base-package="com.manning.junitbook.spring", meaning that Spring will look for annotation-defined beans in that package.
- The RegistrationManager class implements the ApplicationContextAware interface (#B). This will mean that RegistrationManager will have a reference to the application context that it will use to publish events.
- Inside the class, we keep a reference to the application context and a getter for it (#C).
- The setApplicationContext method inherited from ApplicationContextAware is initializing the field applicationContext as a reference to the applicationContext injected by Spring as an argument of this method (#D).

John is creating the PassengerRegistrationListener class, to serve as an observer of the passenger registration events (listing 16.18).

Listing 16. 18 The PassengerRegistrationListener class

```
@Service #A
public class PassengerRegistrationListener {

    @EventListener #B
    public void confirmRegistration(PassengerRegistrationEvent #B
                                    passengerRegistrationEvent) { #B
        passengerRegistrationEvent.getPassenger().setIsRegistered(true); #C
        System.out.println("Confirming the registration for the passenger: " #C
                           + passengerRegistrationEvent.getPassenger()); #C
    }
}
```

In the listing above, we are doing the following:

- We are annotating the class with the `@Service` annotation (#A). This means that Spring will automatically create a bean of the type of this class. Remember that, into the `application-context.xml` file, we have inserted the `component-scan` directive, with the `base-package="com.manning.junitbook.spring"`, meaning that Spring will look for annotation-defined beans into that package.
- The `confirmRegistration` method receives as argument a `PassengerRegistrationEvent` and is annotated with `@EventListener` (#B). This will mean that Spring will automatically register this method as a listener (observer) for the `PassengerRegistrationEvent` type of events. Whenever such an event will occur, this method will be executed.
- Inside the method, once we receive the passenger registration event, we are confirming the registration by setting the passenger as registered inside the system and by printing a message (#C).

John is finally creating the RegistrationTest class, to verify the behavior of the code that he has implemented (listing 16.19).

Listing 16. 19 The RegistrationTest class

```
@ExtendWith(SpringExtension.class) #A
@ContextConfiguration("classpath:application-context.xml") #B
public class RegistrationTest {

    @Autowired #C
    private Passenger passenger; #C

    @Autowired #C
    private RegistrationManager registrationManager; #C

    @Test
    public void testPersonRegistration() {
        registrationManager.getApplicationContext() #D
            .publishEvent(new PassengerRegistrationEvent(passenger)); #D
    }
}
```

```
    assertTrue(passenger.isRegistered()); #E
}
}
```

In the listing above, we are doing the following:

- We are extending the test with the help of SpringExtension (#A). We are also annotating the test class to look for the context configuration into the application-context.xml file from the classpath (#B). Through these annotations, we take care of the creation of the context from the application-context.xml file from the classpath and injecting the defined beans into our test.
- We are declaring a field of type Passenger and a field of type RegistrationManager and we are annotating them as @Autowired (#C). Spring will automatically look into the container and try to auto-wire the declared passenger and RegistrationManager fields to unique injected beans of the same type, declared into the container. It is important that there is one single bean of each of these types inside the container, otherwise, there will be ambiguity and the test will fail to throw UnsatisfiedDependencyException. We have declared a Passenger bean into the application-context.xml file. And we have annotated RegistrationManager with the @Service annotation, meaning that Spring will automatically create a bean of the type of this class.
- Into the test method, we are using the RegistrationManager field to publish an event of type PassengerRegistrationEvent with the help of the reference to the application context that it is holding (#D). Then, we are checking that the registration status of the passenger has really changed (#E).

The result of running the newly created RegistrationTest will be successful, as shown in fig. 16.12. John has successfully implemented and tested the passengers' registration feature using the capabilities of JUnit 5 and Spring 5.

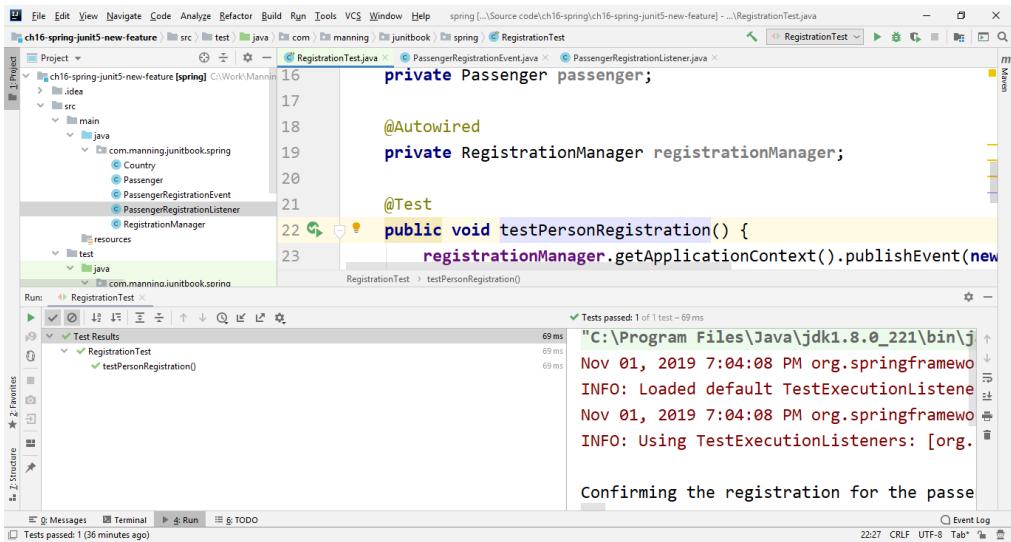


Figure 16.12 RegistrationTest successfully executes as Spring JUnit 5 test

In the next chapter, we will start building and testing software using Spring Boot, one of the extensions of the Spring Framework. Spring Boot eliminates the boilerplate configurations required for setting up a Spring application.

16.6 Summary

This chapter has covered the following:

- Introducing the Spring Framework, one of the widely used Java frameworks nowadays. Spring is a lightweight but at the same time flexible and universal framework used for creating Java applications.
- Demonstrating the pattern that is the base of Spring, Dependency Injection (Inversion of Control): objects are inserted into a container that injects the dependencies when it creates the object.
- Introducing beans, the objects that form the backbone of a Spring application and that are instantiated, assembled and managed by the Spring Inversion of Control container.
- Executing the first steps to using and testing a Spring application both by creating the Spring context programmatically and by using the Spring TestContext framework.
- Demonstrating the usage of the `SpringExtension` for JUnit Jupiter to create the first JUnit 5 tests for a Spring application.
- Developing a new feature of the Spring application using beans creation through annotations and implementing the observer pattern with the help of Spring 5 defined events and listeners and testing it with JUnit 5.

17

Testing Spring Boot applications

This chapter covers:

- Introducing Spring Boot
- Creating a project with Spring Initializr
- Moving a Spring application tested with JUnit 5 to Spring Boot
- Implementing test specific configuration for Spring Boot
- Developing a new feature into a Spring Boot application and testing it with JUnit 5

Working with Spring Boot is like pair-programming with the Spring developers.

- Anonymous

Spring Boot is a Spring convention-over-configuration solution. It enables the creation of Spring applications ready to immediately run. Spring Boot is an extension of the Spring framework which strongly reduces the initial Spring applications configurations. A Spring Boot application is already preconfigured and is given dependencies to outside libraries so that we can start using it. Most Spring Boot applications need very little or even no Spring configuration.

CONVENTION-OVER-CONFIGURATION

Convention over configuration is a software design principle adopted by different frameworks in order to reduce the number of configuration actions that a programmer using that framework needs to make. This means that the programmer only needs to specify the non-standard configuration of the application.

17.1 Introducing Spring Boot

A shortcoming of the Spring Framework is that it takes some time to make the application ready for production. The configuration is time-consuming and can be a little bit overwhelming for the new developers.

Spring Boot is built on the top of the Spring Framework as an extension of it. It is strongly supporting developers through a convention-over-configuration approach that can help the rapid creation of an application. As most Spring Boot applications need little Spring configuration, developers can focus on the business logic instead of infrastructure and configuration. The business logic is that part of the program focusing on the business rules that determine the effective way the application works.

Among the Spring Boot features, we enumerate:

- Create stand-alone Spring applications
- Automatically configure Spring when possible
- Embed web servers like Tomcat or Jetty directly (no need to deploy WAR files)
- Provide preconfigured Maven POMs (Project Object Models)

17.2 Creating a project with Spring Initializr

In the Spring Boot spirit of supporting developers through the convention-over-configuration approach and supporting the rapid creation of an application, we will use Spring Initializr in order to generate the project with just what we need to start quickly.

To do this, we'll access the <https://start.spring.io/> website and we'll insert there the configuration data of our new project. Spring Boot will generate a skeleton of the application, where we'll transfer the business logic of tracking the registration of passengers through the registration manager of the flight management system, as we have implemented it in chapter 16.

We see in fig. 17.1 that we have chosen to create a new Java project, managed by Maven as build tool, the group name will be `com.manning.junitbook` and the artifact id will be `spring-boot`.

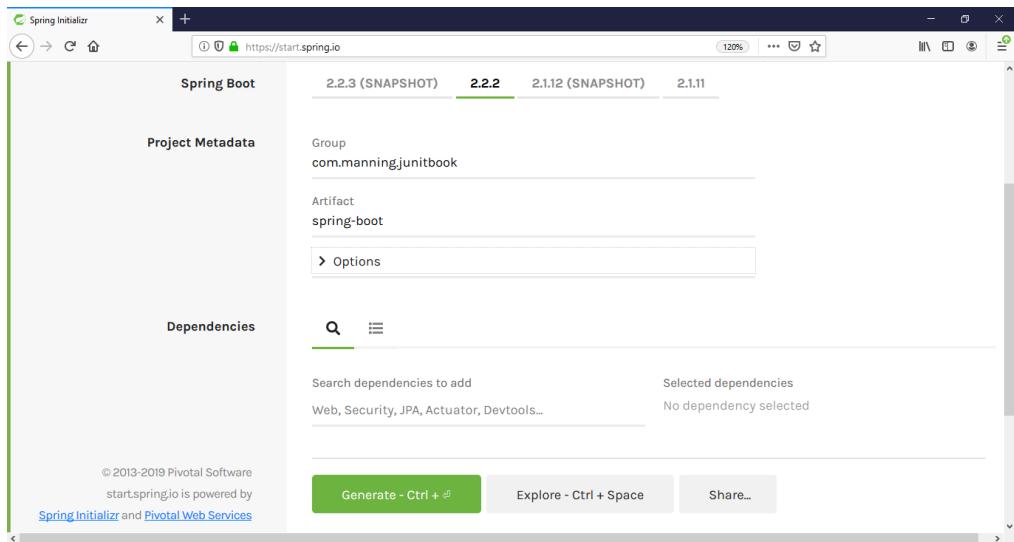


Figure 17.1 Creating a new Spring Boot project with the help of Spring Initializr – the project is managed by Maven as the build tool, the group name is com.manning.junitbook, the artifact id is spring-boot

In fig. 17.2 we are providing some more configuration to the newly created project. We leave the description as “Demo project for Spring Boot” and the packaging as .jar. We notice that we are provided the possibility to add new dependencies (Web, Security, JPA, Actuator, Devtools). We are not adding anything for the moment, as our first goal will be to move the Spring application from chapter 16 to Spring Boot, but you should be aware of this possibility: with a few clicks, you will be provided the needed dependencies directly into the Maven pom.xml file.

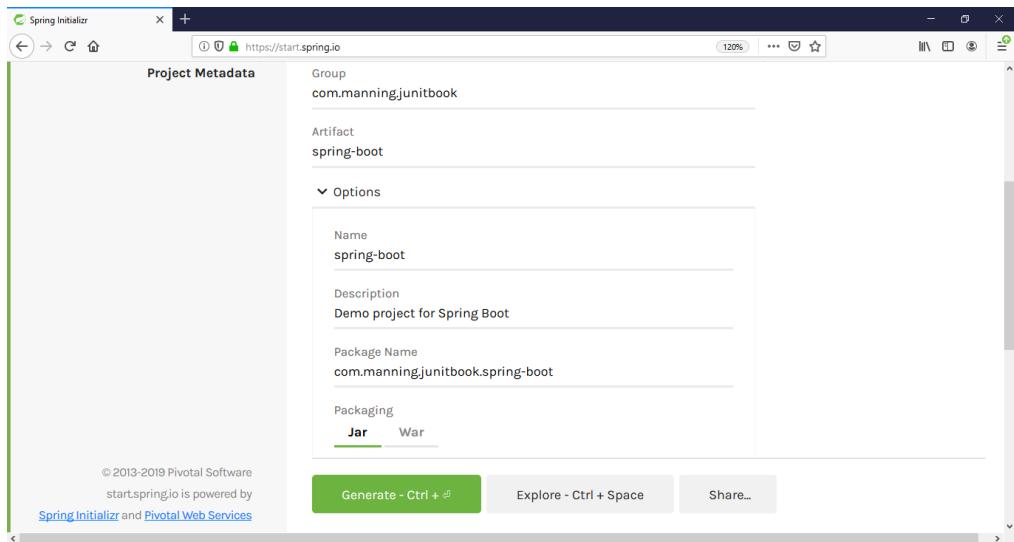


Figure 17.2 Available options when creating a new Spring Boot project with Spring Initializr: the description is “Demo project for Spring Boot”, the packaging is .jar. We may add new dependencies (Web, Security, JPA, Actuator, Devtools)

If we click now on the “Explore –Ctrl + Space” button, we’ll see a few details about what will be generated: the structure of the project and the content of the Maven `pom.xml` file.



Figure 17.3 The structure of the project (expanded folders) and the content of the Maven `pom.xml` file (including the `spring-boot-starter` dependency)

We can press here the “Download the ZIP” link or, on the previous screen, the “Generate – Ctrl +” button, and we’ll get the archive containing the project.

Opening this generated project, we see its structure containing the Maven `pom.xml` file, a main method into the `Application` class, a test into the `ApplicationTests` class, and an `application.properties` as resource file (empty for now, but we'll use it later). So, a few clicks, choices made within a few seconds, and we have a new Spring application. We can simply run the tests (fig. 17.4), then we'll focus on moving our previously created Spring application from chapter 16 to this new Spring Boot structure.

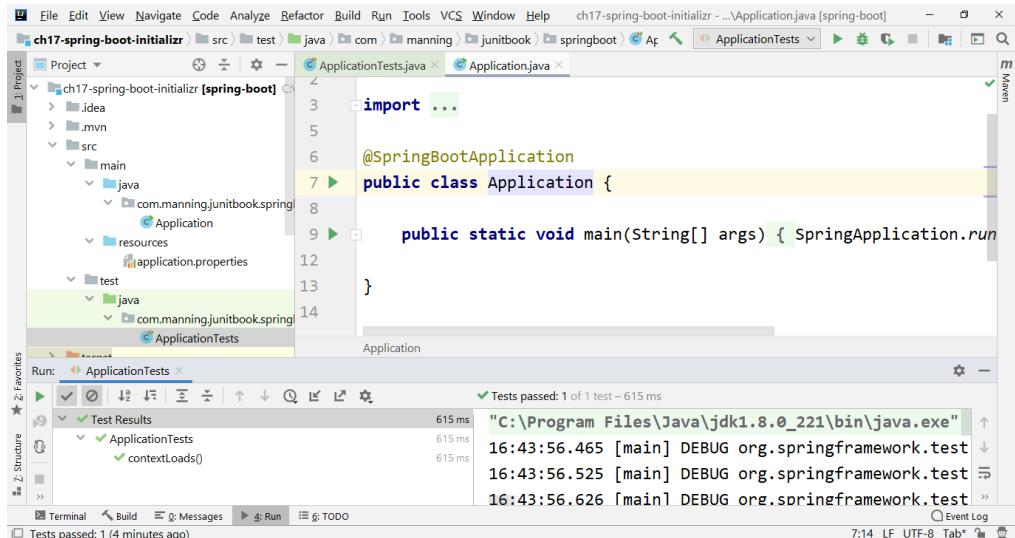


Figure 17.4 Loading the generated Spring Boot application into IntelliJ IDEA and successfully running the test that has been automatically provided

In our demonstration, we'll use different ways of configuring the Spring container. XML is still the traditional way for configurations, although recent years show some tendency to work more and more with annotations. However, for our testing purposes, XML provides the advantage of being less intrusive – the classes do not need additional dependencies and the configuration can be quickly changed for data beans meant for testing, without recompiling the code. This idea may be beneficial for testers who have less technical knowledge. Anyway, you'll encounter the different possibilities of Spring configuration across this book.

17.3 Moving the Spring application to Spring Boot

We'll start moving our previously developed Spring application to the skeleton provided by Spring Boot. For a successfully systematic and organized migration, we'll do the following:

- We'll introduce two new packages into the original application: first, `com.manning.junitbook.springboot.model` where we'll keep the domain classes, `Passenger` and `Country`; and a second package,

`com.manning.junitbook.springboot.registration`, where we'll keep the classes related to the registration events: `PassengerRegistrationEvent`, `PassengerRegistrationListener`, and `RegistrationManager`. The new structure is displayed in fig. 17.5.

- The `application-context.xml` file will no longer contain the directive `<context:component-scan base-package="..." />`. We'll provide an equivalent of this, using Spring annotations at the level of the test. The updated `application-context.xml` file will look as in listing 17.1.
- We'll introduce new annotations into the `RegistrationTest` file, to replace the directive that we have eliminated from the `application-context.xml` file (listing 17.2).
- We'll keep only the `RegistrationTest` class for testing our application, as the `SpringAppTest` class was only testing the behavior of a single passenger. We'll also adapt `RegistrationTest` to match the structure of the `ApplicationTests` class initially provided by Spring Boot.

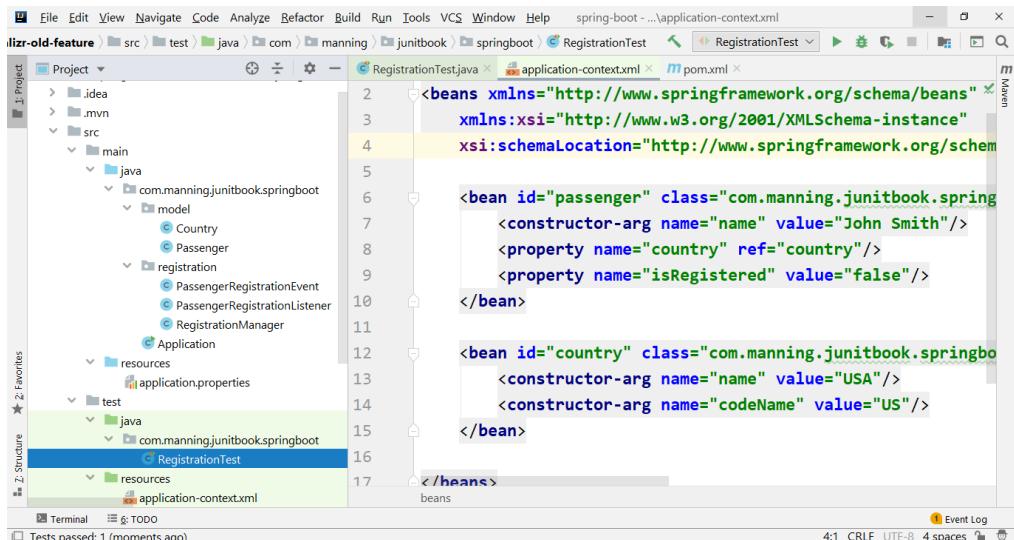


Figure 17.5 The new structure of the Spring Boot registration application – the classes have been distributed to two different packages

Listing 17.1 The application-context.xml configuration file

```

<bean id="passenger"
      class="com.manning.junitbook.springboot.model.Passenger">#A
        <constructor-arg name="name" value="John Smith"/>#A
        <property name="country" ref="country"/>#A
        <property name="isRegistered" value="false"/>#A
</bean>

```

```

<bean id="country"                                     #B
      class="com.manning.junitbook.springboot.model.Country"> #B
      <constructor-arg name="name" value="USA"/>                 #B
      <constructor-arg name="codeName" value="US"/>             #B
</bean>                                              #B

```

The updated application-context.xml file from listing 17.1 keeps the definition of the passenger and country beans, but removes the original directive <context:component-scan base-package="..." />. We'll provide an equivalent of this, using Spring annotations at the level of the test.

Listing 17.2 The rewritten Spring Boot RegistrationTest

```

@SpringBootTest                                         #A
@EnableAutoConfiguration                                #B
@ImportResource("classpath:application-context.xml")   #C
class RegistrationTest {

    [...]

    @Autowired
    private RegistrationManager registrationManager;     #D
    [...]

}

```

In the listing above, we see the following:

- We are annotating our class with the @SpringBootTest annotation (#A), initially provided by the test that has been generated by Spring Boot. This annotation provides a few features, from which we are now using the one that, together with @EnableAutoConfiguration (#B), it searches in the current package of the test class and in its sub-packages for beans definitions. This way, it will be able to discover and auto-wire the RegistrationManager bean (#D).
- There are still beans defined at the level of the XML configuration, which we are importing with the help of the @ImportResource annotation (#C). XML still remains a traditional way to configure Spring applications, so it is largely in use. The recent years show some tendency to adopt more annotations and the Java-based configuration, so we are going to demonstrate all possibilities.
- The rest of the original RegistrationTest class remains the same as the Spring Core application built in chapter 16.

Running the current test will be successful, as shown in fig. 17.6.

The screenshot shows the IntelliJ IDEA interface. The top navigation bar includes File, Edit, View, Navigate, Code, Analyze, Refactor, Build, Run, Tools, VCS, Window, Help, and spring-boot - ...RegistrationTest.java. The left sidebar shows the Project structure for 'ch17-spring-boot-initializr-old-feature [spring-boot]'. The main editor window displays the code for 'RegistrationTest.java':

```

14
15     @SpringBootTest
16     @EnableAutoConfiguration
17     @ImportResource("classpath:application-context.xml")
18     class RegistrationTest {
19
20         @Autowired
21         private Passenger passenger;
22

```

The 'Run' tool bar at the bottom has 'RegistrationTest' selected. The 'Test Results' section shows one test passed:

- Test Results: 686 ms
 - RegistrationTest: 686 ms
 - testPersonRegistration(): 686 ms

The 'Console' tab shows the command line output:

```

C:\Program Files\Java\jdk1.8.0_221\bin\java.exe" 
17:03:27.994 [main] DEBUG org.springframework.test
17:03:28.055 [main] DEBUG org.springframework.test
17:03:28.145 [main] DEBUG org.springframework.test
17:03:28.226 [main] INFO org.springframework.boot.

```

Figure 17.6 Successfully running the RegistrationTest,- the registration person functionality works fine after moving to the application to Spring Boot

17.4 Implementing test specific configuration for Spring Boot

So far, we are using for this chapter a Spring Boot application, including the business logic of the original Spring Core application. The configuration of the beans is still at the level of the XML file. XML is the traditional way for Spring configuration, as it is non-intrusive (the classes do not need external dependencies), and it can be easily changed (the source files do not need to be recompiled). As the country and passenger data beans are for testing purposes, keeping them in an XML file will enable different testers with less programming skills to quickly create their own configurations.

We have considered useful to show the different configuration alternatives for such an application. We are focusing on how to test Spring applications. For comprehensive works on the topic, you may use other Manning books: "Spring in Action" (<https://www.manning.com/books/spring-in-action-fifth-edition>) and "Spring Boot in Action" (<https://www.manning.com/books/spring-boot-in-action>), both authored by Craig Walls.

As the developers at Tested Data Systems have moved to Spring Boot, they would like now a different way of configuration, closer to the Spring Boot spirit. Then, they would like to add more business logic. Mike is in charge of these tasks and the first thing he decides to do is to introduce the test-specific configuration capabilities of Spring Boot.

Mike will replace the initial definition of beans from the application-context.xml file with the help of the Spring Boot @TestConfiguration annotation. The @TestConfiguration annotation can be used to define additional beans or customizations for a test. In Spring Boot, the beans configured in a top-level class annotated with @TestConfiguration must be

explicitly registered in the class that contains the tests. The previously existing test data beans from the application-context.xml will be moved to this @TestConfiguration annotated class, while the functional beans (RegistrationManager and PassengerRegistrationListener from the code base) are declared through the @Service annotation.

So, Mike will introduce the following test configuration class instead of the application-context.xml file (listing 17.3).

Listing 17.3 The TestBeans class

```
@TestConfiguration  
public class TestBeans {  
  
    @Bean  
    Passenger createPassenger() {  
        Passenger passenger = new Passenger("John Smith");  
        passenger.setCountry(createCountry());  
        passenger.setIsRegistered(false);  
        return passenger;  
    }  
  
    @Bean  
    Country createCountry() {  
        Country country = new Country("USA", "US");  
        return country;  
    }  
}
```

In the listing above we are doing the following:

- We are introducing the new class TestBeans that will replace the previously existing application-context.xml and we are annotating it with @TestConfiguration (#A). Its purpose is to provide the beans while executing the tests.
- We are writing the createPassenger method (#C) and we are annotating it with @Bean (#B). Its purpose is to create and configure a Passenger bean that will be injected into the tests.
- We are writing the createCountry method (#E) and we are annotating it with @Bean (#D). Its purpose is to create and configure a Country bean that will be injected into the tests.

Mike will modify the previously existing test in order to use the @TestConfiguration annotated class instead of the application-context.xml file for the creation and configuration of the bean (listing 17.4). He will import the configuration defined within the TestBeans class by using the @Import annotation (#A).

Listing 17.4 The modified RegistrationTest class

```
@SpringBootTest  
@Import(TestBeans.class)  
class RegistrationTest {  
}
```

```

@.Autowired
private Passenger passenger;

@Autowired
private RegistrationManager registrationManager;

@Test
void testPersonRegistration() {
    registrationManager.getApplicationContext()
        .publishEvent(new PassengerRegistrationEvent(passenger));
    System.out.println("After registering:");
    System.out.println(passenger);
    assertTrue(passenger.isRegistered());
}
}

```

The change that is propagated into the test class is the replacement of the previously existing `@EnableAutoConfiguration` and `@ImportResource` annotations with the `@Import(TestBeans.class)` one. This way, he will explicitly register the beans defined in `TestBeans` into the class that contains the tests. The advantage of making the step to using the `@TestConfiguration` annotation is not only the fact that this approach is more specific to Spring Boot or that we are replacing two annotations with only one. Working with Java-based defined beans will be *type-safe*: Java prevents errors and the compiler will report issues if we are configuring a wrong way. Search and navigation is much simpler, as we take advantage of the IDE usage. Working with full Java code, you will also get a helping hand for refactoring, code completion and finding references in the code.

Running the test in its new format will be successful, as shown in fig. 17.7. We also notice the removal of the previously existing `application-context.xml` configuration file.

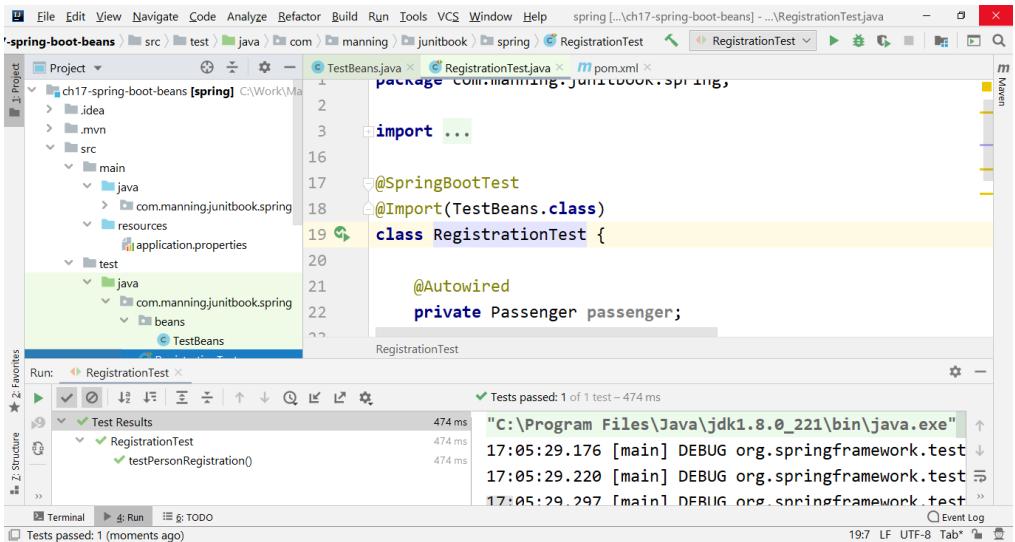


Figure 17.7 Successfully running the RegistrationTest, – the registration person functionality works fine after implementing test specific configuration for Spring Boot

17.5 Introducing and testing a new feature into the Spring Boot application

Mike receives a requirement for a new feature to be introduced. The application must be able to create and set up flights, add and remove passengers to the flights. Mike will introduce a new class to model the reality: Flight. He must check that a list of passengers can be correctly registered on a flight and that all persons from the list will receive the registration confirmation. First, he has to introduce the Flight class (listing 17.5).

Listing 17.5 The Flight class

```
public class Flight {

    private String flightNumber;
    private int seats;
    private Set<Passenger> passengers = new HashSet<>();

    public Flight(String flightNumber, int seats) {
        this.flightNumber = flightNumber;
        this.seats = seats;
    }

    public String getFlightNumber() {
        return flightNumber;
    }

    public int getSeats() {
        return seats;
    }
}
```

```

}

public Set<Passenger> getPassengers() { #C
    return Collections.unmodifiableSet(passengers);
}

public boolean addPassenger(Passenger passenger) { #D
    if(passengers.size() >= seats) {
        throw new RuntimeException(
            "Cannot add more passengers than the capacity of the flight!");
    }
    return passengers.add(passenger);
}

public boolean removePassenger(Passenger passenger) { #E
    return passengers.remove(passenger);
}

@Override
public String toString() { #F
    return "Flight " + getFlightNumber();
}
}

```

In listing 17.5, we are implementing the Flight class, doing the following:

1. We are providing three fields, `flightNumber`, `seats` and `passengers`, to describe a Flight (#A).
2. We provide a constructor having the first two mentioned parameters (#B).
3. We are additionally providing three getters (#C) addressing the fields that we have defined. The getter of the `passengers` field will return an unmodifiable list of passengers so that the list cannot be changed from outside after being returned by the getter.
4. The `addPassenger` method will add a passenger to the flight. It also compares the number of passengers with the number of seats so the flight will not be overbooked. (#D).
5. We provide the possibility to remove a passenger from a flight (#E).
6. We finally override the `toString` method (#F).

We are describing the list of passengers of the flight through the CSV in listing 17.6. A passenger is described through the name and the country code. There are 20 passengers in total.

Listing 17.6 The flights_information.csv file

```

John Smith; UK
Jane Underwood; AU
James Perkins; US
Mary Calderon; US
Noah Graves; UK
Jake Chavez; AU
Oliver Aguilar; US

```

```

Emma McCann; AU
Margaret Knight; US
Amelia Curry; UK
Jack Vaughn; US
Liam Lewis; AU
Olivia Reyes; US
Samantha Poole; AU
Patricia Jordan; UK
Robert Sherman; US
Mason Burton; AU
Harry Christensen; UK
Jennifer Mills; US
Sophia Graham; UK

```

Mike will implement the `FlightUtilBuilder` class, which is parsing the CSV file and is populating the flight with the corresponding passengers. The information is brought from an external file to the memory of the application (listing 17.7).

Listing 17.7 The FlightUtilBuilder class

```

@TestConfiguration
public class FlightBuilder { #A

    private static Map<String, Country> countriesMap = new HashMap<>(); #B

    static {
        countriesMap.put("AU", new Country("Australia", "AU")); #C
        countriesMap.put("US", new Country("USA", "US")); #C
        countriesMap.put("UK", new Country("United Kingdom", "UK")); #C
    } #C

    @Bean
    Flight buildFlightFromCsv() throws IOException { #D
        Flight flight = new Flight("AA1234", 20); #E
        try(BufferedReader reader = new BufferedReader(new FileReader(
            "src/test/resources/flights_information.csv"))){ #F
            String line = null; #G
            do { #H
                line = reader.readLine(); #H
                if (line != null) { #H
                    String[] passengerString = line.toString().split(":"); #I
                    Passenger passenger = new Passenger( #J
                        passengerString[0].trim()); #J
                    passenger.setCountry( #K
                        countriesMap.get(passengerString[1].trim())); #K
                    passenger.setIsRegistered(false); #L
                    flight.addPassenger(passenger); #M
                }
            } while (line != null); #H
        }
        return flight; #N
    }
}

```

In the listing above, we are doing the following:

- We are creating the `FlightBuilder` class that will parse the CSV file and construct the list of flights. The class is annotated with the previously introduced annotation `@TestConfiguration` (#A). Thus, it signals that it defines the beans needed for a test.
- We are defining the countries map (#B) and we are populating it with three countries (#C). The key of the map is the country code, the value is its name.
- We are creating the `buildFlightFromCsv` method and annotating it with `@Bean` (#D). Its purpose is to create and configure a `Flight` bean that will be injected into the tests.
- We are creating a flight with the help of the constructor (#E), then we are constructing it by parsing the CSV file (#F).
- We are initializing the `line` variable with `null` (#G), then we are parsing the CSV file and reading line by line (#H).
- The line is split by the ; separator (#I), we are creating a passenger with the help of the constructor having as argument the first part of the line, before the separator (#J). We set the country of the passenger by taking from the countries map the value corresponding to the country code included in the second part of the line, after the separator (#K).
- We are setting the passenger as not registered (#L), adding him to the flight (#M) and, after finishing parsing all the lines from the CSV file we are returning the fully configured flight (#N).

Mike will finally implement the `FlightTest` class, which is registering all passengers from a flight and checks that all of them are confirmed (listing 17.8).

Listing 17.8 The FlightTest class

```

@SpringBootTest                                     #A
@Import(FlightBuilder.class)                      #A
public class FlightTest {                         #A

    @Autowired                                         #B
    private Flight flight;                           #B

    @Autowired                                         #C
    private RegistrationManager registrationManager; #C

    @Test                                              #D
    void testFlightPassengersRegistration() {        #D
        for (Passenger passenger : flight.getPassengers()) { #E
            assertFalse(passenger.isRegistered());       #F
            registrationManager.getApplicationContext()   #G
                .publishEvent(new PassengerRegistrationEvent(passenger)); #G
        }
        for (Passenger passenger : flight.getPassengers()) { #H
            assertTrue(passenger.isRegistered());       #H
        }
    }
}

```

```
}
```

In the listing above, we are doing the following:

- We are creating the `FlightTest` class, annotating it with `@SpringBootTest` and importing the beans from the `FlightBuilder` class (#A). We remind that `@SpringBootTest` searches in the current package of the test class and in its sub-packages for beans definitions. This way, it will be able to discover and auto-wire the `RegistrationManager` bean (#C).
- We are auto-wiring the `Flight` bean injected by the `FlightBuilder` class (#B).
- We are creating the `testFlightPassengersRegistration` method and annotating it with `@Test` (#D). Inside it, we are browsing all passengers from the injected `Flight` bean (#E) and first checking that they are not registered (#F). Then we are using the `RegistrationManager` field to publish an event of type `PassengerRegistrationEvent` with the help of the reference to the application context that it is holding (#G).
- Finally, we are browsing all passengers from the injected `Flight` bean and checking that they are now registered (#H).

Running `FlightTest` will be successful, as shown in fig. 17.8.

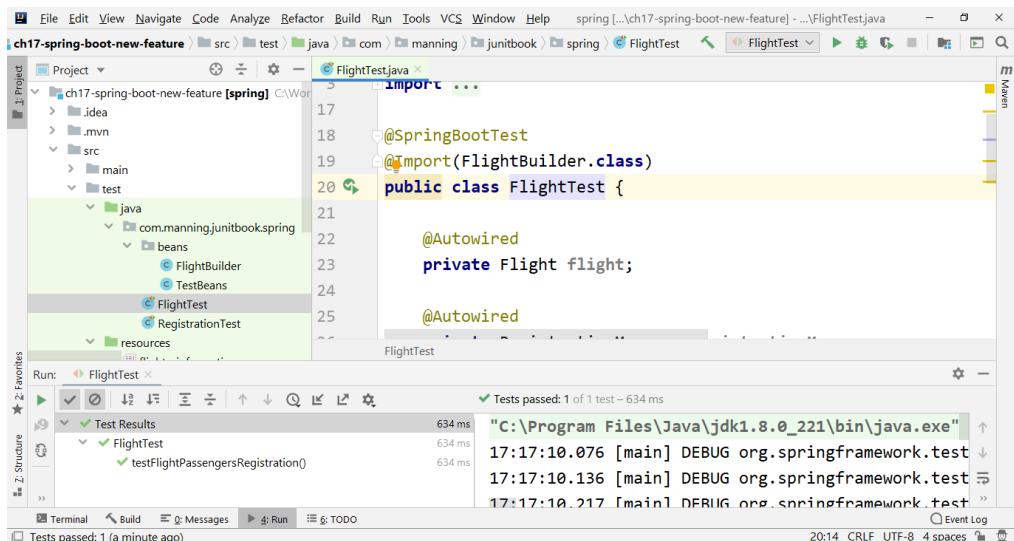


Figure 17.8 Successfully running the `FlightTest`, that checks the registration of all passengers from a flight

In the next chapter, we will start building and testing a REST (Representational Transfer) application with the help of Spring Boot.

17.6 Summary

This chapter has covered the following:

- Introducing Spring Boot as the Spring convention-over-configuration solution for creating Spring-based applications that you can simply run.
- Creating a ready to run Spring Boot project with Spring Initializr, the web application that helps to generate a Spring Boot project structure.
- Moving our previously developed Spring application tested with JUnit 5 to the skeleton provided by Spring Boot.
- Implementing test specific configuration for Spring Boot with the help of the `@TestConfiguration` annotation and with Java-based defined beans and taking benefit of the Spring Boot convention over configuration approach.
- Developing a new feature to create and set up flights, add and remove passengers to the flights, integrating it into a Spring Boot application and testing it with JUnit 5.

18

Testing a REST API

This chapter covers:

- Introducing REST applications
- Creating a RESTful API to manage one entity
- Creating a RESTful API to manage two related entities
- Testing the RESTful API managing two related entities

For now, let's just say that if your API is re-defining the HTTP verbs or if it is assigning new meanings to HTTP status codes or making up its own status codes, it is not RESTful.

George Reese, The REST API Design Handbook

REST (Representational state transfer) is a software architectural style for creating Web services, that also provides a set of constraints. Web services following this REST architectural style are called RESTful Web services. RESTful Web services allow interoperability between the Internet and computer systems. The requesting systems will be able to access and manipulate Web resources represented as text using a well-known set of stateless operations.

18.1 Introducing REST applications

The American computer scientist Roy Fielding has first defined REST, presenting the REST principles in his Ph.D. dissertation in 2000. Roy Fielding is also one of the authors of the HTTP specification.

We will first define the terms of *client* and *resource* in order to shape what makes an API RESTful.

- The client may be a person or software, using the RESTful API. A programmer using a

RESTful API to execute actions against the LinkedIn web site is, in fact, such a client. The client can also be a web browser. When we go to the LinkedIn website, our browser is the client that calls the website API and that displays then the obtained information on the screen.

- The resource can be any object the API can obtain information about. In LinkedIn API, a resource can be a message, a photo, or a user. Each resource has a unique identifier.

The REST architecture style defines six constraints:

- Client-server
- Stateless
- Uniform interface
- Layered system
- Cacheable
- Code on demand (optional)

CLIENT-SERVER

Clients are separated from servers, each one with its own concern:

- A client is concerned with the representation for the user.
- A server is concerned with data storage and domain model logic – the conceptual model of a domain including data and behavior.

STATELESS

- The server does not keep any information about the client between requests.
- Each request from a client contains all of the information necessary to respond to that request. The client keeps the state on its side.

UNIFORM INTERFACE

- The client and the server may evolve independently of each other. The uniform interface between them makes them loosely coupled.

LAYERED SYSTEMS

- The client does not have any way to determine if it interacting directly with the server or with an intermediary.
- Layers can be dynamically added and removed. They may provide security, load balancing, shared caching.

CACHEABLE

- Clients are able to cache responses. Responses define themselves as cacheable or not.

CODE ON DEMAND

- Servers are able to temporarily customize or extend the functionality of a client. The server can transfer to the client some logic that the client can execute: JavaScript client-side scripts, Java applets.

A RESTful web application provides information about its resources. Resources are identified with the help of URLs (Uniform Resource Locators). The client can execute actions against such a resource: read a resource, create a new resource, modify or delete an existing resource.

The REST architectural style is not protocol-specific, but the most widely used is REST over HTTP. HTTP is a synchronous application network protocol, based on requests and responses.

To make our API RESTful, we have to follow a set of rules developing it. A RESTful API will transfer to the client that uses it as a representation of the state of the accessed resource. For example, when calling the LinkedIn API to access a specific user, the API will return the state of that user (name, biography, professional experience, posts). The REST set of rules will make the API easier to understand and to start being used by new programmers joining a team.

The representation of the state can be in a JSON, XML or HTML format.

The client uses the API to send to the server:

- The identifier (URL - Uniform Resource Locator) of the resource we want to access.
- The operation we want the server to perform on that resource. This is an HTTP method.
The common HTTP methods are GET, POST, PUT, PATCH, and DELETE.

For example, fetching a specific LinkedIn user, using LinkedIn RESTful API, will require a URL that identifies that user and the HTTP method GET.

18.2 Creating a RESTful API to manage one entity

The flights management application developed at Tested Data Systems, as we have left it at the end of chapter 17, allows the registration of a list of passengers to a flight and testing the confirmation of this registration.

Mike, the programmer involved in developing the application, receives a new feature to implement. He has to create a REST API to manage the passengers of the flight.

The first thing that Mike will implement is to create the REST API that will deal with the countries from the application.

For the purpose of creating a REST API, Mike will add new dependencies to the Maven pom.xml configuration file (listing 18.1).

Listing 18.1 The newly added dependencies into the pom.xml configuration file

```
<dependency> #A  
  <groupId>org.springframework.boot</groupId> #A  
  <artifactId>spring-boot-starter-web</artifactId> #A  
</dependency> #A
```

```

<dependency> #B
    <groupId>org.springframework.boot</groupId> #B
        <artifactId>spring-boot-starter-data-jpa</artifactId> #B
</dependency> #B
<dependency> #C
    <groupId>com.h2database</groupId> #C
        <artifactId>h2</artifactId> #C
</dependency> #C

```

The updated pom.xml file from listing 18.1 adds the following new dependencies:

- spring-boot-starter-web is the dependency for building web, including RESTful, applications using Spring Boot (#A).
- As we need to use Spring and JPA (Java Persistence API) for persisting information, we include the spring-boot-starter-data-jpa dependency (#B).
- H2 is an open-source lightweight Java database that we'll embed in our Java application to persist the information to it. Consequently, we need to add the h2 dependency (#C).

As the application will be a functional RESTful one, allowing it to be used for accessing, creating, modifying and deleting the information about the list of passengers, Mike will move the FlightBuilder class from the test folder into the main folder. He will also extend a little its functionality, in order to also provide access to the management of countries (listing 18.2).

Listing 18.2 The FlightBuilder class

```

public class FlightBuilder {

    private Map<String, Country> countriesMap = new HashMap<>(); #A

    public FlightBuilder() throws IOException #B
    {
        try(BufferedReader reader = new BufferedReader(new
            FileReader("src/main/resources/countries_information.csv"))){ #C
            String line = null;
            do { #D
                line = reader.readLine(); #E
                if (line != null) { #E
                    String[] countriesString = line.toString().split(";");
                    Country country = new Country(countriesString[0].trim(), #F
                        countriesString[1].trim()); #G
                    countriesMap.put(countriesString[1].trim(), country); #H
                }
            } while (line != null); #E
        }
    }

    @Bean #I
    Map<String, Country> getCountriesMap() { #I
        return Collections.unmodifiableMap(countriesMap); #I
    } #I
}

```

```

@Bean
public Flight buildFlightFromCsv() throws IOException {
    Flight flight = new Flight("AA1234", 20);
    try(BufferedReader reader = new BufferedReader(new
        FileReader("src/main/resources/flights_information.csv"))) {
        String line = null;
        do {
            line = reader.readLine();
            if (line != null) {
                String[] passengerString = line.toString().split(";");
                Passenger passenger = new
                    Passenger(passengerString[0].trim());
                passenger.setCountry(
                    countriesMap.get(passengerString[1].trim()));
                passenger.setIsRegistered(false);
                flight.addPassenger(passenger);
            }
        } while (line != null);
    }
    return flight;
}

```

In the listing above, we are doing the following:

- We are defining the countries map and we are populating it with three countries (#A).
- We are populating the countries map inside the constructor (#B), by parsing the CSV file (#C).
- We are initializing the line variable with `null` (#D), then we are parsing the CSV file and reading line by line (#E).
- The line is split by the ; separator (#F), we are creating a passenger with the help of the constructor having as argument the first part of the line, before the separator (#G).
- We are adding the parsed information to the `countriesMap` (#H).
- We are creating the `getCountriesMap` method and annotating it with `@Bean` (#I). Its purpose is to create and configure a Map bean that will be injected into the application.
- We are creating the `buildFlightFromCsv` method and annotating it with `@Bean` (#J). Its purpose is to create and configure a Flight bean that will be injected into the application.
- We are creating a flight with the help of the constructor (#K), then we are constructing it by parsing the CSV file (#L).
- We are initializing the line variable with `null` (#M), then we are parsing the CSV file and reading line by line (#N).
- The line is split by the ; separator (#O), we are creating a passenger with the help of the constructor having as argument the first part of the line, before the separator (#P). We set the country of the passenger by taking from the countries map the value corresponding to the country code included in the second part of the line, after the

separator (#Q).

- We are setting the passenger as not registered (#R), adding him to the flight (#S) and, after finishing parsing all the lines from the CSV file we are returning the fully configured flight (#T).

The content of the countries_information.csv file used for building the countries map is shown in listing 18.3.

Listing 18.3 The countries_information.csv file

```
Australia; AU  
USA; US  
United Kingdom; UK
```

Mike would like to start the RESTful application on the custom 8081 port, in order to avoid possible port conflicts. Spring Boot allows the externalization of the configuration so that different persons can work with the same application code in different environments. Various properties can be specified inside the application.properties file, from which only server.port is now needed to be set as 8081 (listing 18.4).

Listing 18.4 application.properties file

```
server.port=8081
```

In the listing above we are doing the following:

- We are introducing the new class TestBeans that will replace the previously existing application-context.xml and we are annotating it with @TestConfiguration (#A). Its purpose is to provide the beans while executing the tests.

Mike will modify the Country class in order to make it a model component of the RESTful application. The model is one of the components of the MVC (Model-View-Controller) pattern. It is the application dynamic data structure, independent of the user interface. It directly manages the data of the application.

MVC (model-view-controller)

MVC is a software design pattern used for creating a program that accesses data through user interfaces. The program is split into three related parts: Model, View, and Controller. Thus, the inner representation of the information is separated from its presentation on the side of the user and the system parts are loosely coupled.

The changes are shown in listing 18.5.

Listing 18.5 The modified Country class

```
@Entity #A  
public class Country {  
    @Id #B
```

```

private String codeName; #B
private String name;

// avoid "No default constructor for entity"
public Country() { #C
}
[...] #C
}

```

In listing 18.5, we are changing the `Country` class, doing the following:

- We are annotating it with `@Entity`, meaning that it has the ability to represent objects that can be persisted (#A).
- We are annotating the `codeName` field with the `@Id` annotation (#B). This means that the `codeName` field is a primary key, it uniquely identifies an entity that is persisted.
- We are adding a default constructor to the `Country` class (#C). Every class annotated with `@Entity` needs a default constructor, as the persistence layer will use it to create a new instance of this class through reflection. The default constructor is no longer provided by the compiler as we have created another one.

Mike will then create the `CountryRepository` interface, which extends `JpaRepository` (listing 18.6).

Listing 18.6 The CountryRepository interface

```
public interface CountryRepository extends JpaRepository<Country, Long> { }
```

Defining this `CountryRepository` interface serves two purposes. First, by extending `JpaRepository` we get a bunch of generic CRUD (Create, Read, Update and Delete) methods into our `CountryRepository` type that allows working with `Country` objects. Second, this will allow Spring to scan the classpath for this interface and create a Spring bean for it.

CRUD (create, read, update, delete)

CRUD (Create, Read, Update, Delete) are the four basic functions of persistence. Besides REST, they are also in large use in database applications and in user interfaces.

Mike will then write a controller for the `CountryRepository`. A controller is responsible for controlling the application logic and acts as the coordinator between the view (the way data is displayed to the user) and the model (the data) (listing 18.7).

Listing 18.7 The CountryController class

```
@RestController #A
public class CountryController { #A
```

```

@.Autowired
private CountryRepository repository; #B
#B

@GetMapping("/countries")
List<Country> findAll() { #C
    return repository.findAll(); #C
}

}

```

In listing 18.7, we are implementing the `CountryController` class, doing the following:

- We are creating the `CountryController` class and annotating it with `@RestController` (#A). The `@RestController` annotation has been introduced in Spring 4.0 to simplify the creation of RESTful web services. It marks the class as a controller, and also eliminates the need to annotate every of its request handling methods with the `@ResponseBody` annotation, as it needed to be before Spring 4.0.
- We are declaring a `CountryRepository` field and auto-wiring it (#B). As `CountryRepository` extends `JpaRepository`, Spring will scan the classpath for this interface, create a Spring bean for it and auto-wire it here.
- We are creating the `findAll` method and annotating it with `@GetMapping("/countries")` (#C). This `@GetMapping` annotation maps the HTTP GET requests to the `"/countries"` URL onto the specific handler method. As we are using the `@RestController` annotation on the class itself, we do not need to annotate the response object of the method with `@ResponseBody`. That was necessary before Spring 4.0 when we also had to use `@Controller` instead of `@RestController` on the class itself.
- The `findAll` method will return the result of executing `repository.findAll()` (#D). `repository.findAll()` is one of the CRUD (Create, Read, Update and Delete) methods that are automatically generated by the fact that `CountryRepository` extends `JpaRepository`. As its name says, it will return all objects from the repository.

Mike will now change the Application class (listing 18.8).

Listing 18.8 The modified Application class

```

@SpringBootApplication
@Import(FlightBuilder.class)
public class Application { #A

    @Autowired
    private Map<String, Country> countriesMap; #B
    #B

    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }

    @Bean
    CommandLineRunner configureRepository #C
    #C

```

```

        (CountryRepository countryRepository) { #C
    return args -> {
        for (Country country: countriesMap.values()) { #D
            countryRepository.save(country); #D
        }
    };
}

}

```

In listing 18.8 we have made the following changes to the Application class:

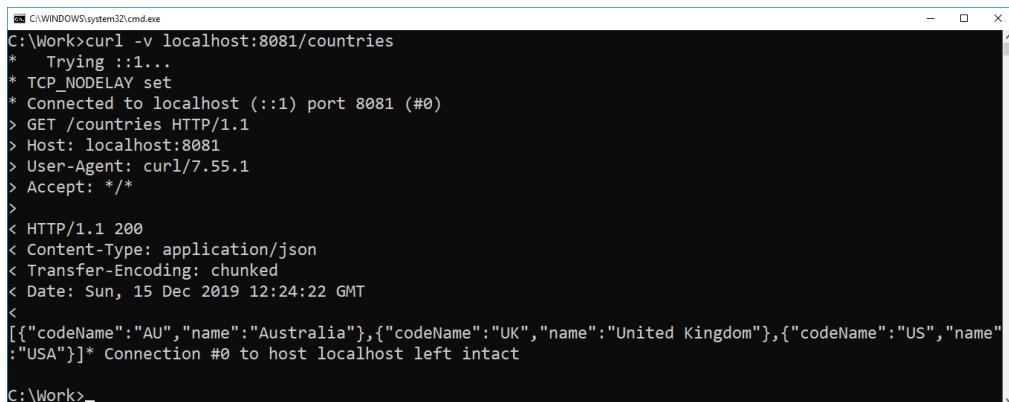
- We are importing FlightBuilder, which created the countriesMap bean (#A), and we are auto-wiring it (#B).
- We are creating a bean of type CommandLineRunner (#C). CommandLineRunner is a Spring Boot functional interface (an interface with one single method) that gives access to application arguments as a string array. The created bean will browse through all countries in the countriesMap and save them into the countryRepository (#D). This CommandLineRunner interface gets created and its single method executed just before the method run() from SpringApplication completes.

Mike will now launch Application into execution. What the RESTful application is providing so far is only the access to the “/countries” endpoint through the GET method. An endpoint is a resource that can be referenced and to which client messages can be addressed.

We can test the only REST API endpoint using the curl program. curl is a command-line tool that is transferring data using various protocols, including HTTP. curl stands for Client URL. We will simply execute the following command:

```
curl -v localhost:8081/countries
```

As the application is running on the 8081 port and “/countries” is the only available endpoint. The result of the execution is shown in fig. 18.1 – it lists the countries in JSON format.



```
C:\Work>curl -v localhost:8081/countries
*   Trying ::1...
* TCP_NODELAY set
* Connected to localhost (::1) port 8081 (#0)
> GET /countries HTTP/1.1
> Host: localhost:8081
> User-Agent: curl/7.55.1
> Accept: */*
>
< HTTP/1.1 200
< Content-Type: application/json
< Transfer-Encoding: chunked
< Date: Sun, 15 Dec 2019 12:24:22 GMT
<
[{"codeName": "AU", "name": "Australia"}, {"codeName": "UK", "name": "United Kingdom"}, {"codeName": "US", "name": "USA"}]* Connection #0 to host localhost left intact
C:\Work>
```

Figure 18.1 The result of running the `curl -v localhost:8081/countries` command is the list of created countries

The access to this endpoint can also be tested through the browser, by simply going to `localhost:8081/countries`. The result is also provided in JSON format and can be seen in fig. 18.2.

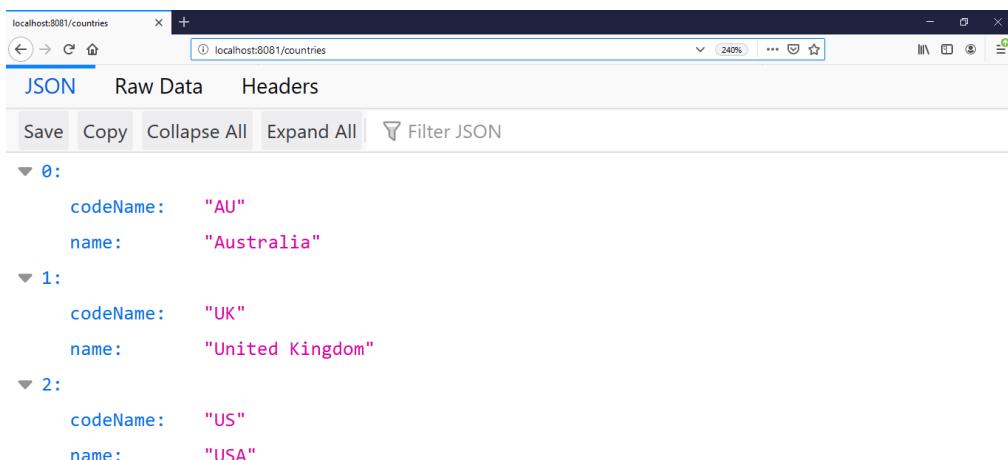


Figure 18.2 The result of accessing the `localhost:8081/countries` URL from the browser is the list of created countries

18.3 Creating a RESTful API to manage two related entities

Mike will continue the implementation and will modify the `Passenger` class in order to also make it a model component of the RESTful application. The changes are shown in listing 18.9.

18.9 The modified Passenger class

```
@Entity #A
public class Passenger {

    @Id #B
    @GeneratedValue #B
    private Long id; #B
    private String name;
    @ManyToOne #C
    private Country country;
    private boolean isRegistered;

    // avoid "No default constructor for entity"
    public Passenger() { #D
    }
    [...]
}
```

In listing 18.9, we are changing the `Passenger` class, doing the following:

- We are annotating it with `@Entity`, meaning that it has the ability to represent objects that can be persisted (#A).
- We are introducing a new `Long` field, `id`, and annotating it with the `@Id` and `@GeneratedValue` annotations (#B). This means that the `id` field is a primary key, it uniquely identifies an entity that is persisted, and that its value will be automatically generated for that field by the persistence layer.
- We are annotating the `country` field with the `@ManyToOne` annotation (#C). This means that many passengers are mapped to one single country.
- We are adding a default constructor to the `Passenger` class (#D). Every class annotated with `@Entity` needs a default constructor, as the persistence layer will use it to create a new instance of this class through reflection.

Mike will then create the `PassengerRepository` interface, which extends `JpaRepository` (listing 18.10).

Listing 18.10 The PassengerRepository interface

```
public interface PassengerRepository extends JpaRepository<Country, Long> { }
```

Defining this `PassengerRepository` interface serves two purposes. First, by extending `JpaRepository` we get a bunch of generic CRUD (Create, Read, Update and Delete) methods into our `PassengerRepository` type that allows working with `Passenger`

objects. Second, this will allow Spring to scan the classpath for this interface and create a Spring bean for it.

Mike will write a custom exception to be thrown when a passenger is not found (listing 18.11).

Listing 18.11 The PassengerNotFoundException class

```
public class PassengerNotFoundException extends RuntimeException { #A  
    public PassengerNotFoundException(Long id) { #B  
        super("Passenger id not found : " + id); #B  
    } #B  
}
```

In listing 18.11 we are declaring the `PassengerNotFoundException` by extending `RuntimeException` (#A) and we are creating a constructor that receives an `id` as a parameter (#B).

Mike will then write a controller for the `PassengerRepository`. A controller is responsible for controlling the application logic and acts as the coordinator between the view (the way data is displayed to the user) and the model (the data itself) (listing 18.12).

Listing 18.12 The PassengerController class

```
@RestController  
public class PassengerController { #A  
  
    @Autowired  
    private PassengerRepository repository; #B  
  
    @Autowired  
    private Map<String, Country> countriesMap; #C  
  
    @GetMapping("/passengers")  
    List<Passenger> findAll() { #D  
        return repository.findAll();  
    } #D  
  
    @PostMapping("/passengers")  
    @ResponseStatus(HttpStatus.CREATED) #E  
    Passenger createPassenger(@RequestBody Passenger passenger) { #F  
        return repository.save(passenger); #G  
    } #E  
  
    @GetMapping("/passengers/{id}")  
    Passenger findPassenger(@PathVariable Long id) { #H  
        return repository.findById(id) #I  
            .orElseThrow(() -> new PassengerNotFoundException(id)); #H  
    } #H  
  
    @PatchMapping("/passengers/{id}")  
    Passenger patchPassenger(@RequestBody Map<String, String> updates, #J  
        @PathVariable Long id) { #K  
  
        return repository.findById(id) #L
```

```

.map(passenger -> {
    #L
    String name = updates.get("name");
    #M
    if (null!= name) {
        #M
        passenger.setName(name);
        #M
    }

    Country country =
        #N
        countriesMap.get(updates.get("country"));
    #N
    if (null != country) {
        #N
        passenger.setCountry(country);
        #N
    }

    String isRegistered = updates.get("isRegistered");
    #O
    if(null != isRegistered) {
        #O
        passenger.setIsRegistered(
            #O
            isRegistered.equalsIgnoreCase("true")? true: false);
        #O
    }
    #P
    return repository.save(passenger);
}
.orElseGet(() -> {
    #Q
    throw new PassengerNotFoundException(id);
})
);
}

@DeleteMapping("/passengers/{id}")
void deletePassenger(@PathVariable Long id) {
    #S
    repository.deleteById(id);
}
}
}

```

In listing 18.12, we are implementing the PassengerController class, doing the following:

- We are creating the PassengerController class and annotating it with @RestController (#A). The @RestController annotation has been introduced in Spring 4.0 to simplify the creation of RESTful web services. It marks the class a controller, and also eliminates the need to annotate all of its request handling methods with the @ResponseBody annotation, as was needed before Spring 4.0.
- We are declaring a PassengerRepository field and auto-wiring it (#B). As PassengerRepository extends JpaRepository, Spring will scan the classpath for this interface, create a Spring bean for it and auto-wire it here. We are also declaring a countriesMap field and auto-wiring it (#C).
- We are creating the findAll method and annotating it with @GetMapping("/passengers") (#D). This @GetMapping annotation maps the HTTP GET requests to the "/passengers" URL onto the specific handler method. As we are using the @RestController annotation on the class itself, we do not need to annotate the response object of the method with @ResponseBody. That was necessary before Spring 4.0 when we also had to use @Controller instead of @RestController on the class itself.
- We are declaring the createPassenger method and annotating it with

@PostMapping("/passengers") (#E). `@PostMapping` annotated methods handle the HTTP POST requests matched with given URL expression. We mark that method with `@ResponseStatus`, specifying the response status as `HttpStatus.CREATED` (#F). The `@RequestBody` annotation maps the `HttpRequest` body to the annotated domain object. The `HttpRequest` body is deserialized to a Java object (#G).

- We are declaring the `findPassenger` method that will look for the passenger by `id` and annotating it with `@GetMapping("/passengers/{id}")` (#H). This `@GetMapping` annotation maps the HTTP GET requests to the `"/passengers/{id}"` URL onto the specific handler method. It will search for the passenger into the repository and return him or, in case the passenger does not exist, will throw the custom declared `PassengerNotFoundException`. The `id` argument of the method is annotated with `@PathVariable`, meaning that it will be extracted from the `{id}` value of the URL (#I).
- We are declaring the `patchPassenger` method and annotating it with `@PatchMapping("/passengers/{id}")` (#J). This `@PatchMapping` annotation maps the HTTP PATCH requests to the `"/passengers/{id}"` URL onto the specific handler method. We are annotating the `updates` parameter with `@RequestBody` and the `id` parameter with `@PathVariable` (#K). The `@RequestBody` annotation maps the `HttpRequest` body to the annotated domain object. The `HttpRequest` body is deserialized to a Java object. The `id` argument of the method, annotated with `@PathVariable`, will be extracted from the `{id}` value of the URL.
- We search by the input `id` into the repository (#L). We change the name of an existing passenger (#M), the country (#N) and the registration status (#O), in case any of these ones have been requested into the `updates`. The modified passenger is saved into the repository (#P).
- In case the passenger does not exist, we throw the custom declared `PassengerNotFoundException` (#Q).
- We are declaring the `delete` method and annotating it with `@DeleteMapping("/passengers/{id}")` (#R). This `@DeleteMapping` annotation maps the HTTP DELETE requests to the `"/passengers/{id}"` URL onto the specific handler method. It will delete the passenger from the repository. The `id` argument of the method is annotated with `@PathVariable`, meaning that it will be extracted from the `{id}` value of the URL (#S).

Mike will now change the Application class (listing 18.13).

Listing 18.13 The modified Application class

```
@SpringBootApplication  
@Import(FlightBuilder.class)  
public class Application {  
  
    @Autowired  
    private Flight flight; #A  
    #A
```

```

@.Autowired
private Map<String, Country> countriesMap;

public static void main(String[] args) {
    SpringApplication.run(Application.class, args);
}

@Bean
CommandLineRunner configureRepository
    (CountryRepository countryRepository, #B
     PassengerRepository passengerRepository) { #B
    #B
    return args -> {

        for (Country country: countriesMap.values()) { #B
            countryRepository.save(country);
        }

        for (Passenger passenger : flight.getPassengers()) { #C
            passengerRepository.save(passenger); #C
        }
    };
}
}

```

In listing 18.13, we have made the following changes to the Application class:

- We are auto-wiring the `flight` bean (#A), imported from the `FlightBuilder`.
- We are modifying the bean of type `CommandLineRunner` by adding a new parameter of type `PassengerRepository` to the `configureRepository` method that is creating it (#B). `CommandLineRunner` is a Spring Boot interface that provides access to application arguments as a string array. The created bean will additionally browse through all passengers in the `flight` and save them into the `passengerRepository` (#C). This `CommandLineRunner` interface gets created and its method executed just before the method `run()` from `SpringApplication` completes.

The list of passengers of the flight is described through the CSV in listing 18.14. A passenger is described through the name and the country code and there are 20 passengers in total.

Listing 18.14 The flights_information.csv file

```

John Smith; UK
Jane Underwood; AU
James Perkins; US
Mary Calderon; US
Noah Graves; UK
Jake Chavez; AU
Oliver Aguilar; US
Emma McCann; AU
Margaret Knight; US
Amelia Curry; UK
Jack Vaughn; US
Liam Lewis; AU
Olivia Reyes; US

```

```
Samantha Poole; AU
Patricia Jordan; UK
Robert Sherman; US
Mason Burton; AU
Harry Christensen; UK
Jennifer Mills; US
Sophia Graham; UK
```

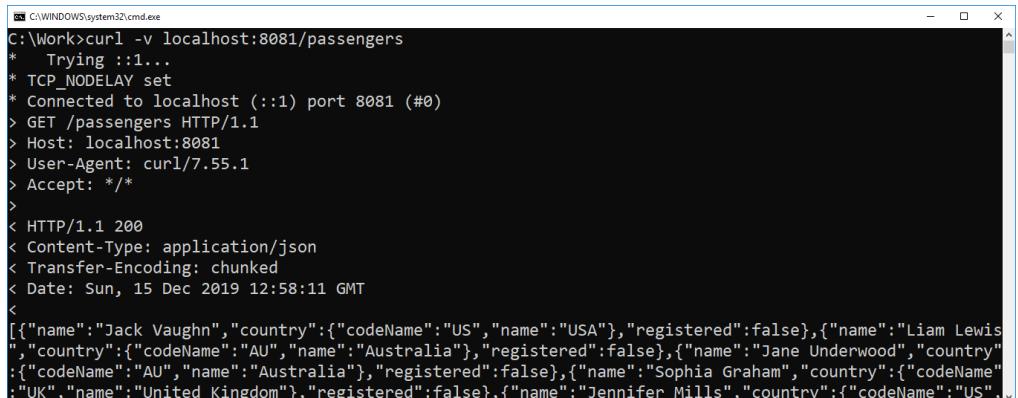
When the application is started, the `FlightBuilder` class will parse the file, create the flight with the list of passengers and inject it into the application. The application will browse the list and save each passenger into the repository.

Mike will now launch Application into execution. The RESTful application is now also providing access to the “/passengers” endpoint. An endpoint is a resource that can be referenced and to which client messages can be addressed.

We can test the new functionalities of the REST API endpoint using the `curl` program. We will execute the following command:

```
curl -v localhost:8081/passengers
```

As the application is running on the 8081 port and “/passengers” is now available as an endpoint. The result of the execution is shown in fig. 18.3 – it lists the passengers in JSON format.



```
C:\Work>curl -v localhost:8081/passengers
*   Trying ::1...
* TCP_NODELAY set
* Connected to localhost (::1) port 8081 (#0)
> GET /passengers HTTP/1.1
> Host: localhost:8081
> User-Agent: curl/7.55.1
> Accept: /*

< HTTP/1.1 200
< Content-Type: application/json
< Transfer-Encoding: chunked
< Date: Sun, 15 Dec 2019 12:58:11 GMT
<

[{"name": "Jack Vaughn", "country": {"codeName": "US", "name": "USA"}, "registered": false}, {"name": "Liam Lewis", "country": {"codeName": "AU", "name": "Australia"}, "registered": false}, {"name": "Jane Underwood", "country": {"codeName": "AU", "name": "Australia"}, "registered": false}, {"name": "Sophia Graham", "country": {"codeName": "UK", "name": "United Kingdom"}, "registered": false}, {"name": "Jennifer Mills", "country": {"codeName": "US", "name": "United States"}, "registered": false}]
```

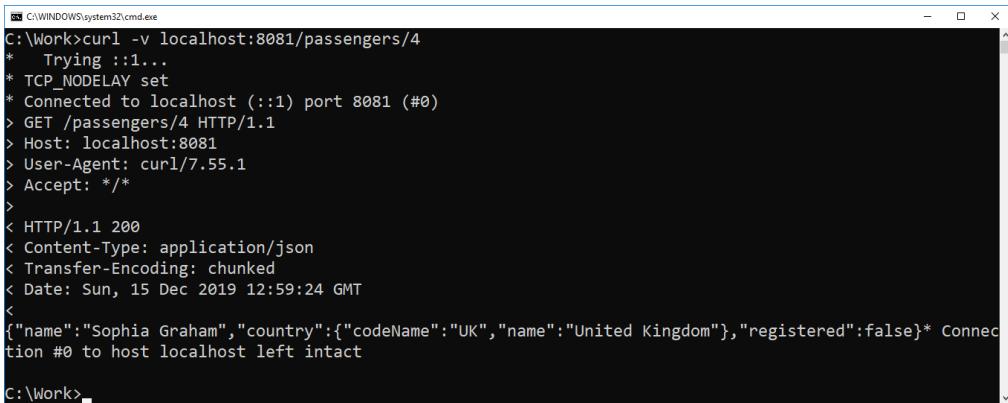
Figure 18.3 The result of running the `curl -v localhost:8081/passengers` command is the list of the passengers

Mike will also test the other functionalities that he has implemented for the `/passengers` endpoint.

To get the passenger having the id 4, he executes the command

```
curl -v localhost:8081/passengers/4
```

The result of the execution is shown in fig. 18.4 – the passenger is provided in JSON format.



```
C:\Work>curl -v localhost:8081/passengers/4
* Trying ::1...
* TCP_NODELAY set
* Connected to localhost (::1) port 8081 (#0)
> GET /passengers/4 HTTP/1.1
> Host: localhost:8081
> User-Agent: curl/7.55.1
> Accept: */*
<
< HTTP/1.1 200
< Content-Type: application/json
< Transfer-Encoding: chunked
< Date: Sun, 15 Dec 2019 12:59:24 GMT
<
>{"name":"Sophia Graham","country":{"codeName":"UK","name":"United Kingdom"},"registered":false}* Connection #0 to host localhost left intact
C:\Work>
```

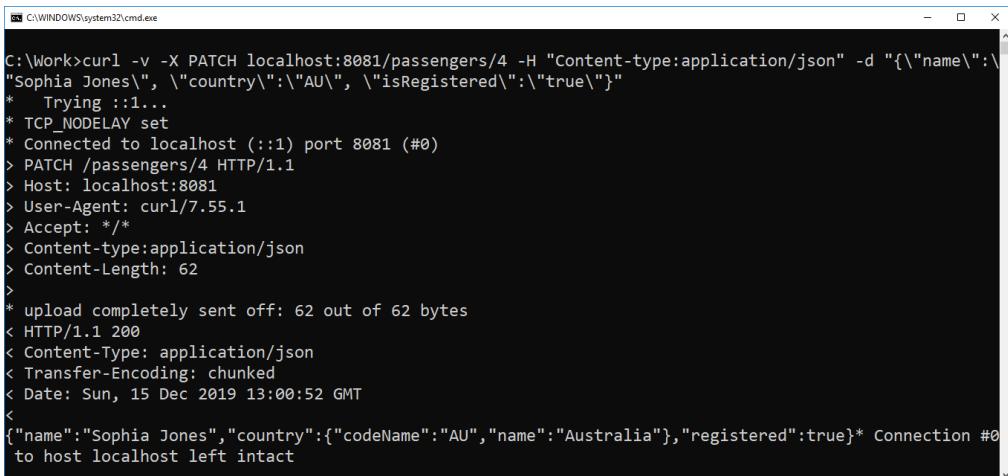
Figure 18.4 The result of running the `curl -v localhost:8081/passengers/4` command is the passenger having id 4

To patch the passenger having the id 4, Mike executes the command

```
curl -v -X PATCH localhost:8081/passengers/4
-H "Content-type:application/json"
-d "{\"name\":\"Sophia Jones\", \"country\":\"AU\",
  \"isRegistered\":true}"
```

This command will update the name, the country and the registered status of the passenger having id 4.

The result of the execution is shown in fig. 18.5 – the passenger is provided in JSON format.



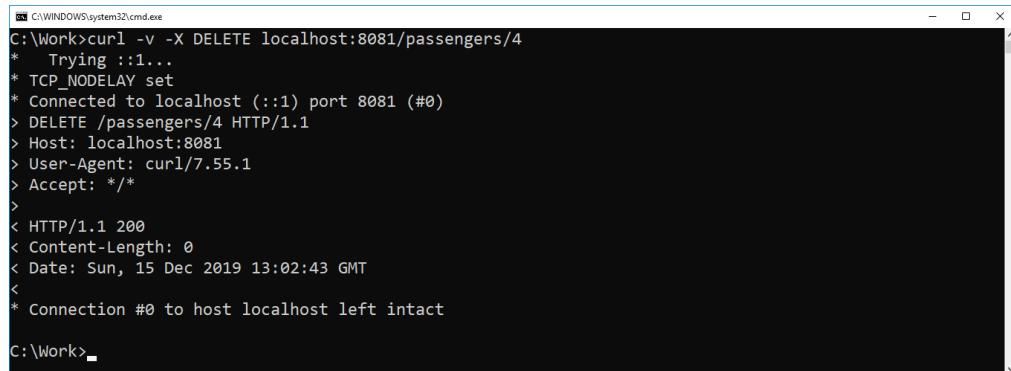
```
C:\Work>curl -v -X PATCH localhost:8081/passengers/4 -H "Content-type:application/json" -d "{\"name\":\"Sophia Jones\", \"country\":\"AU\", \"isRegistered\":true}"
* Trying ::1...
* TCP_NODELAY set
* Connected to localhost (::1) port 8081 (#0)
> PATCH /passengers/4 HTTP/1.1
> Host: localhost:8081
> User-Agent: curl/7.55.1
> Accept: */*
> Content-type:application/json
> Content-Length: 62
>
* upload completely sent off: 62 out of 62 bytes
< HTTP/1.1 200
< Content-Type: application/json
< Transfer-Encoding: chunked
< Date: Sun, 15 Dec 2019 13:00:52 GMT
<
>{"name":"Sophia Jones","country":{"codeName":"AU","name":"Australia"},"registered":true}* Connection #0 to host localhost left intact
C:\Work>
```

Figure 18.5 The result of successfully patching the passenger having id 4 with updated information

To delete the passenger having the id 4, Mike executes the command

```
curl -v -X DELETE localhost:8081/passengers/4
```

This command will delete the passenger having id 4. The result of the execution is shown in fig. 18.6.



The screenshot shows a Windows Command Prompt window titled 'C:\WINDOWS\system32\cmd.exe'. The command entered is 'curl -v -X DELETE localhost:8081/passengers/4'. The output shows the HTTP request being sent and the response received, indicating a successful deletion. The response code is 200, and the date is Sun, 15 Dec 2019 13:02:43 GMT. The connection is left intact.

```
C:\Work>curl -v -X DELETE localhost:8081/passengers/4
*   Trying ::1...
* TCP_NODELAY set
* Connected to localhost (::1) port 8081 (#0)
> DELETE /passengers/4 HTTP/1.1
> Host: localhost:8081
> User-Agent: curl/7.55.1
> Accept: */*
>
< HTTP/1.1 200
< Content-Length: 0
< Date: Sun, 15 Dec 2019 13:02:43 GMT
<
* Connection #0 to host localhost left intact

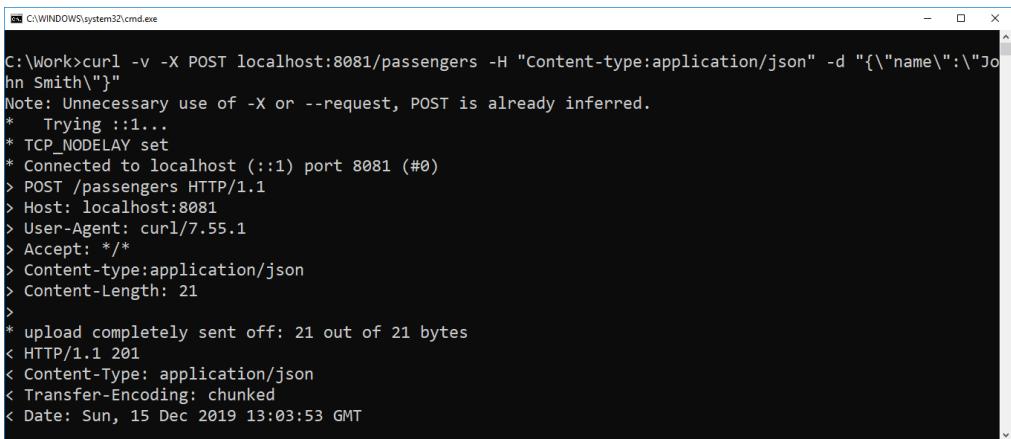
C:\Work>
```

Figure 18.6 The result of deleting the passenger having id 4

Finally, to post a new passenger, Mike executes the command

```
curl -v -X POST localhost:8081/passengers
-H "Content-type:application/json"
-d "{\"name\":\"John Smith\"}"
```

This command will post the newly created passenger. The result of its execution is shown in fig. 18.7 – the passenger is provided in JSON format.



```
C:\Work>curl -v -X POST localhost:8081/passengers -H "Content-type:application/json" -d "{\"name\":\"John Smith\"}"
Note: Unnecessary use of -X or --request, POST is already inferred.
* Trying ::1...
* TCP_NODELAY set
* Connected to localhost (::1) port 8081 (#0)
> POST /passengers HTTP/1.1
> Host: localhost:8081
> User-Agent: curl/7.55.1
> Accept: */*
> Content-type:application/json
> Content-Length: 21
>
* upload completely sent off: 21 out of 21 bytes
< HTTP/1.1 201
< Content-Type: application/json
< Transfer-Encoding: chunked
< Date: Sun, 15 Dec 2019 13:03:53 GMT
```

Figure 18.7 The result of successfully posting the new passenger “John Smith”

18.4 Testing the RESTful API managing two related entities

Mike will now write the tests to automatically verify the behavior of the RESTful API. The code is presented into listing 18.15.

Listing 18.15 The RestApplicationTest class

```
@SpringBootTest #A
@AutoConfigureMockMvc #B
@Import(FlightBuilder.class) #C
public class RestApplicationTest {

    @Autowired #D
    private MockMvc mvc; #D

    @Autowired #E
    private Flight flight; #E

    @Autowired #E
    private Map<String, Country> countriesMap; #E

    @MockBean #F
    private PassengerRepository passengerRepository; #F

    @MockBean #F
    private CountryRepository countryRepository; #F

    @Test
    void testGetAllCountries() throws Exception {
        when(countryRepository.findAll()).thenReturn(new
            ArrayList<>(countriesMap.values())); #G
        mvc.perform(get("/countries")) #G
            .andExpect(status().isOk()) #H
            .andExpect(content().contentType(MediaType.APPLICATION_JSON)) #I
    }
}
```

```

        .andExpect(jsonPath("$.hasSize(3))); #I
    }
    verify(countryRepository, times(1)).findAll(); #J
}

@Test
void testGetAllPassengers() throws Exception {
    when(passengerRepository.findAll()).thenReturn(new
        ArrayList<>(flight.getPassengers()));
    #K
    #K

    mvc.perform(get("/passengers"))
        .andExpect(status().isOk()) #L
        .andExpect(content().contentType(MediaType.APPLICATION_JSON)) #L
        .andExpect(jsonPath("$.hasSize(20))) #L
    verify(passengerRepository, times(1)).findAll(); #M
}

@Test
void testPassengerNotFound() {
    Throwable throwable = assertThrows(NestedServletException.class,
        () -> mvc.perform(get("/passengers/30"))
            .andExpect(status().isNotFound()));
    assertEquals(PassengerNotFoundException.class, #O
        throwable.getCause().getClass()); #O
}

@Test
void testPostPassenger() throws Exception {
    Passenger passenger = new Passenger("Peter Michelsen"); #P
    passenger.setCountry(countriesMap.get("US")); #P
    passenger.setIsRegistered(false); #P
    when(passengerRepository.save(passenger))
        .thenReturn(passenger); #P

    mvc.perform(post("/passengers")
        .content(new ObjectMapper().writeValueAsString(passenger)) #Q
        .header(HttpHeaders.CONTENT_TYPE, MediaType.APPLICATION_JSON)) #Q
        .andExpect(status().isCreated()) #Q
        .andExpect(jsonPath("$.name", is("Peter Michelsen"))); #Q
        .andExpect(jsonPath("$.country.codeName", is("US"))); #Q
        .andExpect(jsonPath("$.country.name", is("USA"))); #Q
        .andExpect(jsonPath("$.registered", is(Boolean.FALSE))); #Q

    verify(passengerRepository, times(1)).save(passenger); #R
}

@Test
void testPatchPassenger() throws Exception {
    Passenger passenger = new Passenger("Sophia Graham"); #S
    passenger.setCountry(countriesMap.get("UK")); #S
    passenger.setIsRegistered(false); #S
    when(passengerRepository.findById(1L))
        .thenReturn(Optional.of(passenger)); #S
    when(passengerRepository.save(passenger))
        .thenReturn(passenger); #T
    String updates =
        "{\"name\":\"Sophia Jones\", \"country\":\"AU\", #U
        #U

```

```

    \"isRegistered\":\"true\"}"; #U

    mvc.perform(patch("/passengers/1")
        .content(updates)
        .header(HttpHeaders.CONTENT_TYPE, MediaType.APPLICATION_JSON)) #U
        .andExpect(content().contentType(MediaType.APPLICATION_JSON)) #U
        .andExpect(status().isOk()); #U

    verify(passengerRepository, times(1)).findById(1L); #V
    verify(passengerRepository, times(1)).save(passenger); #V
}

@Test
public void testDeletePassenger() throws Exception {
    mvc.perform(delete("/passengers/4"))
        .andExpect(status().isOk()); #W
        #W

    verify(passengerRepository, times(1)).deleteById(4L); #X
}

}

```

In the listing above, we are doing the following:

- We are creating the `RestApplicationTest` class and annotate it with `@SpringBootTest (#A)`. `@SpringBootTest` searches in the current package of the test class and in its sub-packages for beans definitions.
- We also annotate the class with `@AutoConfigureMockMvc` in order to enable all auto-configuration related to the `MockMvc` objects we are going to use in the test (#B).
- We are importing `FlightBuilder`, which created a flight bean and a countries map bean (#C).
- We auto-wire a `MockMvc` object (#D). A `MockMvc` is the main entry point for server-side Spring REST application testing. We'll perform a series of REST operations against this `MockMvc` object during the tests.
- We are declaring a `flight` and a `countriesMap` field and auto-wiring them (#E). These fields are injected from the `FlightBuilder` class.
- We are declaring a `countryRepository` and a `passengerRepository` fields and annotate them with `@MockBean (#F)`. `@MockBean` is used to add mock objects to the Spring application context. The mock will replace any existing bean of the same type in the application context. We'll provide instructions for the behavior of the mock objects during the tests.
- In the `testGetAllCountries` test, we instruct the mock `countryRepository` bean to return the array of the values from the `countriesMap` when the `findAll` method is executed on it (#G).
- We simulate the execution of the GET method on the `"/countries"` URL (#H) and verify the returned status, the expectations of the content type, and the returned JSON size (#I). We also verify that the method `findAll` has been executed exactly once on the `countryRepository` bean (#J).

- In the `test GetAllPassengers` test, we instruct the mock `passengerRepository` bean to return the passengers from the `flight` bean when the `findAll` method is executed on it (#K).
- We simulate the execution of the GET method on the “/passengers” URL and verify the returned status, the expectations of the content type and returned JSON size (#L). We also verify that the method `findAll` has been executed exactly once on the `passengerRepository` bean (#M).
- In the `testPassengerNotFound`, we are trying to get the passenger having the id 30 and we are checking that a `NestedServletException` is thrown and that the returned status is “Not Found” (#N). We are also checking that the cause of the `NestedServletException` is a `PassengerNotFoundException` (#O).
- In the `testPostPassenger` we are creating a passenger object, configuring it and instructing the `passengerRepository` to return that object when a `save` is executed on that passenger (#P).
- We simulate the execution of the POST method on the “/passengers” URL and we verify the content to be the JSON string value of the passenger object, the header type, the returned status and the content of the JSON (#Q). We use an object of type `com.fasterxml.jackson.databind.ObjectMapper`, which is the main class of the Jackson library, the standard JSON library for Java. `ObjectMapper` offers functionality for reading and writing JSON, to and from basic POJOs (Plain Old Java Objects).
- We also verify that the `save` method has been executed exactly once on the previously defined passenger (#R).
- In the `testPatchPassenger` we are creating a passenger object, configuring it and instructing the `passengerRepository` to return that object when a passenger `findById` is executed having the argument 1 (#S). When the `save` method is executed on the `passengerRepository`, that passenger is returned as well (#T).
- We set an updates JSON, we perform a PATCH on the “/passengers/1” URL using that update and we check the content and the returned status (#U).
- We verify that the `findById` and `save` methods have been executed exactly once on `passengerRepository` (#V).
- We perform a DELETE operation on the “/passengers/4” URL and we verify the returned status to be OK (#W) and the fact that the `deleteById` method has been executed exactly once (#X).

Running `RestApplicationTest` will be successful, as shown in fig. 18.8.

The screenshot shows the IntelliJ IDEA interface with the RestApplicationTest.java file open. The code is annotated with various IDE features like code completion dropdowns and error markers. Below the code editor is the 'Run' tool window, which shows the 'RestApplicationTest' configuration and its results. The results table indicates 6 tests passed in 859 ms. The log pane at the bottom shows the command-line output of the test execution.

Test	Time	Output
testGetAllPassengers()	375 ms	C:\Program Files\Java\jdk1.8.0_221\bin\java.exe ...
testGetAllCountries()	31 ms	18:19:34.988 [main] DEBUG org.springframework.test.context...
testDeletePassenger()	47 ms	18:19:35.020 [main] DEBUG org.springframework.test.context...
testPostPassenger()	125 ms	18:19:35.082 [main] DEBUG org.springframework.test.context...
testPassengerNotFound()	31 ms	18:19:35.120 [main] INFO org.springframework.boot.test.context...
testPatchPassenger()	250 ms	

Figure 18.8 Successfully running the RestApplicationTest, that checks the RESTful application functionality

The next chapter will be dedicated to testing database applications and the various alternatives to do this.

18.5 Summary

This chapter has covered the following:

- Introducing the REST architectural style and the concept of REST applications.
- Demonstrating what makes an API RESTful and the REST architecture constraints: client-server, stateless, uniform interface, layered system, cacheable, code on demand.
- Creating a RESTful API to manage one entity, the Country from the flights management application, and executing GET operations against it to obtain the list of countries.
- Creating a RESTful API to manage two related entities, the Country and Passenger from the flights management application, and executing GET, PATCH, DELETE, and POST operations against it, to get the list of passengers, to get a particular passenger by id, to create, update or delete a passenger.
- Testing a RESTful API managing two related entities, the Country and Passenger from the flights management application, by creating and executing tests against a Spring REST MockMvc object, to test the previously mentioned GET, PATCH, DELETE, and POST operations.

19

Testing database applications

This chapter covers:

- Examining the challenges of database testing
- Implementing tests for a JDBC application
- Implementing tests for a Spring JDBC application
- Implementing tests for a Hibernate application
- Implementing tests for a Spring Hibernate application
- Comparing the different approaches of building and testing database applications

Dependency is the key problem in software development at all scales.... Eliminating duplication in programs eliminates dependency.

—Kent Beck, Test-Driven Development: By Example

The persistence layer (or, roughly speaking, the database access code) is undoubtedly one of the most important parts of any enterprise project. Despite its importance, the persistence layer is hard to unit test, mainly because of the following three issues:

- Unit tests must exercise code in isolation; the persistence layer requires interaction with an external entity, the database.
- Unit tests must be easy to write and run; code that accesses the database can be cumbersome.
- Unit tests must be fast to run; database access is relatively slow.

We call these issues the *Database Unit Testing Impedance Mismatch*, in a reference to the object-relational impedance mismatch (which describes the difficulties of using a relational database to persist data when an application is written using an object-oriented language).

19.1 The Database Unit Testing Impedance Mismatch

Let's take a deeper look at the three issues that comprise the Database Unit Testing Impedance Mismatch.

19.1.1 Unit tests must exercise code in isolation

From a purist point of view, tests that exercise database access code cannot be considered unit tests, as they depend on an external entity, the almighty database. What should they be called then? Integration Tests? Functional Tests? Non-Unit Unit Tests?

Well, the answer is: there is no secret ingredient! In other words, database tests can fit in many categories, depending on the context.

Pragmatically speaking, though, database access code can be exercised by both unit and integration tests:

- Unit tests are used to test classes that interact directly with the database (like DAOs). A DAO (*Data Access Object*) is an object that provides an interface to a database and maps application calls to the specific database operations without exposing details of the persistence layer. Such tests guarantee these classes execute the proper operation against the database. Although these tests depend on external entities (like the database and/or persistence frameworks), they exercise classes that are building blocks in a bigger application (and hence are units).
- Similarly, unit tests can be written to test the upper layers (like Facades), without the need of accessing the database – in these tests, the persistence layer can be emulated by mocks or stubs. As a facade in architecture, the Facade design pattern provides an object that serves as a front-facing interface masking a more complex underlying code.

There is still a practical question: can't the data present in the database get in the way of the tests? Yes, it is possible, so before we run the tests, we must assure the database is in a known state – and we'll show how to do this in this chapter.

19.1.2 Unit tests must be easy to write and run

It does not matter how much a company, project manager, or technical leader praises unit tests – if they are not easy to write and run, developers will resist writing them. Moreover, writing code that accesses the database is not a straightforward task – one would have to write SQL statements, mix many levels of try/catch/finally code, convert SQL types to and from Java, etc.

Therefore, in order for database unit tests to thrive, it is necessary to alleviate the “database burden” on developers. We'll start our work using pure JDBC (Java Database Connectivity). Then, we'll introduce Spring as the framework used for our application. Finally, we'll move to ORM (Object Relational Mapping) and Hibernate.

JDBC JDBC (Java Database Connectivity) is a Java API (application programming interface) that defines how a client may access a database. JDBC provides methods to query and update data in a relational database.

ORM ORM (Object Relational Mapping) is a programming technique for converting data between relational databases and object-oriented programming languages and vice-versa.

HIBERNATE Hibernate is an object-relational mapping framework for Java. It provides the facilities for mapping an object-oriented domain model to relational database tables. Hibernate manages the incompatibilities between the object-oriented model and the relational database model by replacing the direct database access with object manipulation.

19.1.3 Unit tests must be fast to run

Let's say you overcame the first two issues and have a nice environment, with hundreds of unit tests exercising the objects that access the database, and where a developer can easily add new ones. All seems nice, but when a developer runs the build (and they should do that many times a day, at least after updating their workspace and before submitting changes to the source control system), it takes ten minutes for the build to complete, nine of them spent in the database tests. What should you do then?

This is the hardest issue, as it cannot be always solved. Typically, the delay is caused by the database access per se, as the database is probably a remote server, accessed by dozens of users. A possible solution then is to move the database closer to the developer, by either using an embedded database (if either the application uses standard SQL that enables a database switch or the application uses an ORM framework) or locally installing lighter versions of the database.

DEFINITION: EMBEDDED DATABASE An embedded database is a database that is bundled within an application, instead of being managed by external servers (which is the typical scenario). There is a broad range of embedded databases available for Java applications, most of them based on open-source projects - like H2 (<http://h2database.com>), HSQLDB (<http://hsqldb.org>), Apache Derby (<http://db.apache.org/derby>). Notice that the fundamental characteristic of an embedded database is the fact that it is managed by the application, and not the language it is written in. For instance, both HSQLDB and Derby supports client/server mode (besides the embedded option), while SQLite (which is a C-based product) could also be embedded in a Java application.

In the next sections, we will see how we start with a pure JDBC application and with an embedded database. Then, we move to introduce Spring and Hibernate as ORM (Object Relational Mapping) framework and also take steps to solve the Database Unit Testing Mismatch.

19.2 Testing a JDBC application

JDBC (Java Database Connectivity) is a Java API (application programming interface) that defines how a client may access a database. JDBC provides methods to query and update data in a relational database.

It was first released as part of the Java Development Kit (JDK) 1.1 in 1997. Since then, it has been part of the Java Platform, Standard Edition (Java SE). Being one of the early API in use in Java and designed for the largely spread database applications, it may be still encountered in projects, even not mixed with any other technology.

In our demonstration, we'll start from a pure JDBC application, then introduce Spring and Hibernate, and test all these applications. This will prove how these kinds of database applications can be tested, and also show how the Database Unit Testing Mismatch is reduced.

At Tested Data Systems, the old flights management application was providing the possibility to persist the information into a database. It is George's job to take this application over, analyze it and then move it to the new present-day technologies.

The JDBC application that George receives and needs to take over contains a `Country` class, describing the countries of the passengers of the flights management application that we have already met across the chapters (listing 19.1).

Listing 19.1 The Country class

```
public class Country {  
    private String name; #A  
    private String codeName; #A  
  
    public Country(String name, String codeName) { #B  
        this.name = name; #B  
        this.codeName = codeName; #B  
    } #B  
  
    public String getName() { #A  
        return name; #A  
    } #A  
  
    public void setName(String name) { #A  
        this.name = name; #A  
    } #A  
  
    public String getCodeName() { #A  
        return codeName; #A  
    } #A  
  
    public void setCodeName(String codeName) { #A  
        this.codeName = codeName; #A  
    } #A  
  
    @Override  
    public String toString() { #C  
        return "Country{" + #C  
            "name='" + name + '\'' + #C  
            ", codeName='" + codeName + '\'' + #C  
            '}'; #C
```

```

    }

    @Override
    public boolean equals(Object o) { #D
        if (this == o) return true; #D
        if (o == null || getClass() != o.getClass()) return false; #D
        Country country = (Country) o; #D
        return Objects.equals(name, country.name) && #D
               Objects.equals(codeName, country.codeName); #D
    }

    @Override
    public int hashCode() { #D
        return Objects.hash(name, codeName); #D
    }
}

```

In the listing above we do the following:

- We declare the `name` and `codeName` fields of the `Country` class, together with the corresponding getters and setters (#A).
- We create a constructor of the `Country` class, to initialize the `name` and `codeName` fields (#B).
- We override the `toString` method to nicely display a country (#C).
- We override the `equals` and `hashCode` methods, to take into account the `name` and `codeName` fields (#D).

The application that George is taking over is using the embedded H2 database for testing purposes. The Maven `pom.xml` file will include the JUnit 5 dependencies and the H2 dependencies (listing 19.2).

Listing 19.2 The Maven pom.xml dependencies

```

<dependencies>
    <dependency>
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter-api</artifactId>
        <version>5.6.0</version>
        <scope>test</scope>
    </dependency>
    <dependency>
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter-engine</artifactId>
        <version>5.6.0</version>
        <scope>test</scope>
    </dependency>
    <dependency>
        <groupId>com.h2database</groupId>
        <artifactId>h2</artifactId>
        <version>1.4.193</version>
    </dependency>
</dependencies>

```

The application manages the connections to the database and the operations against the database tables through the `ConnectionManager` and `TablesManager` classes.

Listing 19.3 The ConnectionManager class

```
public class ConnectionManager {  
  
    private static Connection connection; #A  
  
    public static Connection openConnection() {  
  
        try {  
            Class.forName("org.h2.Driver"); #B  
            connection = DriverManager.getConnection("jdbc:h2:~/country",  
                "sa", #C  
                ""); #C  
        ); #C  
        return connection; #D  
    } catch(ClassNotFoundException | SQLException e) { #E  
        throw new RuntimeException(e); #E  
    } #E  
  
    public static void closeConnection() {  
        if (null != connection) { #F  
            try {  
                connection.close(); #G  
            } catch(SQLException e) { #H  
                throw new RuntimeException(e); #H  
            } #H  
        } #F  
    } #G  
}
```

In the listing above, we do the following:

- We declare a `Connection` type `connection` field (#A).
- In the `openConnection` method, we load the H2 driver (#B) and we initialize the previously declared `connection` field to access the H2 country database using JDBC, `sa` as a user and no password (#C). If everything goes fine, we return the initialized connection (#D).
- If the H2 driver class hasn't been found or we encounter an `SQLException`, we catch it and re-throw a `RuntimeException` (#E).
- In the `closeConnection` method, we first check the `connection` not to be null (#F), then we try to close it (#G). In case an `SQLException` occurs, we catch it and re-throw a `RuntimeException` (#H).

Listing 19.4 The TablesManager class

```
public class TablesManager {  
  
    public static void createTable() {  
        String sql = "CREATE TABLE COUNTRY( ID IDENTITY,  
            NAME VARCHAR(255), CODE_NAME VARCHAR(255) );"; #A  
        #A
```

```

        executeStatement(sql);                                #B
    }

    public static void dropTable() {
        String sql = "DROP TABLE IF EXISTS COUNTRY;";      #C
        executeStatement(sql);                                #D
    }

    private static void executeStatement(String sql) {
        PreparedStatement statement;                         #E

        try {
            Connection connection = openConnection();       #F
            statement = connection.prepareStatement(sql);   #G
            statement.executeUpdate();                      #H
            statement.close();                            #I
        } catch (SQLException e) {
            throw new RuntimeException(e);                #J
        } finally {
            closeConnection();                           #K
        }
    }
}

```

In the listing above, we do the following:

- In the `createTable` method, we declare an SQL CREATE TABLE statement that creates the COUNTRY table with an ID identity field and the NAME and CODE_NAME fields of type VARCHAR (#A). Then, we execute this statement (#B).
- In the `dropTable` method, we declare an SQL DROP TABLE statement that drops the COUNTRY table, if it exists (#C). Then, we execute this statement (#D).
- The `executeStatement` method declares a `PreparedStatement` variable (#E). Then, it opens a connection (#F), prepares the statement (#G), executes it (#H) and closes it (#I). If an `SQLException` is caught, we re-throw a `RuntimeException` (#J). No matter if the statement is successfully executed or not, we close the connection (#K).

The application declares a `CountryDao` class, an implementation of the DAO (Data Access Object) pattern, which provides an abstract interface to the database and executes queries against it.

Listing 19.5 The CountryDao class

```

public class CountryDao {
    private static final String GET_ALL_COUNTRIES_SQL =      #A
        "select * from country";                          #A
    private static final String GET_COUNTRIES_BY_NAME_SQL = #B
        "select * from country where name like ?";        #B

    public List<Country> getCountryList() {               #C
        List<Country> countryList = new ArrayList<>();

```

```

try {
    Connection connection = openConnection();                                #D
    PreparedStatement statement =
        connection.prepareStatement(GET_ALL_COUNTRIES_SQL);                  #E
    ResultSet resultSet = statement.executeQuery();                            #E

    while (resultSet.next()) {                                              #F
        countryList.add(new Country(resultSet.getString(2),                #F
                                       resultSet.getString(3)));                      #F
    }
    statement.close();                                                       #G
} catch (SQLException e) {                                                 #H
    throw new RuntimeException(e);                                         #H
} finally {
    closeConnection();                                                     #I
}
return countryList;                                                       #J
}

public List<Country> getCountryListStartWith(String name) {               #K
    List<Country> countryList = new ArrayList<>();                         #K

    try {
        Connection connection = openConnection();                            #L
        PreparedStatement statement =
            connection.prepareStatement(GET_COUNTRIES_BY_NAME_SQL);          #M
        statement.setString(1, name + "%");                                    #M
        ResultSet resultSet = statement.executeQuery();                      #M

        while (resultSet.next()) {                                            #N
            countryList.add(new Country(resultSet.getString(2),                #N
                                           resultSet.getString(3)));                      #N
        }
        statement.close();                                                   #O
    } catch (SQLException e) {                                               #P
        throw new RuntimeException(e);                                         #P
    } finally {
        closeConnection();                                                 #Q
    }
    return countryList;                                                    #R
}
}

```

In the listing above, we do the following:

- We declare two SQL SELECT statements, to get all the countries from the COUNTRY table (#A) and to get the countries whose names match a pattern (#B).
- In the `getCountryList` method we initialize an empty country list (#C), we open a connection (#D), prepare the statement and execute it (#E).
- We pass through all the results returned from the database and add them to the countries list (#F). Then, we close the statement (#G). If an `SQLException` is caught, we re-throw a `RuntimeException` (#H). No matter if the statement is successfully executed or not, we close the connection (#I). We return the countries list at the end of the method (#J).

- In the `getCountryListStartWith` method we initialize an empty country list (#K), we open a connection (#L), prepare the statement and execute it (#M).
- We pass through all the results returned from the database and add them to the countries list (#N). Then, we close the statement (#O). If an `SQLException` is caught, we re-throw a `RuntimeException` (#P). No matter if the statement is successfully executed or not, we close the connection (#Q). We return the countries list at the end of the method (#R).

Moving to the side of the test, we have here two classes: `CountriesLoader`, which is populating the database and makes sure that it is in a known state and `CountriesDatabaseTest` that is effectively testing the interaction of the application with the database.

Listing 19.6 The CountriesLoader class

```
public class CountriesLoader {

    private static final String LOAD_COUNTRIES_SQL = #A
        "insert into country (name, code_name) values ";

    public static final String[][] COUNTRY_INIT_DATA = { #B
        { "Australia", "AU" }, { "Canada", "CA" }, { "France", "FR" }, #B
        { "Germany", "DE" }, { "Italy", "IT" }, { "Japan", "JP" }, #B
        { "Romania", "RO" }, { "Russian Federation", "RU" }, #B
        { "Spain", "ES" }, { "Switzerland", "CH" }, #B
        { "United Kingdom", "UK" }, { "United States", "US" } }; #B

    public void loadCountries() {
        for (String[] countryData : COUNTRY_INIT_DATA) { #C
            String sql = LOAD_COUNTRIES_SQL + "(" + countryData[0] + ", #D
                " + countryData[1] + ")"; #D

            try {
                Connection connection = openConnection(); #E
                PreparedStatement statement =
                    connection.prepareStatement(sql); #F
                statement.executeUpdate(); #F
                statement.close(); #F
            } catch (SQLException e) { #G
                throw new RuntimeException(e); #G
            } finally {
                closeConnection(); #H
            }
        }
    }
}
```

In the listing above, we do the following:

- We declare one SQL INSERT statement, to insert a country into the COUNTRY table (#A). We then declare the initialization data for the countries to be inserted (#B).
- In the `loadCountries` method, we browse the initialization data for the countries (#C) and build the SQL query that is inserting each particular country (#D).

- We open a connection (#E), prepare the statement, execute it and close it (#F). If an SQLException is caught, we re-throw a RuntimeException (#G). No matter if the statement is successfully executed or not, we close the connection (#H).

Listing 19.7 The CountriesDatabaseTest class

```
import static com.manning.junitbook.databases.CountriesLoader.COUNTRY_INIT_DATA;           #A
[...]

public class CountriesDatabaseTest {
    private CountryDao countryDao = new CountryDao();                                     #A
    private CountriesLoader countriesLoader = new CountriesLoader();                     #A

    private List<Country> expectedCountryList = new ArrayList<>();                   #B
    private List<Country> expectedCountryListStartsWithA =                                #C
        new ArrayList<>();                                                               #C

    @BeforeEach
    public void setUp() {                                                               #D
        TablesManager.createTable();                                                 #E
        initExpectedCountryLists();                                                 #F
        countriesLoader.loadCountries();                                              #G
    }

    @Test
    public void testCountryList() {                                                     #H
        List<Country> countryList = countryDao.getCountryList();                      #I
        assertNotNull(countryList);                                                 #I
        assertEquals(expectedCountryList.size(), countryList.size());                 #J
        for (int i = 0; i < expectedCountryList.size(); i++) {                         #K
            assertEquals(expectedCountryList.get(i), countryList.get(i));             #K
        }
    }

    @Test
    public void testCountryListStartsWithA() {                                         #L
        List<Country> countryList =                                           #L
            countryDao.getCountryListStartWith("A");                               #L
        assertNotNull(countryList);                                                 #M
        assertEquals(expectedCountryListStartsWithA.size(),                         #N
            countryList.size());                                                 #N
        for (int i = 0; i < expectedCountryListStartsWithA.size(); i++) {          #O
            assertEquals(expectedCountryListStartsWithA.get(i),                      #O
                countryList.get(i));                                              #O
        }
    }

    @AfterEach
    public void dropDown() {                                                       #P
        TablesManager.dropTable();                                                 #Q
    }

    private void initExpectedCountryLists() {
        for (int i = 0; i < COUNTRY_INIT_DATA.length; i++) {                      #R
            String[] countryInitData = COUNTRY_INIT_DATA[i];
            Country country = new Country(countryInitData[0],                      #S
                countryInitData[1]);                                                 #S
            expectedCountryList.add(country);                                         #T
        }
    }
}
```

```

        if (country.getName().startsWith("A")) { #U
            expectedCountryListStartsWithA.add(country); #U
        }
    }
}

```

In the listing above, we do the following:

- We statically import the countries data from CountriesLoader and we initialize a CountryDao and a CountriesLoader (#A). We initialize an empty list of expected countries (#B) and an empty list of expected countries that start with 'A' (#C). We can do this for any letter, but our test is looking now for the countries with names starting with 'A'.
- We mark the `setUp` method with the `@BeforeEach` annotation, to be executed before each test (#D). Inside it, we create the empty COUNTRY table into the database (#E), we initialize the expected countries list (#F) and we load the countries into the database (#G).
- In the `testCountryList` method, we initialize the countries list from the database by using the `getCountryList` from the `CountryDao` class (#H). Then, we check that the list we have obtained is not null (#I), it has the expected size (#J) and that its content is the expected one (#K).
- In the `testCountryListStartsWithA` method, we initialize the countries list starting with 'A' from the database by using the `getCountryListStartWith` from the `CountryDao` class (#L). Then, we check that the list we have obtained is not null (#M), it has the expected size (#N) and that its content is the expected one (#O).
- We mark the `dropDown` method with the `@AfterEach` annotation, to be executed after each test (#P). Inside it, we drop the COUNTRY table from the database (#Q).
- In the `initExpectedCountryLists` method we browse the countries initialization data (#R), create a `Country` object at each step (#S) and add it to the expected countries list (#T). If the name of the country starts with 'A', we also add it to the expected countries list whose names start with 'A' (#U).

The tests are successfully running, as shown in fig. 19.1.

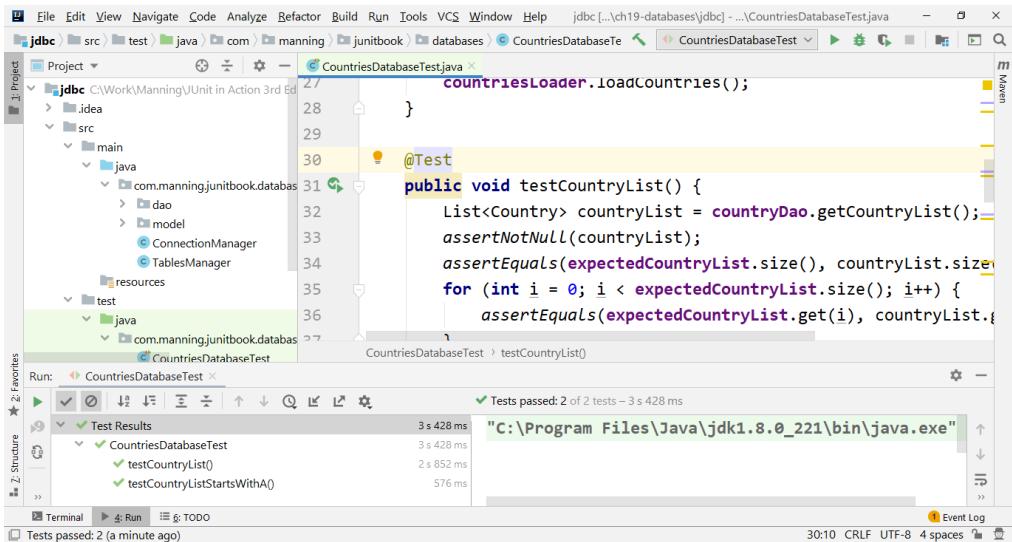


Figure 19.1 Successfully running the tests from the JDBC application that checks the interaction with the COUNTRY table

This is the state of the application that George has to take over, in order to improve the way it is tested. The application is accessing and testing the database through JDBC, which requires a lot of tedious code to do the following:

- Create and open the connection
- Specify, prepare and execute the statement
- Iterate through the results
- Do the work for each iteration
- Process the exceptions
- Close the connection

George will now look for means to reduce the “database burden” from the developers so that they are able to improve the way tests are written and to reduce the Database Unit Testing Mismatch.

19.3 Testing a Spring JDBC application

Spring is a lightweight but at the same time flexible and universal framework used for creating Java applications. We have already introduced the testing of Spring applications in our previous chapters. George has decided to introduce Spring in the database application that he has to take over in order to reduce the “database burden” and to handle through the Spring Inversion of Control some of the tasks of the interaction with the database.

The application will keep the Country class untouched, and George will make some other changes for the migration to Spring. First, he will introduce the new dependencies into the Maven pom.xml file.

Listing 19.8 The new dependencies introduced into the Maven pom.xml file

```
<dependency> #A
    <groupId>org.springframework</groupId> #A
    <artifactId>spring-context</artifactId> #A
    <version>5.2.1.RELEASE</version> #A
</dependency> #A
<dependency> #B
    <groupId>org.springframework</groupId> #B
    <artifactId>spring-jdbc</artifactId> #B
    <version>5.2.1.RELEASE</version> #B
</dependency> #B
<dependency> #C
    <groupId>org.springframework</groupId> #C
    <artifactId>spring-test</artifactId> #C
    <version>5.2.1.RELEASE</version> #C
</dependency>
```

In the listing above, we have added the following dependencies:

- spring-context, the dependency for the Spring Inversion of Control container (#A).
- spring-jdbc, as the application is still using JDBC to access the database. The control of working with connections, preparing and executing statements, processing exceptions is handled by Spring (#B).
- spring-test, the dependency that provides support for writing tests with the help of Spring and that is necessary to use the SpringExtension and the @ContextConfiguration annotation. (#C).

In the test/resources project folder, George will insert two files, one to create the database schema and one to configure the Spring context of the application.

Listing 19.9 The db-schema.sql file

```
create table country( id identity , name varchar (255) , code_name varchar (255) );
```

In the listing above, we are creating the COUNTRY table with three fields: ID (identity field), NAME and CODE_NAME (VARCHAR type).

Listing 19.10 The application-context.xml file

```
<jdbc:embedded-database id="dataSource" type="H2">
    <jdbc:script location="classpath:db-schema.sql"/>
</jdbc:embedded-database> #A
#A
<bean id="countryDao" #B
      class="com.manning.junitbook.databases.dao.CountryDao"> #B
      <property name="dataSource" ref="dataSource"/> #B
</bean> #B
```

```

<bean id="countriesLoader"                                     #C
      class="com.manning.junitbook.databases.CountriesLoader"> #C
      <property name="dataSource" ref="dataSource"/>                 #C
</bean>                                                       #C

```

In the listing above, we instruct the Spring container to create three beans:

- `dataSource`, pointing to a JDBC embedded database of type H2. We initialize the database with the help of the `db-schema.sql` file from listing 19.9, which is on the classpath (#A).
- `countryDao`, the DAO bean for executing SELECT queries to the database (#B). It has a `dataSource` property pointing out to the previously declared `dataSource` bean.
- `countriesLoader`, the bean that initializes the content of the database and brings it to a known state (#C). It also has a `dataSource` property pointing out to the previously declared `dataSource` bean.

George will change the `CountriesLoader` class that creates the countries from the database and sets it in a known state.

Listing 19.11 The CountriesLoader class

```

public class CountriesLoader extends JdbcDaoSupport          #A
{
    private static final String LOAD_COUNTRIES_SQL =           #B
        "insert into country (name, code_name) values ";
    public static final String[][] COUNTRY_INIT_DATA = {         #C
        { "Australia", "AU"}, { "Canada", "CA" }, { "France", "FR" },
        { "Germany", "DE" }, { "Italy", "IT" }, { "Japan", "JP" },
        { "Romania", "RO" }, { "Russian Federation", "RU" },
        { "Spain", "ES" }, { "Switzerland", "CH" },
        { "United Kingdom", "UK" }, { "United States", "US" } };
    public void loadCountries() {
        for (String[] countryData : COUNTRY_INIT_DATA) {          #D
            String sql = LOAD_COUNTRIES_SQL + "(" + countryData[0] + ", "
                + countryData[1] + ")";
            getJdbcTemplate().execute(sql);                         #E
        }
    }
}

```

In the listing above, we do the following:

- We are declaring the `CountriesLoader` class as extending `JdbcDaoSupport` (#A).
- We declare one SQL INSERT statement, to insert a country into the `COUNTRY` table (#B). We then declare the initialization data for the countries to be inserted (#C).
- In the `loadCountries` method, we browse the initialization data for the countries (#D), build the SQL query that is inserting each particular country and execute it against the database (#E). With the Spring Inversion of Control approach, there is no

need to do the previous tedious jobs: open the connection, prepare the statement, execute it and close it, treat the exceptions and close the connection.

Spring JDBC classes

`JdbcDaoSupport` is a Spring JDBC class that facilitates configuring and transferring the database parameters. If a class extends `JdbcDaoSupport`, `JdbcDaoSupport` hides how a `JdbcTemplate` is created.

`JdbcTemplate` is the central class in the package `org.springframework.jdbc.core`. `getJdbcTemplate` is a final method from the `JdbcDaoSupport` class that provides access to an already initialized `JdbcTemplate` object that executes SQL queries, iterates over results and catches JDBC exceptions.

George will also change the tested code to use Spring and to reduce here as well the “database burden” on the side of the developers. He will first implement the `CountryRowMapper` class that is taking care of the mapping rules between the columns from the COUNTRY database table and the fields of the application `Country` class.

Listing 19.12 The `CountryRowMapper` class

```
public class CountryRowMapper implements RowMapper<Country> { #A
    public static final String NAME = "name"; #B
    public static final String CODE_NAME = "code_name"; #B

    @Override
    public Country mapRow(ResultSet resultSet, int i) throws SQLException { #C
        Country country = new Country(resultSet.getString(NAME), #C
            resultSet.getString(CODE_NAME)); #C
        return country; #D
    }
}
```

In the listing above, we do the following:

- We are declaring the `CountryRowMapper` class as implementing `RowMapper` (#A). `RowMapper` is a Spring JDBC interface that is doing mapping of the `ResultSet` obtained by accessing a database to certain objects.
- We declare the string constants to be used in the class representing the names of the table columns (#B). The class will define once how to map the columns to the object fields and may be reused. There is no more need to set the parameters of the statement each time, as we were doing in the JDBC version.
- We override the `mapRow` method inherited from the `RowMapper` interface. We are getting the two string parameters from the `ResultSet` coming from the database and build a `Country` object (#C) that we return at the end of the method (#D).

George will change the existing `CountryDao` class to use Spring to interact with the database.

Listing 19.13 The CountryDao class

```
public class CountryDao extends JdbcDaoSupport {  
    private static final String GET_ALL_COUNTRIES_SQL =  
        "select * from country";  
    private static final String GET_COUNTRIES_BY_NAME_SQL =  
        "select * from country where name like :name";  
  
    private static final CountryRowMapper COUNTRY_ROW_MAPPER =  
        new CountryRowMapper();  
  
    public List<Country> getCountryList() {  
        List<Country> countryList =  
            getJdbcTemplate().query(GET_ALL_COUNTRIES_SQL, COUNTRY_ROW_MAPPER);  
        return countryList;  
    }  
  
    public List<Country> getCountryListStartWith(String name) {  
        NamedParameterJdbcTemplate namedParameterJdbcTemplate =  
            new NamedParameterJdbcTemplate(getDataSource());  
        SqlParameterSource sqlParameterSource =  
            new MapSqlParameterSource("name", name + "%");  
        return namedParameterJdbcTemplate.query(GET_COUNTRIES_BY_NAME_SQL,  
            sqlParameterSource, COUNTRY_ROW_MAPPER);  
    }  
}
```

In the listing above, we do the following:

- We are declaring the CountryDao class as extending `JdbcDaoSupport` (#A).
- We declare two SQL SELECT statements, to get all the countries from the COUNTRY table and to get the countries whose names match a pattern (#B). Into the second statement, we have replaced the parameter with a named parameter (:name) and we are going to use it this way into the class.
- We are initializing a `CountryRowMapper` instance, the class that we have previously created (#C).
- In the `getCountryList` method, we query the COUNTRY table using the SQL that returns all the countries and the `CountryRowMapper` that will match the columns from the table to the fields from the `Country` object. We are directly returning a list of `Country` objects (#D).
- In the `getCountryListStartWith` method, we initialize a `NamedParameterJdbcTemplate` variable (#E). `NamedParameterJdbcTemplate` allows the use of named parameters instead of the previously used '?' placeholders. The `getDataSource` method which is the argument of the `NamedParameterJdbcTemplate` constructor is a final method inherited from `JdbcDaoSupport` and it returns the JDBC `DataSource` used by a DAO.
- We initialize an `SqlParameterSource` variable (#F). `SqlParameterSource` defines the functionality of the objects that can offer parameter values for named SQL parameters and can serve as an argument for `NamedParameterJdbcTemplate`.

operations.

- We query the COUNTRY table using the SQL that returns all the countries having names starting with 'A' and the CountryRowMapper that will match the columns from the table to the fields from the Country object (#G).

George will finally change the existing CountriesDatabaseTest to take advantage of the Spring JDBC approach.

Listing 19.14 The CountriesDatabaseTest class

```
@ExtendWith(SpringExtension.class)                                     #A
@ContextConfiguration("classpath:application-context.xml")          #B
public class CountriesDatabaseTest {

    @Autowired
    private CountryDao countryDao;                                     #C
    #C

    @Autowired
    private CountriesLoader countriesLoader;                           #D
    #D

    private List<Country> expectedCountryList = new ArrayList<Country>(); #E
    private List<Country> expectedCountryListStartsWithA =           #F
        new ArrayList<Country>();                                     #F

    @BeforeEach
    public void setUp() {                                              #G
        initExpectedCountryLists();                                    #H
        countriesLoader.loadCountries();                             #I
    }

    @Test
    @DirtiesContext
    public void testCountryList() {                                     #J
        List<Country> countryList = countryDao.getCountryList();      #K
        assertNotNull(countryList);                                    #L
        assertEquals(expectedCountryList.size(), countryList.size()); #M
        for (int i = 0; i < expectedCountryList.size(); i++) {       #N
            assertEquals(expectedCountryList.get(i), countryList.get(i)); #N
        }
    }

    @Test
    @DirtiesContext
    public void testCountryListStartsWithA() {                         #J
        List<Country> countryList = countryDao.getCountryListStartWith("A");#O
        assertNotNull(countryList);                                    #P
        assertEquals(expectedCountryListStartsWithA.size(),           #Q
            countryList.size());                                     #Q
        for (int i = 0; i < expectedCountryListStartsWithA.size(); i++) { #R
            assertEquals(expectedCountryListStartsWithA.get(i),         #R
                countryList.get(i));                                    #R
        }
    }

    private void initExpectedCountryLists() {
        for (int i = 0; i < CountriesLoader.COUNTRY_INIT_DATA.length; i++) {
```

```
        String[] countryInitData = CountriesLoader.COUNTRY_INIT_DATA[i]; #S
        Country country = new Country(countryInitData[0], #T
                                       countryInitData[1]); #T
        expectedCountryList.add(country); #U
        if (country.getName().startsWith("A")) { #V
            expectedCountryListStartsWithA.add(country); #V
        } #V
    } #V
}
```

In the listing above, we do the following:

- We are annotating the test class to be extended with SpringExtension (#A). SpringExtension is used to integrate the Spring TestContext with the JUnit 5 Jupiter Test.
 - We are also annotating the test class to look for the context configuration into the application-context.xml file from the classpath (#B).
 - We are auto-wiring a CountryDao (#C) and a CountriesLoader bean (#D), which are declared into the application-context.xml file.
 - We initialize an empty list of expected countries (#E) and an empty list of expected countries that start with 'A' (#F).
 - We mark the `setUp` method with the `@BeforeEach` annotation, to be executed before each test (#G). Inside it, we initialize the expected countries list (#H) and we load the countries into the database (#I). The database is initialized by Spring, and we have eliminated its manual initialization.
 - We annotate the test methods with the `@DirtiesContext` annotation (#J). This annotation is used when a test has modified the context (in our case, the state of the embedded database). This will reduce the "database burden" from the developers. Subsequent tests will be supplied with a new unmodified context.
 - In the `testCountryList` method, we initialize the countries list from the database by using the `getCountryList` from the `CountryDao` class (#K). Then, we check that the list we have obtained is not null (#L), it has the expected size (#M) and that its content is the expected one (#N).
 - In the `testCountryListStartsWithA` method, we initialize the countries list starting with 'A' from the database by using the `getCountryListStartWith` from the `CountryDao` class (#O). Then, we check that the list we have obtained is not null (#P), it has the expected size (#Q) and that its content is the expected one (#R).
 - In the `initExpectedCountryLists` method we browse the countries initialization data (#S), create a `Country` object at each step (#T) and add it to the expected countries list (#U). If the name of the country starts with 'A', we also add it to the expected countries list whose names start with 'A' (#V).

The tests are successfully running, as shown in fig. 19.2.

The screenshot shows the IntelliJ IDEA interface with the following details:

- Project View:** Shows the project structure under "spring-jdbc".
- Code Editor:** Displays `CountryDatabaseTest.java` with the following code:

```
private CountriesLoader countriesLoader;

private List<Country> expectedCountryList = new ArrayList<>();
private List<Country> expectedCountryListStartsWithA = new ArrayList<>();

@BeforeEach
public void setUp() {
    initExpectedCountryLists();
    countriesLoader.loadCountries();
}
```
- Run Results:** Shows test results for `CountriesDatabaseTest`:
 - Tests passed: 2 of 2 tests – 179 ms
 - Test 1: CountriesDatabaseTest.testCountryList() (179 ms)
 - Test 2: CountriesDatabaseTest.testCountryListStartsWithA() (74 ms)
- Event Log:** Displays log entries from the Java runtime environment.

Figure 19.2 Successfully running the tests from the Spring JDBC application that checks the interaction with the COUNTRY table

The application is now accessing and testing the database through Spring JDBC, this approach has a few advantages:

- It no longer requires the previously existing large amount of tedious code.
- We no longer create and open the connections by ourselves.
- We no longer prepare and execute the statement, process the exceptions or close the connections.
- Mainly, we must take care of the application context configuration to be handled by Spring and we also must take care of the row mapper. Otherwise, we have to specify only the statement and iterate through the results.

Spring allows configuration alternatives, as we have already demonstrated in the previous chapters. We have used here the XML-based configuration for the tests, as being easier to change, especially by less technical people. Be aware that the Spring Framework is a large topic and for this chapter, we are mostly discussing the things related to testing database applications with JUnit 5. For a comprehensive book on this topic, also showing in detail the configuration possibilities, please refer to the Manning “Spring in Action” by Craig Walls (<https://www.manning.com/books/spring-in-action-fifth-edition>).

George will consider further alternatives to test the interaction with the database and to keep a reduced “database burden” for the developers. We’ll see more in the next sections.

19.4 Testing a Hibernate application

JPA (Java Persistence API) is the specification describing the management of the relational data, the API that the client will operate with and the metadata for the object-relational mapping.

Hibernate is an object-relational mapping framework for Java, implementing the JPA specifications. It existed before the first publishing of the JPA specifications. Therefore, Hibernate has also retained its old native API, being able to offer some non-standard features. For our demonstration, we'll use the standard Java Persistence API.

Hibernate provides the facilities for mapping an object-oriented domain model to relational database tables. Hibernate manages the incompatibilities between the object-oriented model and the relational database model by replacing the direct database access with object handling functions.

Working with Hibernate provides a series of advantages for accessing and testing the database:

- Speeding development. It eliminates repetitive code like mapping query result columns to object fields and vice-versa.
- Making data access more abstract and portable. The ORM implementation classes know how to write vendor-specific SQL, so we do not have to.
- Cache management. Entities are cached in memory, thereby reducing the load on the database.
- Generating boilerplate code for basic CRUD operations (Create, Read, Update, Delete).

George will first introduce the Hibernate dependency into the Maven pom.xml configuration (listing 19.15).

Listing 19.15 The Hibernate dependency introduced into the Maven pom.xml file

```
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-core</artifactId>
    <version>5.4.9.Final</version>
</dependency>
```

Then, George will change the Country class to annotate it as an entity and to annotate the fields from it as columns in a table.

Listing 19.16 The annotated Country class

```
@Entity #A
@Table(name = "COUNTRY") #B
public class Country {
    @Id #C
    @GeneratedValue(strategy = GenerationType.IDENTITY) #D
    @Column(name = "ID") #E
    private int id;

    @Column(name = "NAME") #F
```

```

private String name;

@Column(name = "CODE_NAME") #F
private String codeName;

[...]
}

```

In the listing above we do the following:

- We annotate the `Country` class with `@Entity`, meaning that it has the ability to represent objects in a database (#A). The corresponding table into the database is provided by the `@Table` annotation and is named `COUNTRY` (#B).
- The `id` field is marked as the primary key (#C), its value is automatically generated using a database identity column (#D). The corresponding table column is `ID` (#E).
- We also mark the corresponding columns of the `name` and `codeName` fields in the class by annotating them with `@Column` (#F).

The `persistence.xml` file is the standard configuration for Hibernate. It is located in the `test/resources/META-INF` folder.

Listing 19.17 The `persistence.xml` file

```

<persistence-unit name="manning.hibernate"> #A
    <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider> #B
    <class>com.manning.junitbook.databases.model.Country</class> #C

    <properties>
        <property name="javax.persistence.jdbc.driver" #D
                 value="org.h2.Driver"/> #D
        <property name="javax.persistence.jdbc.url" #E
                 value="jdbc:h2:mem:test;DB_CLOSE_DELAY=-1"/> #E
        <property name="javax.persistence.jdbc.user" value="sa"/> #F
        <property name="javax.persistence.jdbc.password" value="" /> #F
        <property name="hibernate.dialect" #G
                 value="org.hibernate.dialect.H2Dialect"/> #G
        <property name="hibernate.show_sql" value="true"/> #H
        <property name="hibernate.hbm2ddl.auto" value="create"/> #I
    </properties>
</persistence-unit>

```

In the listing above we do the following:

- We specify the persistence unit as `manning.hibernate` (#A). The `persistence.xml` file must define a persistence unit with a unique name in the currently scoped classloader.
- We specify the provider, meaning the underlying implementation of the JPA (Java Persistence API) `EntityManager` (#B). An `EntityManager` manages a set of persistent objects and has an API to insert new objects and read/update/delete the existing ones. In our case, the `EntityManager` is `Hibernate`, currently the most popular JPA implementation.
- We define the entity class that is managed by `Hibernate` as the `Country` class from our

application (#C).

- We specify the JDBC driver as H2, as this is the database type is use (#D).
- We specify the URL of the H2 database. In addition, the `DB_CLOSE_DELAY=-1` will keep the database open and the content of the in-memory database as long as the virtual machine is alive (#E).
- We specify the credentials to access the database – user and password (#F).
- We set the SQL dialect for the generated query to `H2Dialect` (#G), we show the generated SQL query on the console (#H).
- We create the database schema from the scratch every time when we execute the tests (#I).

George will finally rewrite the test that verifies the functionality of the database application, this time using Hibernate.

Listing 19.18 The CountriesHibernateTest file

```
public class CountriesHibernateTest {  
  
    private EntityManagerFactory emf; #A  
    private EntityManager em; #B  
  
    private List<Country> expectedCountryList = #C  
        new ArrayList<>(); #C  
    private List<Country> expectedCountryListStartsWithA = #C  
        new ArrayList<>(); #C  
  
    public static final String[][] COUNTRY_INIT_DATA = { #D  
        { "Australia", "AU" }, { "Canada", "CA" }, { "France", "FR" }, #D  
        { "Germany", "DE" }, { "Italy", "IT" }, { "Japan", "JP" }, #D  
        { "Romania", "RO" }, { "Russian Federation", "RU" }, #D  
        { "Spain", "ES" }, { "Switzerland", "CH" }, #D  
        { "United Kingdom", "UK" }, { "United States", "US" } }; #D  
  
    @BeforeEach #E  
    public void setUp() { #F  
        initExpectedCountryLists();  
  
        emf = Persistence.createEntityManagerFactory("manning.hibernate"); #G  
        em = emf.createEntityManager(); #G  
  
        em.getTransaction().begin(); #H  
  
        for (int i = 0; i < COUNTRY_INIT_DATA.length; i++) { #I  
            String[] countryInitData = COUNTRY_INIT_DATA[i]; #I  
            Country country = new Country(countryInitData[0], #I  
                countryInitData[1]); #I  
            em.persist(country); #J  
        }  
  
        em.getTransaction().commit(); #H  
    }  
  
    @Test  
    public void testCountryList() {
```

```

        List<Country> countryList = em.createQuery(
            "select c from Country c").getResultList();                      #K
        assertNotNull(countryList);                                         #K
        assertEquals(COUNTRY_INIT_DATA.length, countryList.size());          #L
        for (int i = 0; i < expectedCountryList.size(); i++) {               #M
            assertEquals(expectedCountryList.get(i), countryList.get(i));    #N
        }                                                               #N

    }

    @Test
    public void testCountryListStartsWithA() {
        List<Country> countryList = em.createQuery(
            "select c from Country c where c.name like 'A%'").           #O
            getResultList();                                              #O
        assertNotNull(countryList);                                         #P
        assertEquals(expectedCountryListStartsWithA.size(),                 #Q
                    countryList.size());                                     #Q
        for (int i = 0; i < expectedCountryListStartsWithA.size(); i++) {   #R
            assertEquals(expectedCountryListStartsWithA.get(i),             #R
                        countryList.get(i));                                #R
        }                                                               #R
    }

    @AfterEach
    public void dropDown() {                                              #S
        em.close();                                                       #T
        emf.close();                                                      #T
    }

    private void initExpectedCountryLists() {
        for (int i = 0; i < COUNTRY_INIT_DATA.length; i++) {                #U
            String[] countryInitData = COUNTRY_INIT_DATA[i];
            Country country = new Country(countryInitData[0],                  #V
                                             countryInitData[1]);                     #V
            expectedCountryList.add(country);                                 #V
            if (country.getName().startsWith("A")) {                         #X
                expectedCountryListStartsWithA.add(country);                 #X
            }
        }
    }
}

```

In the listing above we do the following:

- We initialize an EntityManagerFactory (#A) and an EntityManager objects (#B). EntityManagerFactory provides instances of EntityManager for connecting to the same database, while an EntityManager is used to access a database in a particular application.
- We initialize an empty list of expected countries and an empty list of expected countries that start with 'A' (#C). We then declare the initialization data for the countries to be inserted (#D).
- We mark the setUp method with the @BeforeEach annotation, to be executed before each test (#E). Inside it, we initialize the expected countries list (#F) and we initialize the EntityManagerFactory and the EntityManager (#G).

- Within a transaction (#H), we initialize each country, one after the other (#I) and we persist the newly created country to the database (#J).
- In the `testCountryList` method, we initialize the countries list from the database by using the `EntityManager` and querying the `Country` entity using a JPQL SELECT (#K). JPQL (Java Persistence Query Language) is an object-oriented query language, independent of the platform. It is a part of the JPA (Java Persistence API) specification. Note that we need to spell "Country", as this is the exact name of the class, starting in upper case and having the other letters are in lower case. Then, we check that the list we have obtained is not null (#L), it has the expected size (#M) and that its content is the expected one (#N).
- In the `testCountryListStartsWithA` method, we initialize the countries list starting with 'A' from the database by using the `EntityManager` and querying the `Country` entity using a JPQL SELECT (#O). Then, we check that the list we have obtained is not null (#P), it has the expected size (#Q) and that its content is the expected one (#R).
- We mark the `dropDown` method with the `@AfterEach` annotation, to be executed after each test (#S). Inside it, we close the `EntityManagerFactory` and the `EntityManager` (#T).
- In the `initExpectedCountryLists` method we browse the countries initialization data (#U), create a `Country` object at each step (#V) and add it to the expected countries list (#W). If the name of the country starts with 'A', we also add it to the expected countries list whose names start with 'A' (#X).

The tests are successfully running, as shown in fig. 19.3.

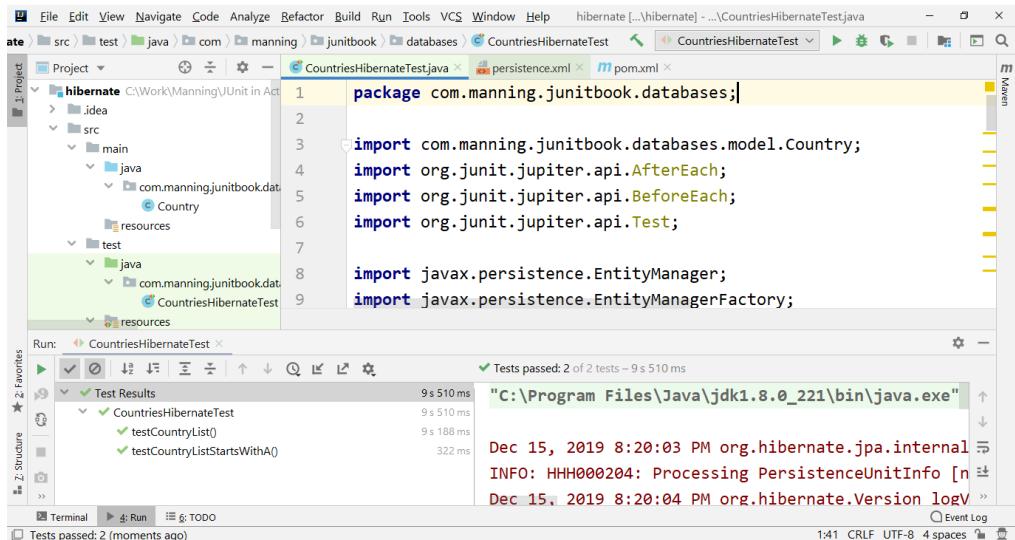


Figure 19.3 Successfully running the tests from the Hibernate application that checks the interaction with the COUNTRY table

The application is now accessing and testing the database through Hibernate. This comes with a few advantages:

- No more SQL code to be written inside the application. The developers are working only with Java code.
- No more mapping of the query result columns to object fields and vice-versa.
- Hibernate knows how to transform the operations with the implemented classes into vendor-specific SQL. So, if we change the underlying database, we will not touch the existing code, we'll only change the Hibernate configuration and the database dialect.

George will make one more step in considering alternatives to test the interaction with the database: combining Spring and Hibernate. We'll see this approach in the next section.

19.5 Testing a Spring Hibernate application

Hibernate is an object-relational mapping framework for Java. It provides the facilities for mapping an object-oriented domain model to relational database tables. Spring can take advantage of the Inversion of Control to simplify the database interaction tasks.

Integrating Hibernate and Spring, George will add the needed dependencies into the Maven pom.xml configuration file.

Listing 19.19 The Spring and Hibernate dependencies in the Maven pom.xml file

```
<dependency>
    <groupId>org.springframework</groupId>
```

```

<artifactId>spring-context</artifactId> #A
<version>5.2.1.RELEASE</version> #A
</dependency> #A
<dependency> #B
    <groupId>org.springframework</groupId> #B
    <artifactId>spring-orm</artifactId> #B
    <version>5.2.1.RELEASE</version> #B
</dependency> #B
<dependency> #C
    <groupId>org.springframework</groupId> #C
    <artifactId>spring-test</artifactId> #C
    <version>5.2.1.RELEASE</version> #C
</dependency> #C
<dependency> #D
    <groupId>org.hibernate</groupId> #D
    <artifactId>hibernate-core</artifactId> #D
    <version>5.4.9.Final</version> #D
</dependency> #D

```

In the listing above, we have added the following dependencies:

- `spring-context`, the dependency for the Spring Inversion of Control container (#A).
- `spring-orm`, as the application is still using Hibernate as ORM (Object Relational Mapping) framework to access the database. The control of working with connections, preparing and executing statements, processing exceptions is handled by Spring (#B).
- `spring-test`, the dependency that provides support for writing tests with the help of Spring and that is necessary to use the `SpringExtension` and the `@ContextConfiguration` annotation. (#C).
- The `hibernate-code` dependency, for the interaction with the database through Hibernate (#D).

George will make some changes to the `persistence.xml` file, the standard configuration for Hibernate. Only some minimal information will remain here, as the database access control will be handled by Spring.

Listing 19.20 The `persistence.xml` file

```

<persistence-unit name="manning.hibernate"> #A
    <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider> #B
    <class>com.manning.junitbook.databases.model.Country</class> #C
</persistence-unit>

```

In the listing above we do the following:

- We specify the persistence unit as `manning.hibernate` (#A). The `persistence.xml` file must define a persistence unit with a unique name in the currently scoped classloader.
- We specify the provider, meaning the underlying implementation of the JPA (Java Persistence API) `EntityManager` (#B). An `EntityManager` manages a set of persistent objects and has an API to insert new objects and read/update/delete the existing ones. In our case, the `EntityManager` is Hibernate.

- We define the entity class that is managed by Hibernate as the `Country` class from our application (#C).

So, George will move the database access configuration to the `application-context.xml` file that is configuring the Spring container.

Listing 19.21 The `application-context.xml` file

```

<tx:annotation-driven transaction-manager="txManager"/> #A
<bean id="dataSource" class= #B
      "org.springframework.jdbc.datasource.DriverManagerDataSource"> #B
        <property name="driverClassName" value="org.h2.Driver"/> #C
        <property name="url" value="jdbc:h2:mem:test;DB_CLOSE_DELAY=-1"/> #D
        <property name="username" value="sa"/> #E
        <property name="password" value="" /> #E
    </bean>

<bean id="entityManagerFactory" class= #F
      "org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean"> #F
        <property name="persistenceUnitName" value="manning.hibernate" /> #G
        <property name="dataSource" ref="dataSource"/> #H
        <property name="jpaProperties">
          <props>
            <prop key= #I
              "hibernate.dialect">org.hibernate.dialect.H2Dialect</prop> #I
            <prop key="hibernate.show_sql">true</prop> #J
            <prop key="hibernate.hbm2ddl.auto">create</prop> #K
          </props>
        </property>
    </bean>

<bean id="txManager" class= #L
      "org.springframework.orm.jpa.JpaTransactionManager"> #L
        <property name="entityManagerFactory" ref="entityManagerFactory" /> #M
        <property name="dataSource" ref="dataSource" /> #N
    </bean>

<bean class="com.manning.junitbook.databases.CountryService"/> #O

```

In the listing above we do the following:

- `<tx:annotation-driven>` tells the Spring context that we are using annotation-based transaction management configuration (#A).
- We are configuring the access to the data source (#B), by specifying the driver as H2, as this is the database type is use (#C). We specify the URL of the H2 database. In addition, the `DB_CLOSE_DELAY=-1` will keep the database open and the content of the in-memory database as long as the virtual machine is alive (#D).
- We specify the credentials to access the database – user and password (#E).
- We create an `EntityManagerFactory` bean (#F). We set its properties: the persistence unit name (as defined into the `persistence.xml` file) (#G), the data source (defined above) (#H), the SQL dialect for the generated query to `H2Dialect` (#I). We show the generated SQL query on the console (#J) and we create the database schema from the scratch every time when we execute the tests (#K).

- To process the annotation-based transaction configuration, a transaction manager bean needs to be created. We declare it (#L) and set its entity manager factory (#M) and data source (#N) properties.
- We declare a `CountryService` bean (#O), as we'll create this class into the code and group here the logic of the interaction with the database.

George creates the `CountryService` class that contains the logic of the interaction with the database.

Listing 19.22 The CountryService class

```
public class CountryService {

    @PersistenceContext
    private EntityManager em; #A #A

    public static final String[][] COUNTRY_INIT_DATA =
        { { "Australia", "AU" }, { "Canada", "CA" }, { "France", "FR" }, #B
          { "Germany", "DE" }, { "Italy", "IT" }, { "Japan", "JP" }, #B
          { "Romania", "RO" }, { "Russian Federation", "RU" }, #B
          { "Spain", "ES" }, { "Switzerland", "CH" }, #B
          { "United Kingdom", "UK" }, { "United States", "US" } }; #B

    @Transactional
    public void init() { #C
        for (int i = 0; i < COUNTRY_INIT_DATA.length; i++) { #D
            String[] countryInitData = COUNTRY_INIT_DATA[i];
            Country country = new Country(countryInitData[0], #D
                                         countryInitData[1]); #D
            em.persist(country); #D
        }
    }

    @Transactional
    public void clear() { #C
        em.createQuery("delete from Country c").executeUpdate(); #E
    }

    public List<Country> getAllCountries() {
        return em.createQuery("select c from Country c") #F
               .getResultList(); #F
    }

    public List<Country> getCountriesStartingWithA() {
        return em.createQuery(
            "select c from Country c where c.name like 'A%'") #G
               .getResultList(); #G
    }
}
```

In the listing above we do the following:

- We declare an `EntityManager` bean and we annotate it with `@PersistenceContext` (#A). The `EntityManager` is used to access a database and is created by the container using the information in the `persistence.xml`. To use it

at runtime, we simply need to request it be injected into one of our components, via `@PersistenceContext`.

- We then declare the initialization data for the countries to be inserted (#B).
- We annotate the `init` and `clear` methods as `@Transactional` (#C). We do this as these methods are modifying the content of the database, and all this kind of methods need to be executed within a transaction.
- We browse the initialization data for the countries, create each `Country` object and persist it within the database (#D).
- The `clear` method will delete all countries from the `Country` entity using a JPQL `DELETE` (#E). JPQL (*Java Persistence Query Language*) is a platform-independent object-oriented query language defined as part of the JPA (Java Persistence API) specification. Note that we need to spell “`Country`”, as this is the exact name of the class, starting in upper case and having the other letters are in lower case.
- The `getAllCountries` method will select all the countries from the `Country` entity using a JPQL `SELECT` (#F).
- The `getCountriesStartingWithA` method will select all the countries from the `Country` entity having names starting with ‘A’, using a JPQL `SELECT` (#G).

George will finally modify the `CountriesHibernateTest` class, testing the logic of the interaction with the database.

Listing 19.23 The CountriesHibernateTest class

```
@ExtendWith(SpringExtension.class)                                     #A
@ContextConfiguration("classpath:application-context.xml")          #B
public class CountriesHibernateTest {

    @Autowired                                              #C
    private CountryService countryService;                      #C

    private List<Country> expectedCountryList = new ArrayList<>();      #D
    private List<Country> expectedCountryListStartsWithA =           #D
        new ArrayList<>();                                         #D

    @BeforeEach                                              #E
    public void setUp() {                                       #F
        countryService.init();                                #F
        initExpectedCountryLists();                          #G
    }

    @Test
    public void testCountryList() {
        List<Country> countryList = countryService.getAllCountries();   #H
        assertNotNull(countryList);                            #I
        assertEquals(COUNTRY_INIT_DATA.length, countryList.size());     #J
        for (int i = 0; i < expectedCountryList.size(); i++) {         #K
            assertEquals(expectedCountryList.get(i), countryList.get(i)); #K
        }
    }
}
```

```

@Test
public void testCountryListStartsWithA() {
    List<Country> countryList =
        countryService.getCountryListStartingWithA(); #L
    assertNotNull(countryList); #L
    assertEquals(expectedCountryListStartsWithA.size(),
                countryList.size()); #N
    for (int i = 0; i < expectedCountryListStartsWithA.size(); i++) { #O
        assertEquals(expectedCountryListStartsWithA.get(i),
                    countryList.get(i)); #O
    } #O
} #P

@AfterEach
public void dropDown() {
    countryService.clear(); #Q
}

private void initExpectedCountryLists() {
    for (int i = 0; i < COUNTRY_INIT_DATA.length; i++) { #R
        String[] countryInitData = COUNTRY_INIT_DATA[i];
        Country country = new Country(countryInitData[0],
                                       countryInitData[1]); #S
        expectedCountryList.add(country); #S
        if (country.getName().startsWith("A")) { #S
            expectedCountryListStartsWithA.add(country); #T
        } #U
    } #U
} #G
}

```

In the listing above we do the following:

- We are annotating the test class to be extended with SpringExtension (#A). SpringExtension is used to integrate the Spring TestContext with the JUnit 5 Jupiter Test. This allows us to use other Spring annotations as well (e.g. @ContextConfiguration, @Transactional), but it also requires the usage of JUnit 5 and its annotations.
- We are also annotating the test class to look for the context configuration into the application-context.xml file from the classpath (#B).
- We declare and auto-wire a CountryService bean. This bean will be created and injected by the Spring container (#C).
- We initialize an empty list of expected countries and an empty list of expected countries that start with 'A' (#D).
- We mark the setUp method with the @BeforeEach annotation, to be executed before each test (#E). Inside it, we initialize the database content through the init method from the CountryService class (#F) and we initialize the expected countries list (#G).
- In the testCountryList method, we initialize the countries list from the database by using the getAllCountries method from the CountryService class (#H). Then, we check that the list we have obtained is not null (#I), it has the expected size (#J)

and that its content is the expected one (#K).

- In the `testCountryListStartsWithA` method, we initialize the countries list starting with 'A' from the database by using the `getCountriesStartingWithA` method from the `CountryService` class (#L). Then, we check that the list we have obtained is not null (#M), it has the expected size (#N) and that its content is the expected one (#O).
- We mark the `dropDown` method with the `@AfterEach` annotation, to be executed after each test (#P). Inside it, we clear the `COUNTRY` table content through the `clear` method from the `CountryService` class (#Q).
- In the `initExpectedCountryLists` method we browse the countries initialization data (#R), create a `Country` object at each step (#S) and add it to the expected countries list (#T). If the name of the country starts with 'A', we also add it to the expected countries list whose names start with 'A' (#U).

The tests are successfully running, as shown in fig. 19.4.

The screenshot shows the IntelliJ IDEA interface with the following details:

- Project View:** Shows the project structure under "spring-hibernate". The "CountryService.java" file is open in the editor.
- Editor:** Displays the Java code for the `testCountryListStartsWithA` test method.
- Run Tab:** Shows the results of the run: "Tests passed: 2 of 2 tests – 1 s 36 ms".
- Test Results:** A tree view showing the test results:
 - Test Results
 - └ CountriesHibernateTest
 - └ testCountryList()
 - └ testCountryListStartsWithA()
- Output Tab:** Displays the command-line output of the test execution.

Figure 19.4 Successfully running the tests from the Spring Hibernate application that checks the interaction with the `COUNTRY` table

The application is now accessing and testing the database through Spring Hibernate. This comes with a few advantages:

- No SQL code to be written inside the application. The developers are working only with Java code.
- No creation/opening/closing of the connections by ourselves.
- No exception process by ourselves.

- We have to take care mainly of the application context to be handled by Spring and to include the data source, transaction manager and entity manager factory configuration.
- Hibernate knows how to transform the operations with the implemented classes into vendor-specific SQL. So, if we change the underlying database, we will not touch the existing code, we'll only change the Hibernate configuration and the database dialect.

19.6 Comparing the approaches of testing database applications

We have followed George as he started with a simple JDBC application and moved it for use with Spring and Hibernate. We have demonstrated how things have changed and how each approach has simplified the way we test and we interact in general with the database. Our purpose was to analyze how each approach works and how the "database burden" can be reduced from the side of the developers.

We will summarize in table 19.1 the characteristics of each approach.

Application type	Characteristics
JDBC	SQL code needs to be written inside the tests No portability between databases Full control on what the application is doing Developer manual work for interacting with the database, e.g.: Create and open the connections Specify, prepare and execute the statement Iterate through the results Do the work for each iteration Process the exceptions Close the connection
Spring JDBC	SQL code needs to be written inside the tests No portability between databases Need to take care mainly of the application context configuration to be handled by Spring and of the row mapper Control on the queries that the application is executing against the database Reduces the manual work for interacting with the database: No creation/opening/closing the connections by ourselves No preparation and execution of the statements No processing of the exceptions
Hibernate	No SQL code inside the application Developers working only with Java code No mapping of the query result columns to object fields and vice-versa Portability between databases by changing the Hibernate configuration and the database

	dialect Database configuration handled through Java code
Spring Hibernate	No SQL code inside the application Developers working only with Java code No mapping of the query result columns to object fields and vice-versa Portability between databases by changing the Hibernate configuration and the database dialect Database configuration is handled by Spring, based on the information from the application context

Table 19.1 Comparison of the alternatives of working with a database application and testing it with JDBC, Spring JDBC, Hibernate, Spring Hibernate

We notice that the introduction of at least one framework as Spring or Hibernate into the application greatly simplifies the testing of the database and developing the application itself. As these are the most popular Java frameworks from today and they provide a lot of benefits (including the work and testing of the interaction with a database, as we have demonstrated), you may consider adopting them into your project (if you haven't done it already).

This chapter has focused on covering the alternatives of testing a database application with JUnit 5, to support you with your possible current project. The tests have covered only the insertions and selections operations, which are already providing comprehensive information about how to do it and how each alternative works in detail. The reader may extend the tests offered here to include the update and delete operations.

The next chapter will start the last part of the book, dedicated to the systematic development of applications with the help of JUnit 5. It will introduce one of the most widely spread development techniques from today: TDD (Test Driven Development).

19.7 Summary

This chapter has covered the following:

- Examining the database unit testing impedance mismatch, including as challenges the fact that: unit tests must exercise code in isolation; unit tests must be easy to write and run; unit tests must be fast to run.
- Implementing tests for a JDBC application, which requires SQL code to be written inside the tests and a lot of tedious work to be done by the developer: creating/opening/closing the connections to the database; specifying, preparing and executing the statements; handling exceptions.
- Implementing tests for a Spring JDBC application. This still requires SQL code to be written inside the tests, but it handles the Spring container the tasks of creating/opening/closing the connections to the database; specifying, preparing and executing the statements; handling exceptions.

- Implementing tests for a Hibernate application. No more SQL code is required, developers work only with the Java code, the application is portable to another database with minimum configuration changes. The database configuration is handled through the Java code.
- Implementing tests for a Spring Hibernate application. No SQL code is required, developers work only with the Java code, the application is portable to another database with minimum configuration changes. Additionally, the database configuration is handled by Spring, based on the information from the application context.

20

Test Driven Development with JUnit 5

This chapter covers:

- Introducing Test Driven Development
- Moving a non-TDD application to TDD
- Refactoring a TDD application
- Working TDD to implement new functionality in an application

TDD helps you to pay attention to the right issues at the right time so you can make your designs cleaner, you can refine your designs as you learn. TDD enables you to gain confidence in the code over time.

- Kent Beck

This chapter is teaching Java developers on how to develop safe and flexible applications. We'll discover together that Test Driven Development is a technique that will greatly increase the development speed and that will remove much of the debugging nightmare - everything with the help of JUnit 5 and its features.

We will point out the main ideas of Test Driven Development and we'll apply them in developing a Java application implementing the business logic for the management of flights and passengers and following a set of policies. The focus will be on clearly explaining the TDD technique and to prove its benefits by following how to put it in practice, step by step.

20.1 Introducing Test Driven Development

Test Driven Development

Test-Driven Development is a programming practice that instructs developers to write first the tests, then to write the code that will make the software pass the newly introduced tests. Then, the developer should examine the code and refactor it to clean up any mess or improve things. TDD is looking for “clean code that works.”

TDD repeats a short development cycle: first, requirements are converted into test cases; then the program is modified to make the tests pass. This is different from traditional software development, where code may be added without having to verify that it meets requirements.

The development of this technique is attributed to the American software engineer Kent Beck. TDD supports simple designs and inspires safety.

In a classical approach, developing a program will mean that we are supposed to write the code, then do some testing by observing its behavior. So, the conventional development cycle goes something like this:

[code, test, (repeat)]

Test Driven Development makes a surprising variation:

[test, code, (repeat)]

The test drives the design and becomes the first client of the method. This is different from the traditional software development, where code may be added without having verified that it meets requirements.

Among the benefits of TDD we will enumerate:

- We'll write code that will be driven by clear goals, and we make sure that we'll address exactly what our application needs to do.
- Introducing new functionality will be much faster. On one side, tests will drive us to implement the code that will do what it is supposed to do. On the other side, they will prevent us from introducing bugs into the well working existing code.
- Tests will act as documentation for the application. We may follow them and understand what problems our code is supposed to solve.

We said that TDD changes the development cycle this way

[test, code, (repeat)].

In fact, a key step is left out. It should go rather like this:

[test, code, refactor, (repeat)].

Refactoring defines the process of modifying a software system so that it does not impact its external behavior, but it improves its internal structure. In order to make sure that the external behavior is not affected, we need to rely on the tests.

The essential approach of TDD involves:

1. Write a failing test before writing new code
2. Write the smallest piece of code that will make the new test pass

Once a developer has received some specifications, he will have to first understand them before being able to put them into the code. Any living application will receive requests for implementing new functionality. In order to be able to implement them, we have to answer the question: what behavior must we follow? What if we'll first implement the test that will show us **WHAT** we have to do, then we may think about **HOW** we have to do it? This is, in fact, one of the fundamental principles of Test Driven Development.

Then, when we work on a new application, we would probably like to get our hands on some documentation in order to easily jump into it. At least, we need to understand the fundamental idea: what is this software supposed to do? But if we need to check what some class or its methods intend to do, our choices are: to read some documentation, or to look for sample code that already invokes it? Most programmers prefer to work with the code, and this is pretty natural. Well-written unit tests do exactly this: they invoke our code and, consequently, they provide a working specification of the functionality. As a result, working TDD will effectively help to build a significant part of the technical documentation.

20.2 Introducing the flights management application

Tested Data Systems Inc. is an outsourcing company creating software projects for different companies. At Tested Data Systems, one of the projects under development for one of the customers is a flights management application. The application is able to create and setup flights, and add and remove passengers to the flights.

We'll follow scenarios close to the everyday work: we'll start, as input, with a non-TDD working application that is supposed to do a few things, as following the company policies for regular and VIP passengers. But we have to understand it and make sure that it is really implementing the expected operations. So, we have to cover the existing code with unit tests. We'll start to write them one by one. After covering most of the code with tests, our application may be well understood and we may have some certainty that it is working well. Then, the new challenge is adding the new functionality, first by understanding what needs to be done, writing tests that initially fail, then writing the code that fixes the tests. We'll enter this working cycle that is one of the foundations of TDD and that we are going to demonstrate.

John is joining the development of the flights management application for one of the customers of Tested Data Systems. It is a Java application built with the help of Maven.

The software must maintain a policy regarding adding and removing a passenger to a flight. The flights may be of a few types - economy and business are known at the time, but we may add other types at a later time, depending on the requirements from the customer. If the flight is an economy one, both VIP passengers and regular ones may be added to it. If the flight is a business one, only VIP passengers may be added to it (fig. 20.1).

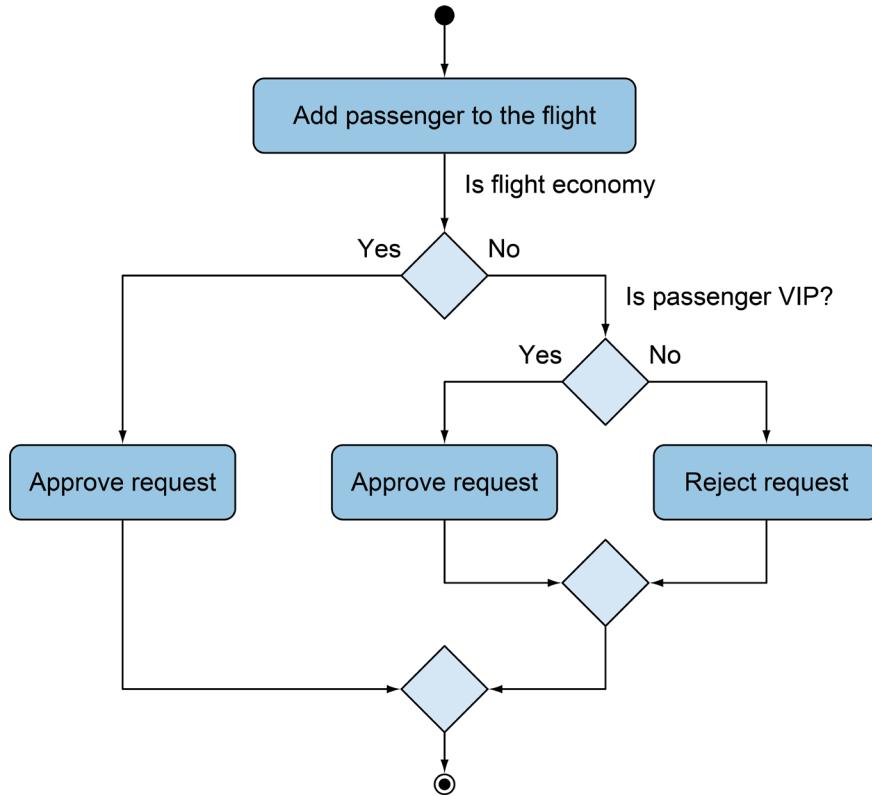


Figure 20.1 The business logic of adding passengers to a flight: if the flight is a business one, only VIP passengers may be added to it. Any passenger can be added to an economy flight

And there is also a policy for removing a passenger from the flight, which also involves answering yes/no questions. A regular passenger may be removed from a flight, a VIP passenger cannot be removed (fig. 20.2). As we see from these two activity diagrams, the initial business logic focuses on decision making.

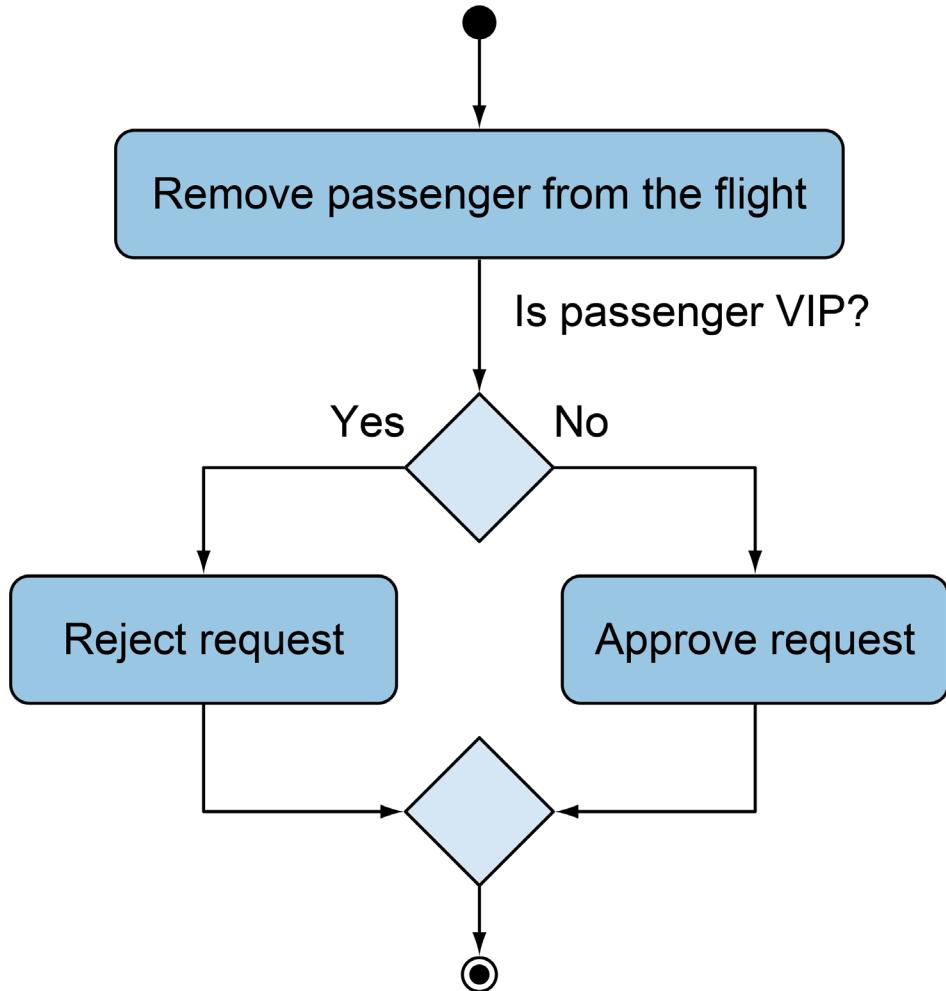


Figure 20.2 The business logic of removing passengers from a flight: only regular passengers are allowed to be removed

Let's have a look at the first design of this application (fig. 20.3). We maintain a field called `flightType` inside the `Flight` class. Its value determines the behavior of the `addPassenger` and `removePassenger` methods. The programmer will have to focus on decision making at the level of the code of these two methods.

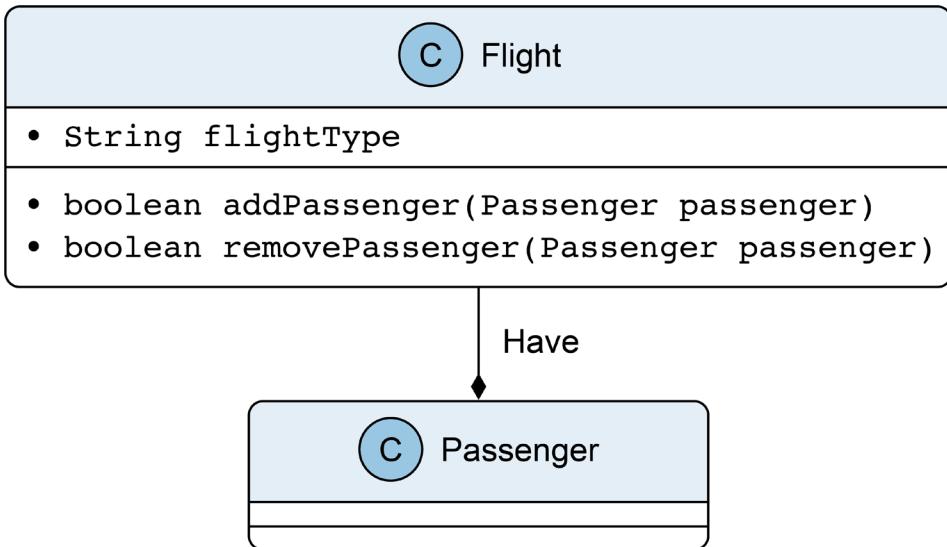


Figure 20.3 The class diagram of the flights management application: the flight type is kept as a field inside the Flight class

Listing 20.1 shows how the `Passenger` class is looking like.

Listing 20.1 The Passenger class

```

public class Passenger {

    private String name;                                #A
    private boolean vip;                               #B

    public Passenger(String name, boolean vip) {        #C
        this.name = name;
        this.vip = vip;
    }                                              #C

    public String getName() {                           #D
        return name;
    }                                              #D

    public boolean isVip() {                           #E
        return vip;
    }                                              #E
}

```

In the listing above we are doing the following:

- The `Passenger` class will contain a `name` field (#A), together with a getter for it (#D).
- It also contains a `vip` field (#B), together with a getter for it (#E).
- The constructor of the `Passenger` class is initializing the `name` and the `vip` fields

(#C).

Listing 20.2 shows how the Flight class looks like.

Listing 20.2 The Flight class

```
public class Flight {  
  
    private String id; #A  
    private List<Passenger> passengers = new ArrayList<Passenger>(); #B  
    private String flightType; #C  
  
    public Flight(String id, String flightType) { #D  
        this.id = id; #D  
        this.flightType = flightType; #D  
    } #D  
  
    public String getId() { #E  
        return id; #E  
    } #E  
  
    public List<Passenger> getPassengersList() { #F  
        return Collections.unmodifiableList(passengers); #F  
    } #F  
  
    public String getFlightType() { #G  
        return flightType; #G  
    } #G  
  
    public boolean addPassenger(Passenger passenger) {  
        switch (flightType) { #H  
            case "Economy": #I  
                return passengers.add(passenger); #I  
            case "Business": #J  
                if (passenger.isVip()) { #J  
                    return passengers.add(passenger); #J  
                } #J  
                return false; #J  
            default: #K  
                throw new RuntimeException("Unknown type: " + flightType); #K  
        }  
    } #K  
  
    public boolean removePassenger(Passenger passenger) {  
        switch (flightType) { #L  
            case "Economy": #M  
                if (!passenger.isVip()) { #M  
                    return passengers.remove(passenger); #M  
                } #M  
                return false; #M  
            case "Business": #N  
                return false; #N  
            default: #O  
                throw new RuntimeException("Unknown type: " + flightType); #O  
        } #O  
    } #O
```

```
}
```

In the listing above we are doing the following:

- The `Flight` class will contain an identifier (#A), together with a getter for it (#E), a list of passengers initialized as an empty list (#B) together with a getter for it (#F) and a flight type (#C) together with a getter for it (#G).
- The constructor of the `Flight` class is initializing the `id` and the `flightType` fields (#D).
- The `addPassenger` method is checking the flight type (#H). If it is an economy one, it will allow adding him to the flight (#I). If it is a business one, it will allow only adding VIP passengers to the flight (#J). Otherwise (if the flight is neither an economy nor a business one), it will throw an exception, as it cannot handle an unknown flight type (#K).
- The `removePassenger` method is checking the flight type (#L). If it is an economy one, it will allow adding removing only a regular passenger from the flight (#M). If it is a business one, it will not allow removing passengers from the flight (#N). Otherwise (if the flight is neither an economy nor a business one), it will throw an exception, as it cannot handle an unknown flight type (#O).

The application has no tests yet. The initial developers have written some code where they were simply following the execution and comparing it with their expectations. There is an `Airport` class including the main method that acts as a client of the `Flight` and `Passenger` classes and works with the different types of flights and passengers (listing 20.3).

Listing 20.3 The Airport class including the main method

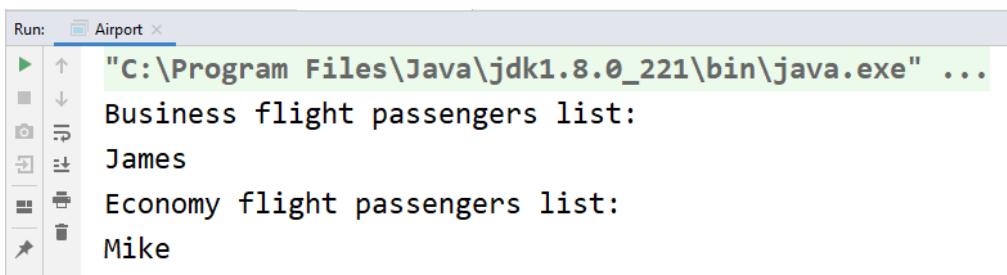
```
public class Airport {  
  
    public static void main(String[] args) {  
        Flight economyFlight = new Flight("1", "Economy"); #A  
        Flight businessFlight = new Flight("2", "Business"); #A  
  
        Passenger james = new Passenger("James", true); #B  
        Passenger mike = new Passenger("Mike", false); #B  
  
        businessFlight.addPassenger(james); #C  
        businessFlight.removePassenger(james); #C  
        businessFlight.addPassenger(mike); #D  
        economyFlight.addPassenger(mike); #E  
  
        System.out.println("Business flight passengers list:"); #F  
        for (Passenger passenger: businessFlight.getPassengersList()) { #F  
            System.out.println(passenger.getName()); #F  
        } #F  
  
        System.out.println("Economy flight passengers list:"); #G  
        for (Passenger passenger: economyFlight.getPassengersList()) { #G  
            System.out.println(passenger.getName()); #G  
        } #G
```

```
    }  
}
```

In the listing above we are doing the following:

- We are initializing an economy flight and a business flight (#A). We are also initializing James as a VIP passenger and Mike as a regular passenger (#B).
- We try to add and remove James from the business flight (#C), then we try to add Mike to the business flight (#D) and to the economy flight (#E).
- We print the passengers on the business flight (#F) and on the economy flight (#G).

The result of running this program is shown in fig. 20.4. James, a VIP passenger, has been added to the business flight, but we could not remove him. Mike, a regular passenger, could not be added to the business flight, but we were able to add him to the economy flight.



```
Run: Airport <input>  
"C:\Program Files\Java\jdk1.8.0_221\bin\java.exe" ...  
Business flight passengers list:  
James  
Economy flight passengers list:  
Mike
```

Figure 20.4 The result of running the non-TDD flights management application: the VIP passenger has been added to the business flight, the regular passenger has been added to the economy flight

So far, things are working as expected, following the policy that we have previously defined. John is satisfied with the way the application works so far, but he needs to develop it further. In order to build a reliable application and to be able to easily and safely understand and implement the business logic, John considers first moving it to the Test Driven Development approach.

20.3 Preparing the flights management application for TDD

In order to move the flights management application to the Test Driven Development approach, John will first cover with JUnit 5 tests the existing business logic. He will add the JUnit 5 dependencies we are already familiar with (`junit-jupiter-api` and `junit-jupiter-engine`) to the Maven `pom.xml` file (listing 20.4):

Listing 20.4 The JUnit 5 dependencies added to the pom.xml file

```
<dependencies>  
  <dependency>  
    <groupId>org.junit.jupiter</groupId>  
    <artifactId>junit-jupiter-api</artifactId>
```

```

<version>5.6.0</version>
<scope>test</scope>
</dependency>
<dependency>
<groupId>org.junit.jupiter</groupId>
<artifactId>junit-jupiter-engine</artifactId>
<version>5.6.0</version>
<scope>test</scope>
</dependency>
</dependencies>

```

Inspecting the business logic from figures 20.1 and 20.2, John understands that he has to check the add/remove passenger scenarios by providing tests for two flight types and two passenger types. So, he will have 2 flight types multiplied by 2 passenger types, this means 4 tests in total. For each of the tests, he has to verify the possible add and remove operations.

John follows the business logic for an economy flight and transposes it into the tests from listing 20.5.

Listing 20.5 Testing the business logic of the economy flight

```

public class AirportTest {

    @DisplayName("Given there is an economy flight")                      #A
    @Nested                                                               #A
    class EconomyFlightTest {                                              #A

        private Flight economyFlight;                                         #B

        @BeforeEach
        void setUp() {                                                       #B
            economyFlight = new Flight("1", "Economy");                      #B
        }                                                                    #B

        @Test
        public void testEconomyFlightRegularPassenger() {                     #C
            Passenger mike = new Passenger("Mike", false);                   #C

            assertEquals("1", economyFlight.getId());                          #D
            assertEquals(true, economyFlight.addPassenger(mike));              #E
            assertEquals(1, economyFlight.getPassengersList().size());          #E
            assertEquals("Mike",                                         #E
                        economyFlight.getPassengersList().get(0).getName());     #E

            assertEquals(true, economyFlight.removePassenger(mike));           #F
            assertEquals(0, economyFlight.getPassengersList().size());          #F
        }                                                                    #F

        @Test
        public void testEconomyFlightVipPassenger() {                         #G
            Passenger james = new Passenger("James", true);                  #G

            assertEquals("1", economyFlight.getId());                          #H
            assertEquals(true, economyFlight.addPassenger(james));             #I
            assertEquals(1, economyFlight.getPassengersList().size());          #I
            assertEquals("James",                                         #I
                        economyFlight.getPassengersList().get(0).getName());     #I
        }                                                                    #I
    }
}

```

```
        assertEquals(false, economyFlight.removePassenger(james));    #J
        assertEquals(1, economyFlight.getPassengersList().size());      #J
    }
}
```

In the listing above we are doing the following:

- We have declared a nested test class `EconomyFlightTest` and we are labeling with "Given there is an economy flight" with the help of the `@DisplayName` annotation (#A).
 - We are declaring an economy flight and we are initializing it before the execution of each test (#B).
 - When testing how the economy flight works with a regular passenger, we are creating Mike as a regular passenger (#C). Then, we are checking the id of the flight (#D), the fact that we can add Mike on the economy flight and that we can find him there (#E) and the fact that we can remove Mike from the economy flight and that he is no longer there (#F).
 - When testing how the economy flight works with a VIP passenger, we are creating James as a VIP passenger (#G). Then, we are checking the id of the flight (#H), the fact that we can add James on the economy flight and that we can find him there (#I) and the fact that we cannot remove James from the economy flight and that he is still there (#J).

John follows the business logic for a business flight and transposes it into the tests from listing 20.6.

Listing 20.6 Testing the business logic of the business flight

```
public class AirportTest {  
    [...]  
  
        @DisplayName("Given there is a business flight")  
        @Nested  
        class BusinessFlightTest {  
            private Flight businessFlight;  
  
            @BeforeEach  
            void setUp() {  
                businessFlight = new Flight("2", "Business");  
            }  
  
            @Test  
            public void testBusinessFlightRegularPassenger() {  
                Passenger mike = new Passenger("Mike", false);  
  
                assertEquals(false, businessFlight.addPassenger(mike));  
                assertEquals(0, businessFlight.getPassengersList().size());  
                assertEquals(false, businessFlight.removePassenger(mike));  
                assertEquals(0, businessFlight.getPassengersList().size());  
            }  
        }  
    }  
}
```

```

    }

    @Test
    public void testBusinessFlightRegularPassenger() {
        Passenger mike = new Passenger("Mike", false);
        businessFlight.addPassenger(mike); #D
        assertEquals(0, businessFlight.getPassengersList().size()); #E
        assertEquals(false, businessFlight.removePassenger(mike)); #F
        assertEquals(0, businessFlight.getPassengersList().size()); #G
    }
}

}

}

```

In the listing above we are doing the following:

- We have declared a nested test class `BusinessFlightTest` and we are labeling with “Given there is a business flight” with the help of the `@DisplayName` annotation (#A).
- We are declaring a business flight and we are initializing it before the execution of each test (#B).
- When testing how the business flight works with a regular passenger, we are creating Mike as a regular passenger (#C). Then, we are checking the fact that we cannot add Mike on the business flight (#D) and the fact that trying to remove him from the business flight also has no effect (#E).
- When testing how the business flight works with a VIP passenger, we are creating James as a VIP passenger (#F). Then, we are checking the fact that we can add James on the business flight and that we can find him there (#G) and the fact that we cannot remove James from the business flight and that he is still there (#H).

If we run the tests with coverage from within IntelliJ IDEA, we get the results in fig. 20.5. For more details about test coverage and how to run tests with coverage from IntelliJ IDEA, you can revisit chapter 6.

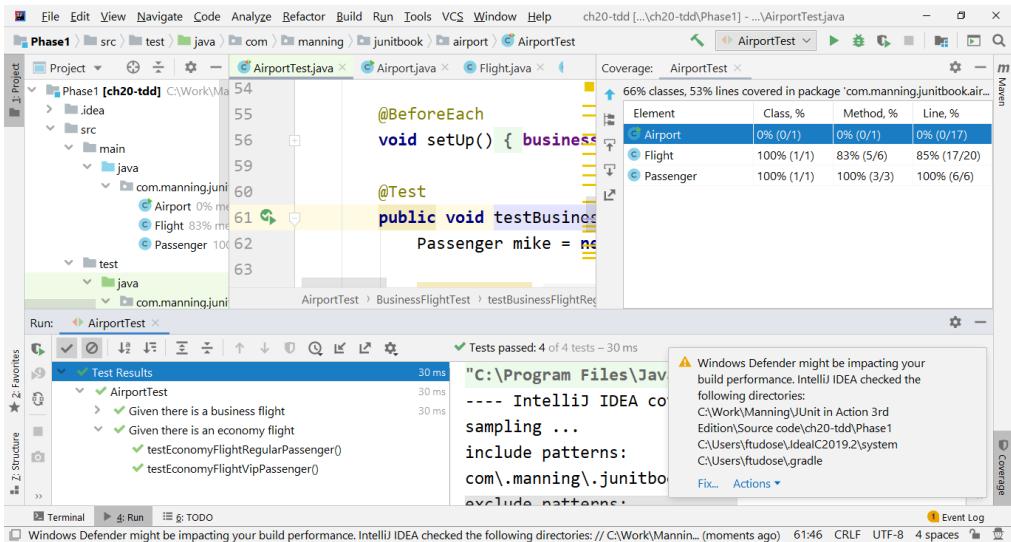


Figure 20.5 The result of running the economy flight and business flight tests with coverage using IntelliJ IDEA: the `Airport` class is uncovered, the `Flight` class has coverage of less than 100%

John has successfully verified the functionality of the application that he has joined, by writing tests for all the scenarios that result from the business logic (fig. 20.1 and 20.2). It is a possible real-life situation that you start with an application having no tests and you would like to move to TDD. Before that, you will have to test the application as it is.

John's work also provides additional conclusions to him. The `Airport` class is not tested – it served as a client for the `Passenger` and `Flight` classes. The tests are now serving as clients, so `Airport` can be removed. Then, the code coverage is not yet 100%. The `getFlightType` method is not used, and the default case, when a flight is neither of type "Economy" nor of type "Business" is not covered. This suggests to John the need for refactoring the application, to remove the situations that are not in use. He is confident to do this now as his application is now covered with tests and, as we were saying, TDD also enables one to gain confidence in the code over time.

20.4 Refactoring the flights management application

What John has already remarked is that the not executed lines of code are the ones related to the usage of the `flightType` field. And the default case here will never be executed, as the flight type is expected to be either "Economy" or "Business". But these default alternatives were needed as the code will not compile otherwise. Cannot we still get rid of them, by doing some refactoring and replacing the conditional with polymorphism?

The key to refactoring is to move the design to use polymorphism instead of procedural style conditional code. With polymorphism (the ability of one object to pass more than one IS-

A tests), the method you are calling is not determined at compile-time, but at runtime, depending on the effective object type. For details, you may revisit chapter 6.

The principle in action here is called "the open/closed principle" (fig. 20.6). Practically, it means that the left-hand-side design will require changes to our existing class each time when we add a new flight type. These changes may reflect in each conditional decision made based on the flight type. And, besides this, we were forced to rely on the `flightType` field and to introduce the unexecuted default cases.

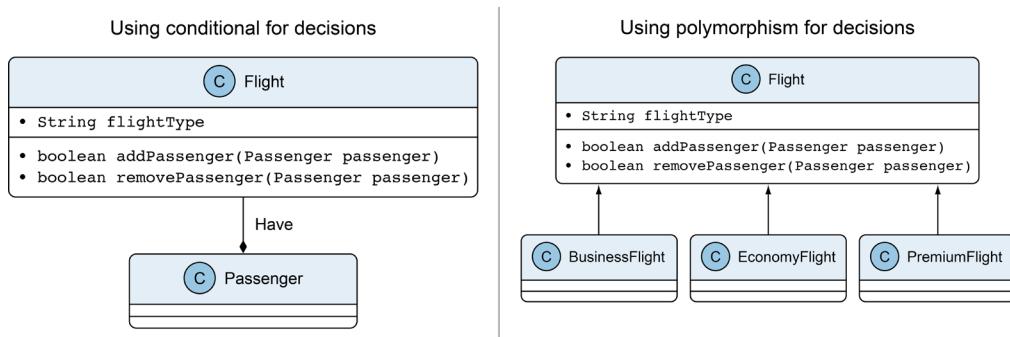


Figure 20.6 Refactoring the flights management application by replacing the conditional with polymorphism – the `flightType` field is removed, a hierarchy of classes is introduced

With the design from the right-hand side - as refactored by replacing conditional with polymorphism - you do not need any `flightType` evaluation and any default value into the `switch` instructions from listing 20.2. You may even add a new type - let's anticipate a little and call it `PremiumFlight` - by simply extending the base class and defining its behavior inside it. According to "the open/closed principle", the hierarchy will be open for extensions (we may easily add new classes) but closed for modifications (the existing classes, starting with the `Flight` base class, will not be modified).

John will, of course, ask himself: how can I be sure I am doing the right thing and I am not affecting an already well-working functionality? The answer is that tests will provide the safety that the already implemented functionality is untouched. The benefits of the TDD approach simply pop-up!

The refactoring will be achieved by keeping the base `Flight` class (listing 20.7) and, for each conditional type, one separate class to extend it. We may change `addPassenger` and `removePassenger` as abstract methods and delegate their effective implementation to the subclasses. The `flightType` field does not have significance any longer and will be removed.

Listing 20.7 The abstract Flight class, the basis of the hierarchy

```

public abstract class Flight { #A
    private String id;
    List<Passenger> passengers = new ArrayList<Passenger>(); #B
}

```

```

public Flight(String id) {
    this.id = id;
}

public String getId() {
    return id;
}

public List<Passenger> getPassengersList() {
    return Collections.unmodifiableList(passengers);
}

public abstract boolean addPassenger(Passenger passenger); #C

public abstract boolean removePassenger(Passenger passenger); #C

}

```

In the listing above, we have made the following changes to the previously existing `Flight` class:

- We have declared the class as abstract, making it the basis of the flights hierarchy (#A).
- We have made the `passengers` list package-private, allowing it to be directly inherited by the sub-classes in the same package (#B).
- We have declared `addPassenger` and `removePassenger` as abstract methods, delegating their implementation to the sub-classes (#C).

We'll introduce one class `EconomyFlight` that extends `Flight` and implements the inherited `addPassenger` and `removePassenger` abstract methods (listing 20.8).

Listing 20.8 The `EconomyFlight` class, extending the abstract `Flight` class

```

public class EconomyFlight extends Flight { #A

    public EconomyFlight(String id) { #B
        super(id);
    } #B

    @Override #C
    public boolean addPassenger(Passenger passenger) { #C
        return passengers.add(passenger);
    } #C

    @Override #D
    public boolean removePassenger(Passenger passenger) { #D
        if (!passenger.isVip()) { #D
            return passengers.remove(passenger);
        } #D
        return false;
    } #D

}

```

In the listing above, we are doing the following:

- We have declared the `EconomyFlight` class extending the `Flight` abstract class (#A) and we are creating a constructor calling the constructor of the superclass (#B).
- We are implementing the `addPassenger` method according to the business logic: we simply add a passenger to an economy flight with no restrictions (#C).
- We are implementing the `removePassenger` method according to the business logic: we are allowed to remove a passenger from a flight only if he is not a VIP (#D).

We'll also introduce one class `BusinessFlight` that extends `Flight` and implements the inherited `addPassenger` and `removePassenger` abstract methods (listing 20.9).

Listing 20.9 The BusinessFlight class, extending the abstract Flight class

```
public class BusinessFlight extends Flight { #A

    public BusinessFlight(String id) { #B
        super(id);
    } #B

    @Override #B
    public boolean addPassenger(Passenger passenger) { #C
        if (passenger.isVip()) { #C
            return passengers.add(passenger); #C
        }
        return false; #C
    } #C

    @Override #C
    public boolean removePassenger(Passenger passenger) { #D
        return false; #D
    } #D

}
```

In the listing above, we are doing the following:

- We have declared the `BusinessFlight` class extending the `Flight` abstract class (#A) and we are creating a constructor calling the constructor of the super class (#B).
- We are implementing the `addPassenger` method according to the business logic: we can only add a VIP passenger to a business flight (#C).
- We are implementing the `removePassenger` method according to the business logic: we are not allowed to remove a passenger from a business flight (#D).

Refactoring by replacing the conditional with polymorphism we immediately see that the methods now look much shorter and clearer, not cluttered with decision making. Also, we are not forced to treat the previously existing default case that was never expected and where we were throwing an exception. Of course, the refactoring and the API changing will propagate into the tests, which will be a little changed, as in listing 20.10.

Listing 20.10 The refactoring propagation into the AirportTest class

```
public class AirportTest {

    @DisplayName("Given there is an economy flight")
    @Nested
    class EconomyFlightTest {
        private Flight economyFlight;

        @BeforeEach
        void setUp() {
            economyFlight = new EconomyFlight("1"); #A
        }
        [...]
    }

    @DisplayName("Given there is a business flight")
    @Nested
    class BusinessFlightTest {
        private Flight businessFlight;

        @BeforeEach
        void setUp() {
            businessFlight = new BusinessFlight("2"); #B
        }
        [...]
    }
}
```

In listing 20.10 we are replacing the previous Flight instantiations with the ones of EconomyFlight (#A) and BusinessFlight (#B).

We have also removed the Airport class that served as a client for the Passenger and Flight classes – it is no longer needed, after having introduced the tests. It was previously serving to declare the main method where we were creating different types of flights and different types of passengers and making them act together. If we run the tests now, we see that the code coverage is 100% (fig. 20.7). So, refactoring the TDD application has helped us both to improve the quality of the code and to increase the testing code.

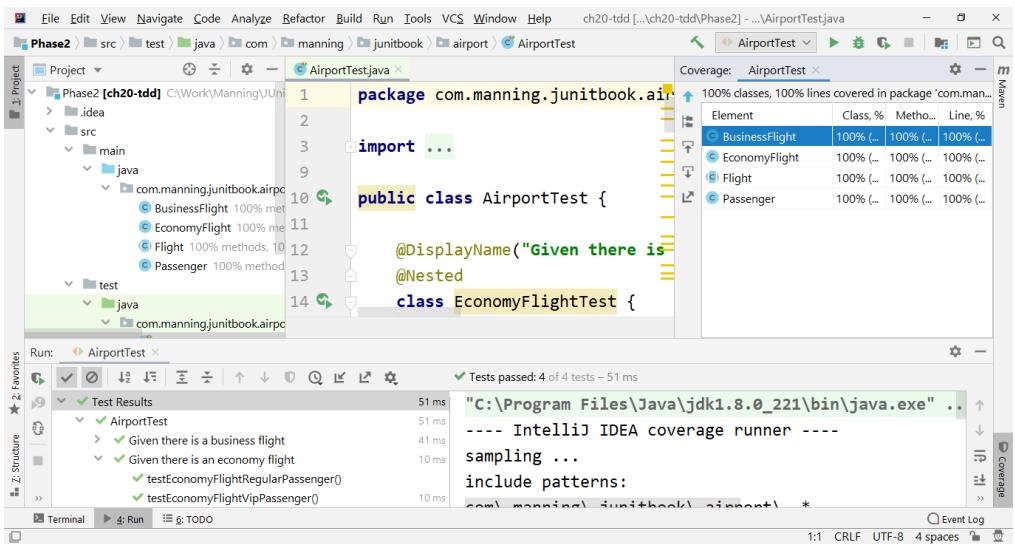


Figure 20.7 Running the economy flight and business flight tests after refactoring the flights management application will bring to 100% code

John has covered the flights management application with tests and has refactored it, having a better code quality and obtaining a 100% code coverage. It is time now for him to start introducing new features by working TDD!

20.5 Introducing new features by working TDD

After moving the software to TDD and refactoring it, John will then be responsible for the implementation of new features required by the customer, which is extending the application policies.

20.5.1 Adding a premium flight

The first new feature that John will implement is a new flight type, Premium, and a policy concerning this flight type. So, as usual, we have a policy for adding a passenger to the flight: if he is a VIP, he should be added to it, otherwise, the request must be rejected (fig. 20.8).

And there is a removing policy as well: if required, a passenger may be removed from a flight (fig. 20.9). We are sorry, you may be an important person, but we still have some rules and restrictions, and we may be forced as well to remove you from a flight.

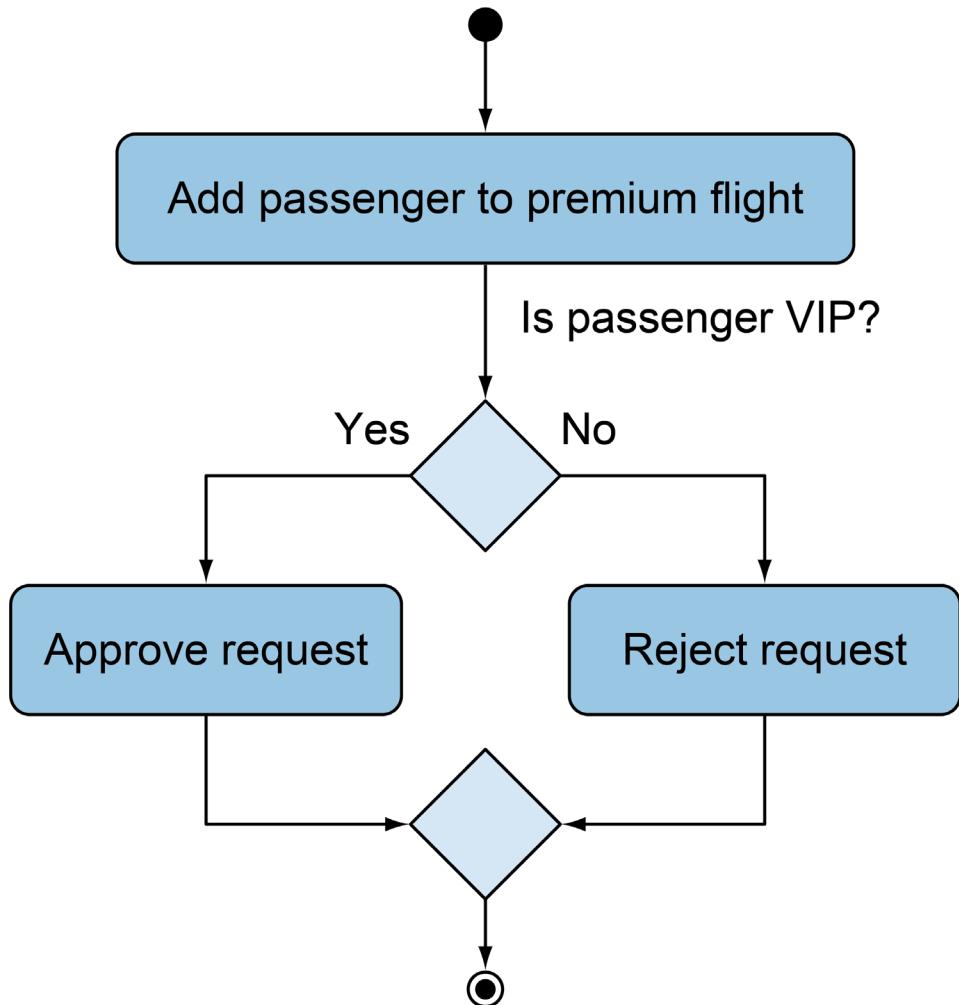


Figure 20.8 The extended business logic of adding a passenger to a premium flight: only VIP passengers are allowed to join

Remove passenger from premium flight

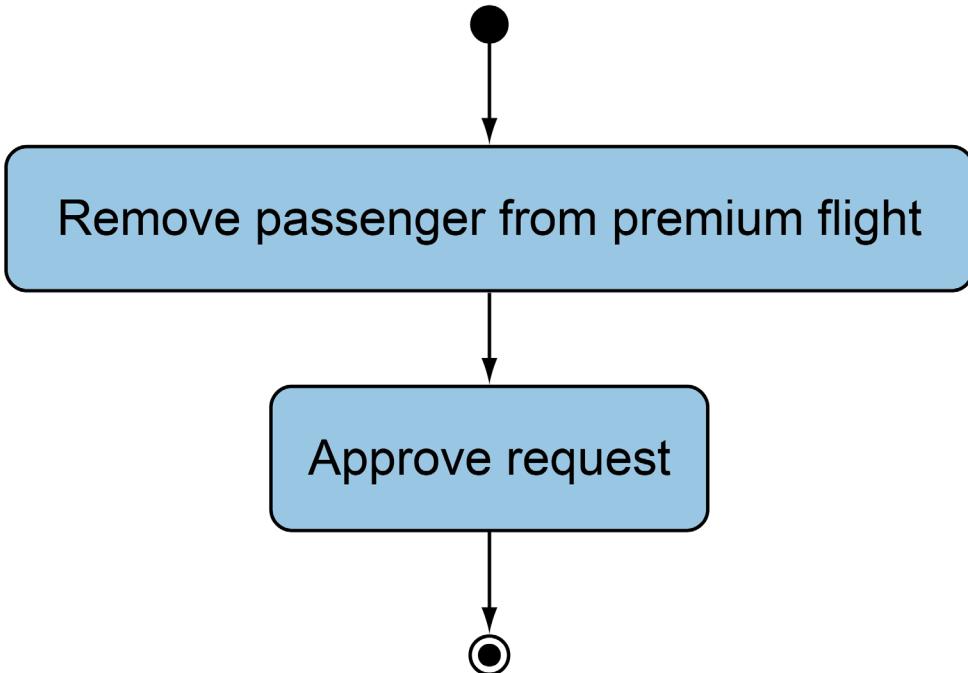


Figure 20.9 The extended business logic of removing a passenger: any type of passenger may be removed from a premium flight

John realizes that this new feature has similarities to the previous ones. He would like to take even more advantage of working TDD style and do more refactoring, to the tests this time. This is in the spirit of the "Rule of Three", stated by Don Roberts.

The Rule of Three

The first time you do something, you just do it. The second time you do something similar, you wince at the duplication, but you do the duplicate thing anyway. The third time you do something similar, you refactor.
So, three strikes and you refactor.

So, John considers that, after receiving the requirement for the implementation of this third flight type, it is time to do some more grouping of the already existing tests and to then implement the premium flight requirement in a similar way. Listing 20.11 shows how the refactored `AirportTest` class looks like before moving to the work for the premium flight.

Listing 20.11 The refactored AirportTest class

```
public class AirportTest {  
  
    @DisplayName("Given there is an economy flight")  
    @Nested  
    class EconomyFlightTest {  
  
        private Flight economyFlight; #A  
        private Passenger mike; #A  
        private Passenger james; #A  
  
        @BeforeEach  
        void setUp() {  
            economyFlight = new EconomyFlight("1"); #B  
            mike = new Passenger("Mike", false); #B  
            james = new Passenger("James", true); #B  
        }  
  
        @Nested  
        @DisplayName("When we have a regular passenger") #C  
        class RegularPassenger {  
  
            @Test  
            @DisplayName(  
                "Then you can add and remove him from an economy flight") #D  
            public void testEconomyFlightRegularPassenger() {  
                assertAll(  
                    "Verify all conditions for a regular passenger  
                     and an economy flight", #E  
                    () -> assertEquals("1", economyFlight.getId()), #E  
                    () -> assertEquals(true, #E  
                        economyFlight.addPassenger(mike)), #E  
                    () -> assertEquals(1, #E  
                        economyFlight.getPassengersList().size()), #E  
                    () -> assertEquals("Mike", #E  
                        economyFlight.getPassengersList()  
                            .get(0).getName()), #E  
                    () -> assertEquals(true, #E  
                        economyFlight.removePassenger(mike)), #E  
                    () -> assertEquals(0, #E  
                        economyFlight.getPassengersList().size()) #E  
                );  
            }  
        }  
  
        @Nested  
        @DisplayName("When we have a VIP passenger") #C  
        class VipPassenger {  
            @Test  
            @DisplayName("Then you can add him but  
                         cannot remove him from an economy flight") #D  
            public void testEconomyFlightVipPassenger() {  
                assertAll("Verify all conditions for a VIP passenger  
                     and an economy flight", #E  
                    () -> assertEquals("1", economyFlight.getId()), #E  
                    () -> assertEquals(true, #E  
                        economyFlight.addPassenger(james)), #E  
                    () -> assertEquals(1, #E  
                        economyFlight.getPassengersList().size()) #E  
                );  
            }  
        }  
    }  
}
```

```

        economyFlight.getPassengersList().size(),
        () -> assertEquals("James", #E
            economyFlight.getPassengersList().get(0).getName()), #E
        () -> assertEquals(false, #E
            economyFlight.removePassenger(james)), #E
        () -> assertEquals(1, #E
            economyFlight.getPassengersList().size()) #E
    );
}

}

}

@DisplayName("Given there is a business flight")
@Nested
class BusinessFlightTest {
    private Flight businessFlight; #A
    private Passenger mike; #A
    private Passenger james; #A

    @BeforeEach
    void setUp() {
        businessFlight = new BusinessFlight("2"); #B
        mike = new Passenger("Mike", false); #B
        james = new Passenger("James", true); #B
    }

    @Nested
    @DisplayName("When we have a regular passenger") #C
    class RegularPassenger { #C

        @Test
        @DisplayName("Then you cannot add or remove him
                    from a business flight") #D
        public void testBusinessFlightRegularPassenger() { #E
            assertAll("Verify all conditions for a regular passenger
                        and a business flight", #E
                () -> assertEquals(false, #E
                    businessFlight.addPassenger(mike)), #E
                () -> assertEquals(0, #E
                    businessFlight.getPassengersList().size()), #E
                () -> assertEquals(false, #E
                    businessFlight.removePassenger(mike)), #E
                () -> assertEquals(0, #E
                    businessFlight.getPassengersList().size())
            );
        }
    }

    @Nested
    @DisplayName("When we have a VIP passenger") #C
    class VipPassenger { #C

        @Test
        @DisplayName("Then you can add him but cannot remove him
                    from a business flight") #D
        public void testBusinessFlightVipPassenger() { #E
            assertAll("Verify all conditions for a VIP passenger
                        and a business flight", #E

```

```
        () -> assertEquals(true,
            businessFlight.addPassenger(james)),
        () -> assertEquals(1,
            businessFlight.getPassengersList().size()),
        () -> assertEquals(false,
            businessFlight.removePassenger(james)),
        () -> assertEquals(1,
            businessFlight.getPassengersList().size())
    );
}
}
```

In the listing above, we are doing the following:

- In the already existing nested classes `EconomyFlightTest` and `BusinessFlightTest` we are grouping the flight and passenger fields, as we would like to add one more testing level and reuse these fields for all tests concerning a particular flight type (#A). We are initializing these fields before the execution of each test (#B).
 - We are introducing a new nesting level, to test different passenger types. We are using the JUnit 5 `@DisplayName` annotation, to label the classes in a more expressive and easier to follow way (#C). All these labels start with the keyword "When".
 - We are labeling all existing tests with the help of the JUnit 5 `@DisplayName` annotation (#D). All these labels start with the keyword "Then".
 - We are refactoring the checking of the conditions by using the `assertAll` JUnit 5 method and grouping all previously existing conditions, which can now be read in a flow (#E).

This is how John has refactored the already existing tests, to facilitate him to continue the work in TDD style and to introduce the newly required premium flight business logic. If we run them at this time, we'll see that we are able to easily follow the way they work and how they are checking the business logic (fig. 20.10). Any new developer joining the project will find these tests extremely valuable as part of the documentation!

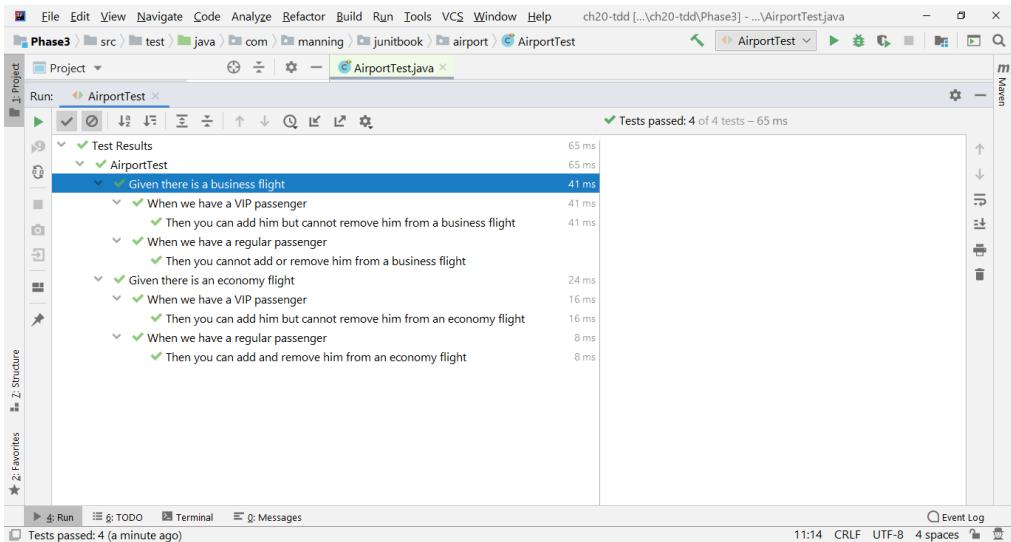


Figure 20.10 Running the refactored AirportTest for the economy flight and the business flight will allow the user to follow how the tests are working

John moves now to the implementation of the PremiumFlight class and its logic. He is creating PremiumFlight as a subclass of Flight, overrides the addPassenger and removePassenger methods, but these ones only act like stubs – they do not do anything and simply return false (listing 20.12). Their behavior will be extended later. Working TDD style involves first creating the tests, then the business logic.

Listing 20.12 The first design of the PremiumFlight class

```
public class PremiumFlight extends Flight { #A

    public PremiumFlight(String id) { #B
        super(id);
    } #B

    @Override
    public boolean addPassenger(Passenger passenger) { #C
        return false;
    } #C

    @Override
    public boolean removePassenger(Passenger passenger) { #D
        return false;
    } #D
}
```

In the listing above, we are doing the following:

- We are declaring the class `PremiumFlight` that extends `Flight` (#A) and we are creating a constructor for it (#B).
- We are creating the `addPassenger` (#C) and `removePassenger` (#D) methods as stubs, without any business logic and simply returning `false`.

John will now implement the tests according to the premium flight business logic from fig. 20.8 and fig. 20.9. This is shown in listing 20.13.

Listing 20.13 The tests for the behavior of the PremiumFlight

```
public class AirportTest {
    [...]

    @DisplayName("Given there is a premium flight") #A
    @Nested #A
    class PremiumFlightTest { #A
        private Flight premiumFlight; #B
        private Passenger mike; #B
        private Passenger james; #B

        @BeforeEach
        void setUp() { #C
            premiumFlight = new PremiumFlight("3"); #C
            mike = new Passenger("Mike", false); #C
            james = new Passenger("James", true); #C
        }

        @Nested #D
        @DisplayName("When we have a regular passenger") #D
        class RegularPassenger { #D

            @Test #E
            @DisplayName("Then you cannot add or remove him #E
                        from a premium flight") #E
            public void testPremiumFlightRegularPassenger() { #E
                assertAll("Verify all conditions for a regular passenger #F
                            and a premium flight", #F
                    () -> assertEquals(false, #G
                                         premiumFlight.addPassenger(mike)), #G
                    () -> assertEquals(0, #G
                                         premiumFlight.getPassengersList().size()), #G
                    () -> assertEquals(false, #H
                                         premiumFlight.removePassenger(mike)), #H
                    () -> assertEquals(0, #H
                                         premiumFlight.getPassengersList().size()) #H
                );
            }
        }

        @Nested #I
        @DisplayName("When we have a VIP passenger") #I
        class VipPassenger { #I
            @Test #J
            @DisplayName("Then you can add and remove him #J
                        from a premium flight") #J
            public void testPremiumFlightVipPassenger() { #J
                assertAll("Verify all conditions for a VIP passenger #K
                            and a premium flight", #K
                    () -> assertEquals(false, #L
                                         premiumFlight.addPassenger(james)), #L
                    () -> assertEquals(1, #L
                                         premiumFlight.getPassengersList().size()), #L
                    () -> assertEquals(false, #M
                                         premiumFlight.removePassenger(james)), #M
                    () -> assertEquals(0, #M
                                         premiumFlight.getPassengersList().size()) #M
                );
            }
        }
    }
}
```

```
        and a premium flight",
    () -> assertEquals(true,
        premiumFlight.addPassenger(james)),
    () -> assertEquals(1,
        premiumFlight.getPassengersList().size()),
    () -> assertEquals(true,
        premiumFlight.removePassenger(james)),
    () -> assertEquals(0,
        premiumFlight.getPassengersList().size())
);
}
}
});
```

In the listing above, we are doing the following:

- We are declaring the nested class PremiumFlightTest (#A) that contains the fields representing the flight and the passengers (#B) that we are setting up before each test (#C).
 - We are creating two classes nested at the second level inside PremiumFlightTest: RegularPassenger (#D) and VipPassenger (#I). We are using the JUnit 5 @DisplayName annotation, to label these classes in a more expressive and easier to follow way. All these labels start with the keyword "When".
 - We are inserting one test in each of the newly added RegularPassenger (#E) and VipPassenger (#J) classes. We are labeling these tests with the help of the JUnit 5 @DisplayName annotation. All these labels start with the keyword "Then".
 - Testing a premium flight and a regular passenger, we use the assertAll method to verify multiple conditions (#F). We check that we cannot add him to a premium flight and that trying to add him will not change the size of the passengers list (#G). Then, we check that we cannot remove him from a premium flight and that trying to remove him will not change the size of the passengers list (#H).
 - Testing a premium flight and a VIP passenger, we use the assertAll method to verify multiple conditions (#K). We check that we can add him to a premium flight and that adding him will increase the size of the passengers list (#L). Then, we check that we can remove him from a premium flight and that removing him will decrease the size of the passengers list (#M).

After having written the tests, John will now run them. Remember, we are working TDD style, so tests are coming first. The result of running now the tests is shown in fig. 20.11.

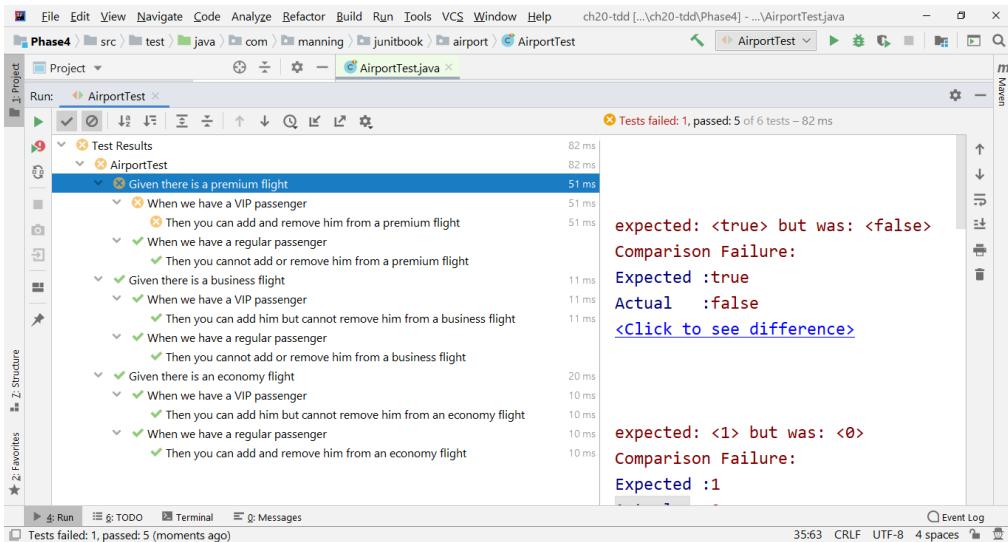


Figure 20.11 Running the newly added tests to check premium flights, before the effective code implementation, will result in some test failures. The programmer will understand which behavior he has to introduce in order to fix the failing tests

The fact that one of the tests is failing is no problem at this time. On the contrary, it is what John is expecting. Remember, we are working TDD, we are driven by tests, so we are first creating the test to fail then we are writing that piece of code that will make the test pass. But there is one more remarkable thing here: the test for a premium flight and a regular passenger is already green. This means that the existing business logic (the `addPassenger` and `removePassenger` methods returning `false`) is just enough for this case. He understands that he has only to focus on the VIP passenger. To quote Kent Beck again: "TDD helps you to pay attention to the right issues at the right time so you can make your designs cleaner, you can refine your designs as you learn. TDD enables you to gain confidence in the code over time."

So, John will move back to the `PremiumFlight` class and will add the business logic only for the VIP passengers. Driven by tests, he will go straight to the point (listing 20.14).

Listing 20.14 The PremiumFlight class with the full business logic

```
public class PremiumFlight extends Flight {

    public PremiumFlight(String id) {
        super(id);
    }

    @Override
    public boolean addPassenger(Passenger passenger) {
        if (passenger.isVip()) { #A
            return passengers.add(passenger); #A
        }
    }
}
```

```

        }
    return false;
}

@Override
public boolean removePassenger(Passenger passenger) {
    if (passenger.isVip()) {
        return passengers.remove(passenger);
    }
    return false;
}
}

```

In the listing above, we have added the following to the previously existing PremiumFlight class:

- We are adding a passenger only if he is a VIP (#A).
- We are removing a passenger only if he is a VIP (#B).

The result of running the tests now is shown in fig. 20.12. Everything went smoothly and was driven by the tests that guide the developer in writing the code that will make them pass. Additionally, the code coverage is 100%.

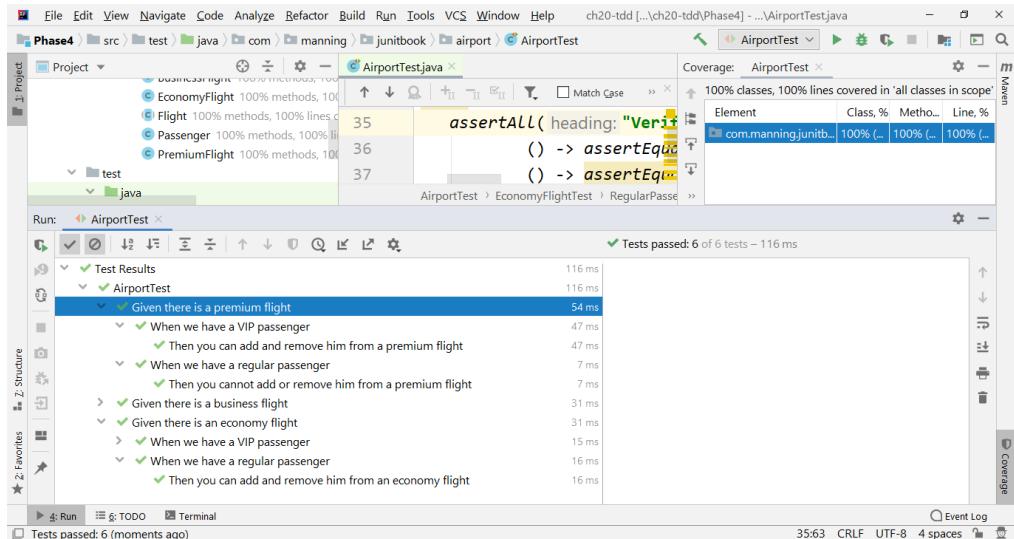


Figure 20.12 Running the full tests suite (economy, business and premium flights) after adding the business logic for the PremiumFlight:- the code coverage is 100%

20.5.2 Adding a passenger only once

There were a few situations when, on purpose or by mistake, the same passenger has been added to a flight more than once. This has generated a few problems with the management of the seats, and these situations must be avoided. We have to make sure that whenever one tries to add a passenger, if he has been previously added to the flight, the request should be rejected. This is new business logic and John will implement it TDD style.

John starts the implementation of this new feature by adding the test to check it. He will try repeatedly to add the same passenger to a flight (listing 20.15). We'll detail only the case of a regular passenger repeatedly added to an economy flight, all other cases being similar.

Listing 20.15 Trying to add the same passenger repeatedly to the same flight

```
public class AirportTest {  
    @DisplayName("Given there is an economy flight")  
    @Nested  
    class EconomyFlightTest {  
        private Flight economyFlight;  
        private Passenger mike;  
        private Passenger james;  
  
        @BeforeEach  
        void setUp() {  
            economyFlight = new EconomyFlight("1");  
            mike = new Passenger("Mike", false);  
            james = new Passenger("James", true);  
        }  
  
        @Nested  
        @DisplayName("When we have a regular passenger")  
        class RegularPassenger {  
            [...]  
            @DisplayName("Then you cannot add him to an economy flight  
                more than once")  
            @RepeatedTest(5)  
            public void testEconomyFlightRegularPassengerAddedOnlyOnce  
                (RepetitionInfo repetitionInfo) {  
                    for (int i=0; i<repetitionInfo.getCurrentRepetition(); i++) {  
                        economyFlight.addPassenger(mike);  
                    }  
                    assertAll("Verify a regular passenger can be added  
                            to an economy flight only once",  
                            () -> assertEquals(1,  
                                economyFlight.getPassengersList().size()),  
                            () -> assertTrue(  
                                economyFlight.getPassengersList().contains(mike)),  
                            () -> assertTrue(  
                                economyFlight.getPassengersList()  
                                    .get(0).getName().equals("Mike")));  
                }  
            }  
        }  
    }  
}
```

In the listing above, we are doing the following:

- We have marked the test as @RepeatedTest, 5 times, and we'll use the RepetitionInfo parameter inside it (#B).
- Each time a test is executed, we try to add the passenger the number of times that is specified by the RepetitionInfo parameter (#C).
- We'll do verifications using the assertAll method (#D): we check that the list of passengers has the size of 1 (#E), that the list contains the added passenger (#F), and on the first position (#G).

If we run the tests, they will fail. There is no business logic yet to prevent adding a passenger more than once (fig. 20.13).

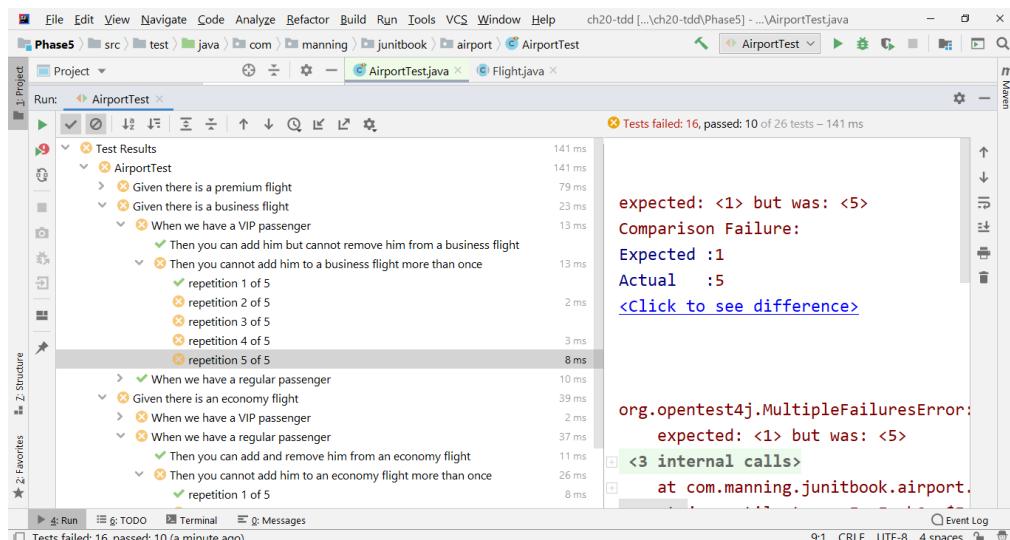


Figure 20.13 Running the tests that check that a passenger can be added only once to a flight, before implementing the business logic, will result in tests failure

To ensure the unicity of the passengers of a flight, John will change the passengers list structure to a set. So, he will do some refactoring into the code, which will also propagate inside the tests. The Flight class will change as shown in listing 20.16.

Listing 20.16 The Flight class after changing the list of passengers to a set

```
public abstract class Flight {  
    [...]  
    Set<Passenger> passengers = new HashSet<>(); #A  
    [...]  
    public Set<Passenger> getPassengersSet() { #B
```

```

        return Collections.unmodifiableSet(passengers); #C
    }

    [...]
}

```

In the listing above, we are doing the following:

- We have changed the type and the initialization of the `passengers` attribute to a set (#A), changed the name of the method (#B) and we are now returning an unmodifiable set (#C).

Listing 20.17 The new test checking that a passenger can be added only once to a flight

```

@DisplayName("Then you cannot add him to an economy flight
             more than once")
@RepeatedTest(5)
public void testEconomyFlightRegularPassengerAddedOnlyOnce
    (RepetitionInfo repetitionInfo) {
    for (int i=0; i<repetitionInfo.getCurrentRepetition(); i++)
        economyFlight.addPassenger(mike);
    }
    assertAll("Verify a regular passenger can be added
              to an economy flight only once",
        () -> assertEquals(1, #A
            economyFlight.getPassengersSet().size()), #A
        () -> assertTrue( #B
            economyFlight.getPassengersSet().contains(mike)), #B
        () -> assertTrue( #C
            new ArrayList<>()(economyFlight.getPassengersSet()) #C
            .get(0).getName().equals("Mike")));
}

```

In the listing above, we have changed the following:

- We are checking the size of the passengers set (#A), the fact that this set contains the newly added passenger (#B) and we are verifying the passenger on the first position (#C), after constructing a list from the existing set (we need to do this because a set has no order of the elements).

Running the tests will be now successful, with code coverage of 100% (fig. 20.14). John has implemented this new feature in TDD style.

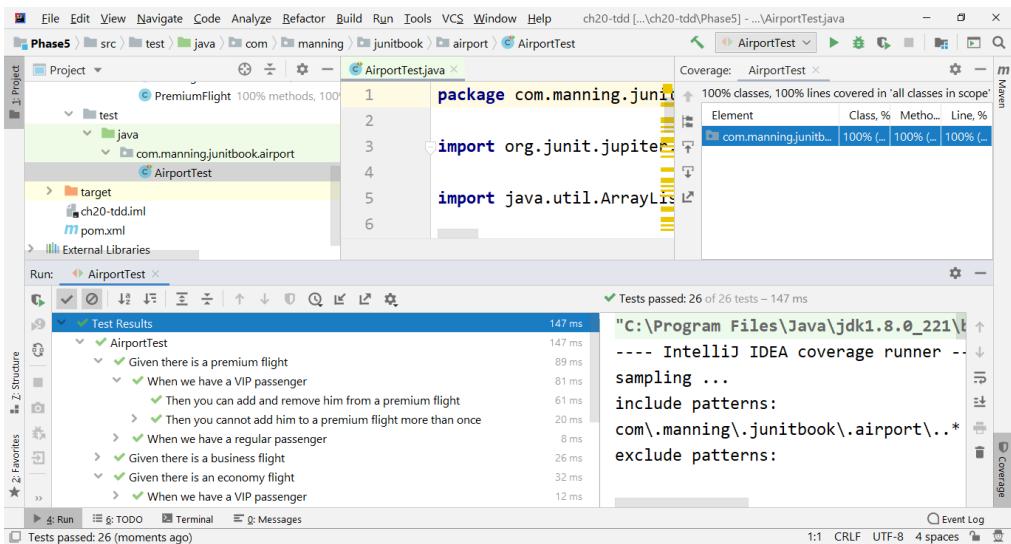


Figure 20.14 Successfully running the whole tests suite after implementing the business logic checking that a passenger can be added only once to a flight

The next chapter will be dedicated to another software development process largely used nowadays: Behavior Driven Development (BDD).

20.6 Summary

This chapter has covered the following:

- Examining the concept of Test Driven Development (TDD) and demonstrating how it helps developing safe applications, as tests will prevent introducing bugs into the well working existing code and will act as part of the documentation.
- Preparing a non-TDD flights management application to be moved to TDD by adding hierarchical JUnit 5 tests that cover the existing business logic.
- Refactoring and improving the code quality of this TDD application by replacing conditional with polymorphism while relying on the tests that we have developed.
- Implementing the premium flight feature to the flights management application by working TDD, starting by writing the tests and then implementing the business logic.
- Implementing the feature of adding a passenger only once to a flight by working TDD, starting by writing the tests and then changing the list of passengers to a set.

21

Behavior Driven Development with JUnit 5

This chapter covers:

- Introducing Behavior Driven Development
- Analyzing the benefits and challenges of Behavior Driven Development
- Moving a TDD application to BDD
- Developing a BDD application with the help of Cucumber and JUnit 5
- Developing a BDD application with the help of JBehave and JUnit 5
- Comparing Cucumber and JBehave

Some people refer to BDD as "TDD done right." You can also think of BDD as "how we build the right thing" and TDD as "how we build the thing right."

- **Millard Ellingsworth**

Kent Beck invented Test Driven Development in the early years of Agile. It is an effective technique that uses unit tests to verify the code. Working TDD style, the programmer will have first to write a test that checks the not yet implemented feature. The test is expected to fail. Then, the programmer will write the smallest piece of code that fixes the test. Eventually, the developer will refactor the code to be easier to understand and to maintain.

Despite its clear benefits, this usual loop

[test, code, refactor, (repeat)]

can bring developers to lose the overall picture of the business goals of the application. The project will become larger and more complex, the numbers of unit tests will increase and

will become harder to understand and to maintain. The tests may also be strongly coupled with the implementation. They focus on the unit (the class or the method) that is tested, while the business goals may not be considered.

Starting from TDD, a new technique has been created: the Behavior Driven Development. It focuses on the features themselves, it makes sure that these ones work as expected.

21.1 Introducing Behavior Driven Development

Behavior Driven Development

Behavior Driven Development is a software development technique that directly addresses the business requirements. It starts from the business requirements and goals and then transforms them into working features.

BDD encourages teams to interact and use concrete examples to communicate how the application must behave. It emerged from Test Driven Development.

Test Driven Development helps us to build safe software. Behavior Driven Development helps us building software providing business value.

Dan North originated the Behavior Driven Development in the mid-2000s. It is a software development technique that encourages teams to deliver software that matters, supporting the cooperation between stakeholders.

BDD helps us write software that really matters. We can find out the features that the organization really needs and we then focus on implementing them. We will be able to discover what the user actually needs and not only what he asks about.

BDD is a large topic and this chapter focuses on demonstrating how to use it in conjunction with JUnit 5 and how to effectively build features using this technique. For a comprehensive work on this subject, you may refer to another Manning book, "BDD in Action" by John Ferguson Smart. The second edition of this book is under development at the time of writing this chapter (<https://www.manning.com/books/bdd-in-action-second-edition>).

The communication between the people who are involved in the same project may bring up problems and misunderstandings. Usually, the flow works this way:

- The customer communicates to the business analyst his understanding about the functionality of a feature.
- The business analyst builds the requirements for the developers, describing the way the software must work.
- The developer creates the code based on the requirements and writes unit tests to implement the new feature.
- The tester creates the test cases based on the requirements and uses them to verify the way the new feature works.

It is possible that the information gets misunderstood, modified or ignored. The new feature may not do exactly what it has been initially expected.

21.1.1 Introducing a new feature

The business analyst discusses with the customer to decide the software features that will be able to address the business goals. These features are general requirements, like: "Allow the traveler to choose the shortest way to the destination", "Allow the traveler to choose the cheapest way to the destination".

These features need to be broken into stories. The stories might look like: "Find the route between source and destination with the smallest number of flight changes", "Find the quickest route between source and destination".

Stories will be defined through concrete examples. These examples will become the acceptance criteria for a story.

Acceptance criteria may be expressed BDD style through the keywords "Given", "When", "Then".

As an example, we may present the following acceptance criteria:

Given the flights operated by company X

When I want to find the quickest route from Bucharest to New York on May 15 20...

Then I will be provided the route Bucharest - Frankfurt - New York, with a duration of...

21.1.2 From requirements analysis to acceptance criteria

For the company using the flights management application, one business goal that we can formulate is "Increase sales by providing higher quality overall flight services". This is a very general goal, and it can be detailed through requirements:

- Provide an interactive application to choose flights
- Provide an interactive application to change flights
- Provide an interactive application to calculate the shortest route between source and destination

To make the customer happy, the features generated by the requirements analysis need to achieve the customer business goals or deliver business value. The initial ideas need to be described in more detail. One way to describe the previous requirements would be:

As a passenger

I want to know the flights for a given destination within a given period of time

So that I can choose the flight(s) that suit(s) my needs

or

As a passenger

I want to be able to change my initial flight(s) to a different one(s)

So that I can follow the changes in my schedule

A feature like "I can choose the flights that suit my needs" might be too large to be implemented at once - so, it must be divided. You may also want to get some feedback while passing through the milestones of the implementation of a feature.

The previous feature may be broken into smaller stories, such as the following:

- Find the direct flights that suit my needs (if any).
- Find the alternatives of flights with stopovers that suit my needs.
- Find the one-way flights that suit my needs.
- Find the there and back flights that suit my needs.

Generally, particular examples are used as acceptance criteria. Acceptance criteria will express what will make the stakeholder agree that the application is working the way it was expected.

In BDD, the definition of acceptance criteria is made using the Given/When/Then keywords. More exactly:

Given <a context>

When <an action occurs>

Then <expect a result>

As a concrete example:

Given the flights operated by the company

When I want to travel from Bucharest to London next Wednesday

Then I should be provided 2 possible flights: 10:35 and 16:20

21.1.3 BDD benefits and challenges

We'll point out a few benefits of the BDD approach:

- **Address user needs.** The users care less about the implementation and they are mainly interested in the application functionality. Working BDD style you get closer to addressing these needs.
- **Clarity.** Scenarios will clarify what software should do. Scenarios are described in simple language, easy to understand by technical and non-technical people. Ambiguities can be clarified by analyzing the scenario or by adding another scenario.
- **Supports change.** The scenarios will represent a part of the documentation of the software - in fact, living documentation, as it evolves simultaneously with the application. It also helps locate an incoming change. The automated acceptance tests will hinder the introduction of regressions when new changes are introduced.
- **Supports automation.** Scenarios can be transformed into automated tests, as the steps of the scenario are already defined.
- **Focuses on adding business value.** BDD will prevent introducing features that are not useful to the project. You will also be able to prioritize the functionalities.
- **Reduces costs.** Prioritizing the importance of the functionalities and avoiding the unnecessary ones will hinder the waste of resources and concentrate them to do

exactly what is needed.

As challenges, BDD requires engagement and strong collaboration. It requires interaction, direct communication, and constant feedback. This may be a challenge for some persons and, in the context of the present-day globalization and distributed teams, may require language skills and fitting time zones.

21.2 Working BDD with Cucumber and JUnit 5

Tested Data Systems Inc. is an outsourcing company creating software projects for different companies. The flights management application we have worked with is one of the projects under development.

Working TDD style, at the end of chapter 20, John, a programmer at Tested Data Systems, has left the development of the flights management application in a stage where it was able to work with three types of flights: economy, business, and premium. Besides this, he implemented the requirement that a passenger can be added only once to a flight. The functionality of the application can be quickly reviewed if we run the tests (fig. 21.1).

John has already introduced, in a discrete way, a first taste of the Behavior Driven Development way of working. We can easily read how the application works by following the tests using the "Given", "When", "Then" keywords. John will take it over from here, moving it to BDD with Cucumber and also introducing new features.

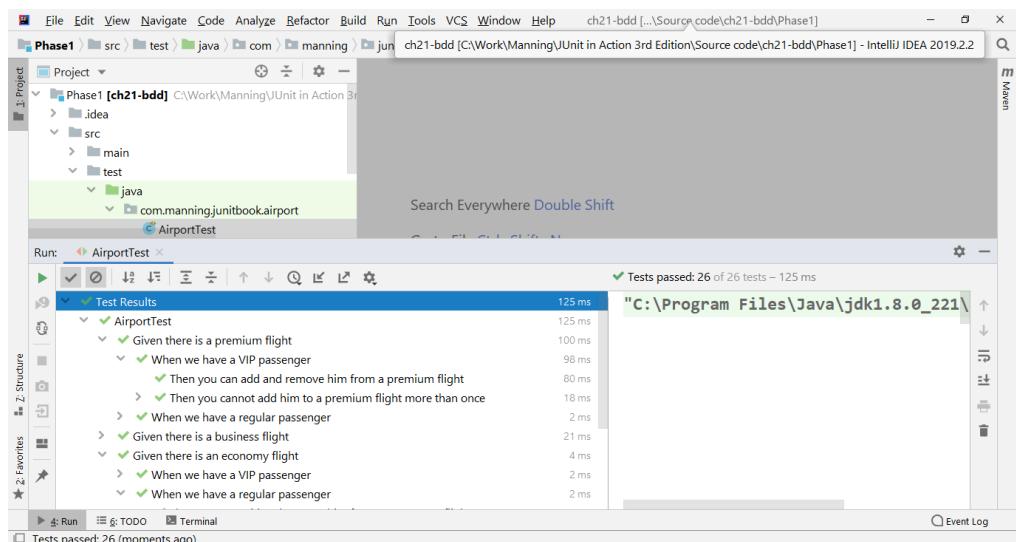


Figure 21.1 Successfully running the tests of the TDD flights management application – the user may follow the execution of the annotated methods and read the scenarios

21.2.1

Introducing Cucumber

Cucumber

Cucumber is a Behavior Driven Development testing tool framework. It describes the application scenarios in plain English text, using a language called Gherkin. Cucumber is easy to read and understand by stakeholders and allows automation.

The main capabilities of Cucumber are:

- Scenarios or examples describe the requirements.
- A scenario is defined through a list of steps to be executed by Cucumber.
- Cucumber executes the code corresponding to the scenarios, checks that the software follows these requirements and generates a report about the success or failure of each scenario.

The main capabilities of Gherkin are:

- Gherkin defines simple grammar rules that allow Cucumber to understand English plain text.
- Gherkin documents the behavior of the system. The requirements are always up to date, as they are provided through these scenarios which represent living specifications.

Cucumber ensures that technical and non-technical persons can easily read, write and understand the acceptance tests. The acceptance tests became an instrument of communication between the stakeholders of the project.

A Cucumber acceptance test can look this way:

Given there is an economy flight

When we have a regular passenger

Then you can add and remove him from an economy flight

We notice again the Given/When/Then words. We have already explained that they are the keywords for describing a scenario and we have already introduced them in our previous work with JUnit 5. But we have to know from the very beginning that we'll no longer use them just for labeling. Cucumber will take care to interpret the sentences starting with these keywords and generate methods which it will annotate using exactly these annotations: @Given, @When and @Then.

The acceptance tests will be written in Cucumber feature files. A feature file is an entry point to the Cucumber tests. It is a file where we will describe our tests in Gherkin. A feature file can contain one or many scenarios.

John makes plans for starting the work with Cucumber in the project. He will first introduce the Cucumber dependencies into the existing Maven configuration. He will create a first

Cucumber feature and will generate the skeleton of the Cucumber tests. Then, he will move the previously existing JUnit 5 tests to fill in this Cucumber generated tests skeleton.

Listing 21.1 The Cucumber dependencies added to the pom.xml file

```
<dependency>
    <groupId>info.cukes</groupId>
    <artifactId>cucumber-java</artifactId>
    <version>1.2.5</version>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>info.cukes</groupId>
    <artifactId>cucumber-junit</artifactId>
    <version>1.2.5</version>
    <scope>test</scope>
</dependency>
```

In listing 21.1, John has introduced the two needed Maven dependencies: `cucumber-java` and `cucumber-junit`.

21.2.2 Moving a TDD feature to Cucumber

John will start creating the Cucumber features. He will follow the Maven standard folders structure and introduce the features into the test/resources folder. He will create the test/resources/features folder and, inside it, he will create the `passengers_policy.feature` file (fig. 21.2).

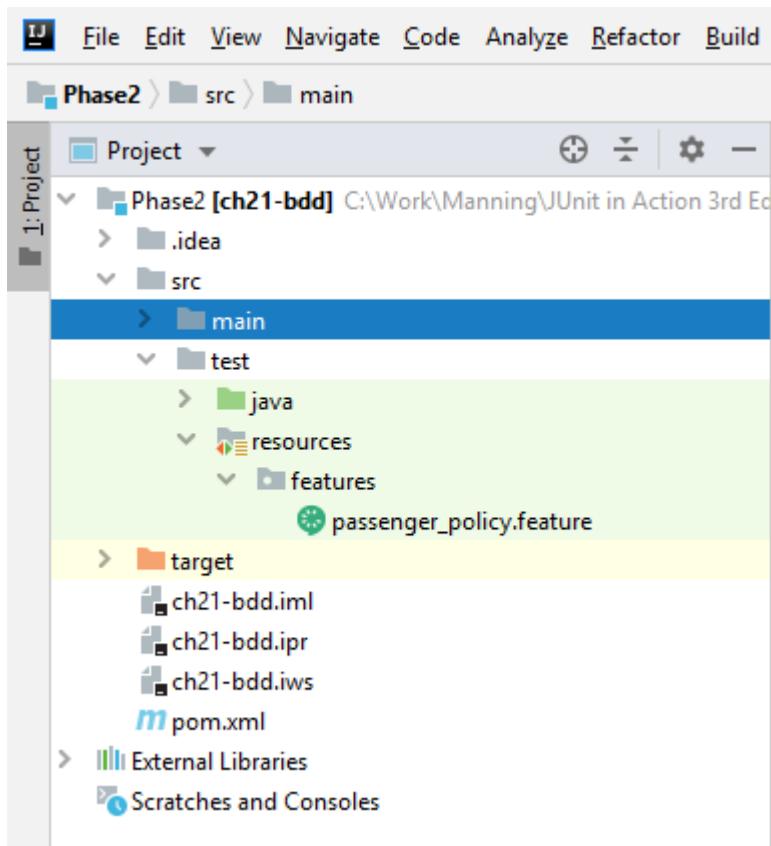


Figure 21.2 The new Cucumber `passenger_policy.feature` file is created into the `test/resources/features` folder, following the Maven rules

John will follow the Gherkin syntax and introduce the feature named "Passengers Policy", together with a short description of what it intends to do. Then, he will follow the same Gherkin syntax and write the scenarios (listing 21.2).

Listing 21.2 The `passenger_policy.feature` file

Feature: Passengers Policy

The company follows a policy of adding and removing passengers, depending on the passenger type and on the flight type

Scenario: Economy flight, regular passenger

Given there is an economy flight

When we have a regular passenger

Then you can add and remove him from an economy flight

And you cannot add a regular passenger to an economy flight more than once

```

Scenario: Economy flight, VIP passenger
  Given there is an economy flight
  When we have a VIP passenger
  Then you can add him but cannot remove him from an economy flight
  And you cannot add a VIP passenger to an economy flight more than once

Scenario: Business flight, regular passenger
  Given there is a business flight
  When we have a regular passenger
  Then you cannot add or remove him from a business flight

Scenario: Business flight, VIP passenger
  Given there is a business flight
  When we have a VIP passenger
  Then you can add him but cannot remove him from a business flight
  And you cannot add a VIP passenger to a business flight more than once

Scenario: Premium flight, regular passenger
  Given there is a premium flight
  When we have a regular passenger
  Then you cannot add or remove him from a premium flight

Scenario: Premium flight, VIP passenger
  Given there is a premium flight
  When we have a VIP passenger
  Then you can add and remove him from a premium flight
  And you cannot add a VIP passenger to a premium flight more than once

```

We see the keywords Feature, Scenario, Given, When, Then, And which are highlighted. If we right-click on this feature file, we see that we have the possibility to run it directly (fig. 21.3).

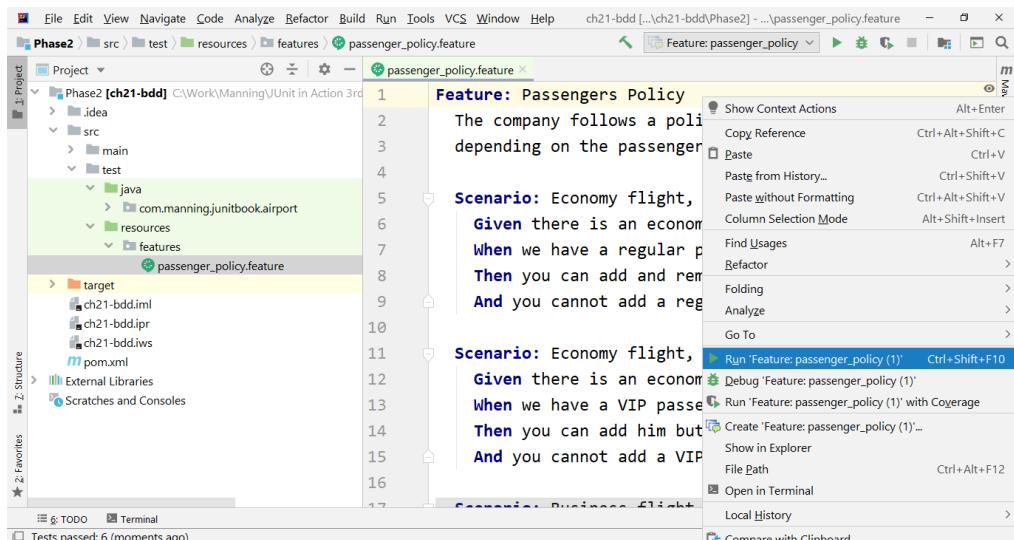


Figure 21.3 Directly running the passengers_policy.feature file by right-clicking on the feature file

This is possible only if two things are fulfilled. First, the appropriate plugins must be activated. In order to do this, we must go to File -> Settings->Plugins, and we install from here the Cucumber for Java and Gherkin plugins (fig. 21.4 and 21.5).

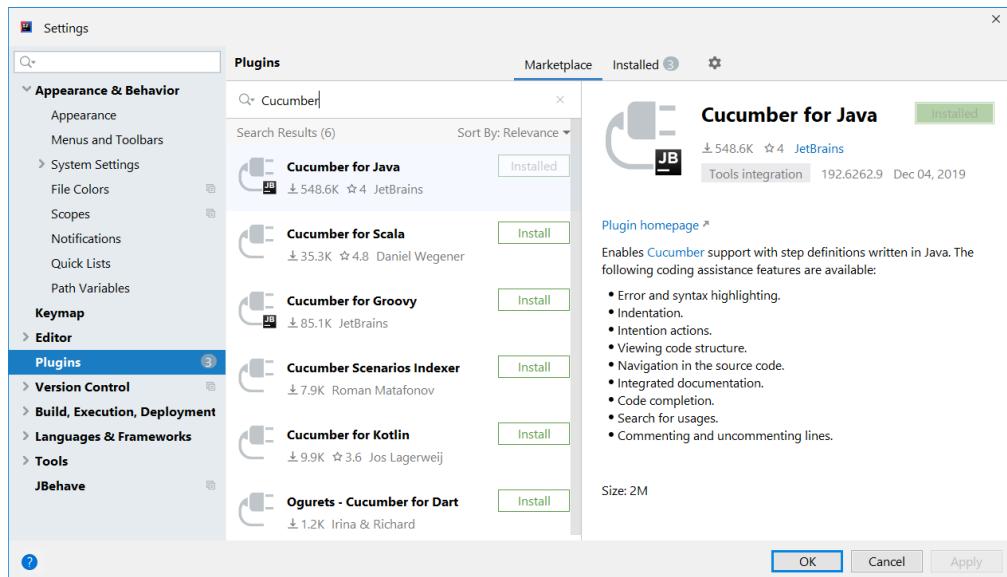


Figure 21.4 Installing the Cucumber for Java plugin from the File -> Settings -> Plugins menu

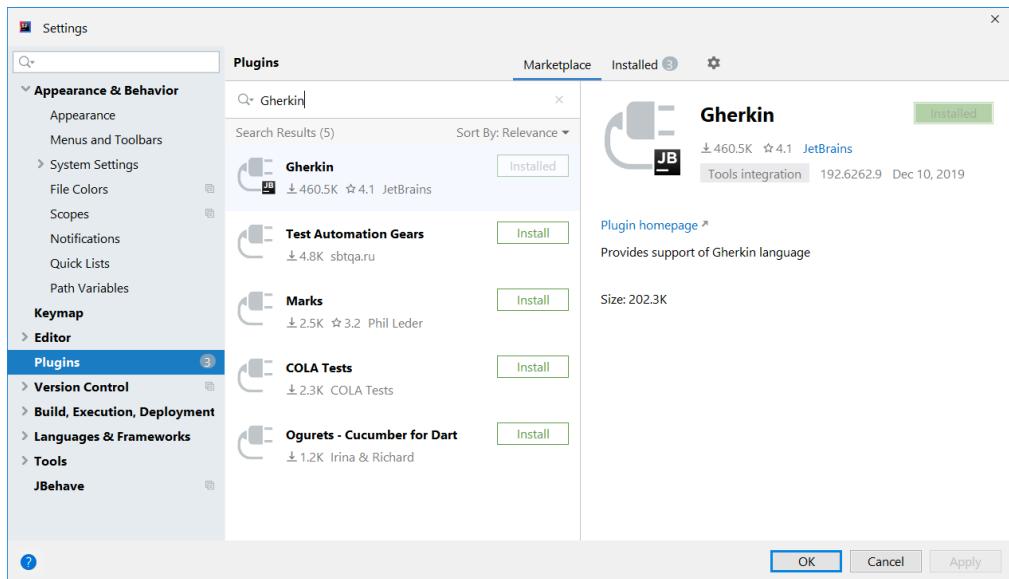


Figure 21.5 Installing the Gherkin plugin from the File -> Settings -> Plugins menu

Then, we must configure the way the feature is run. We need to go to Run -> Edit Configurations, and to set the following (fig. 21.6):

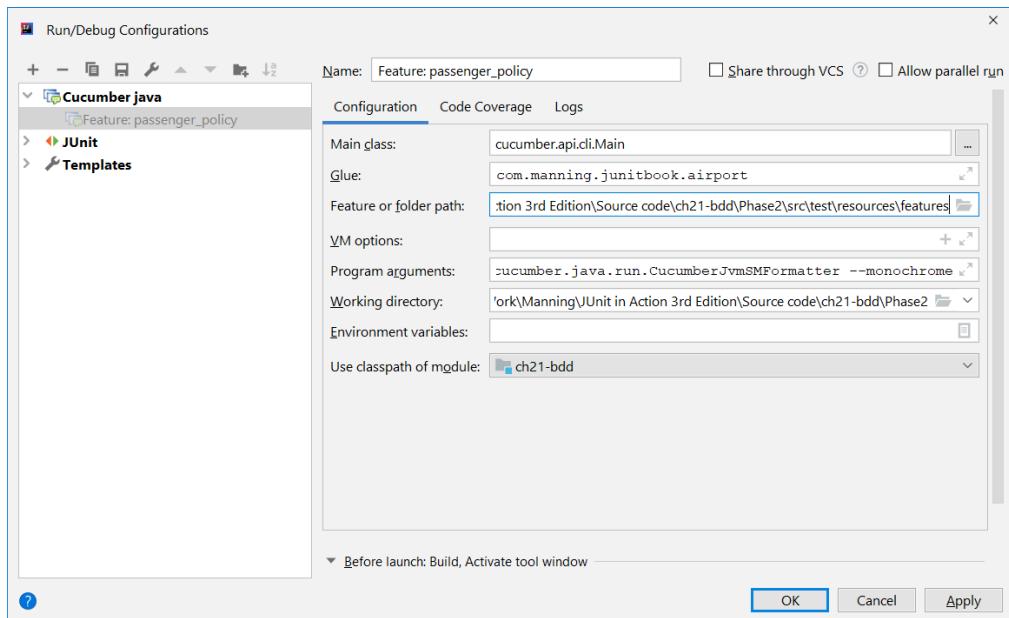


Figure 21.6 Setting the feature configuration by filling in the Main class, Glue, Feature or folder path, and Working directory

- Main class: cucumber.api.cli.Main
- Glue (the package where step definitions are stored): com.manning.junitbook.airport
- Feature or folder path: the test/resources/features folder we have created
- Working directory: the project folder

Running the feature directly will generate the skeleton of the Java Cucumber tests (fig. 21.7).

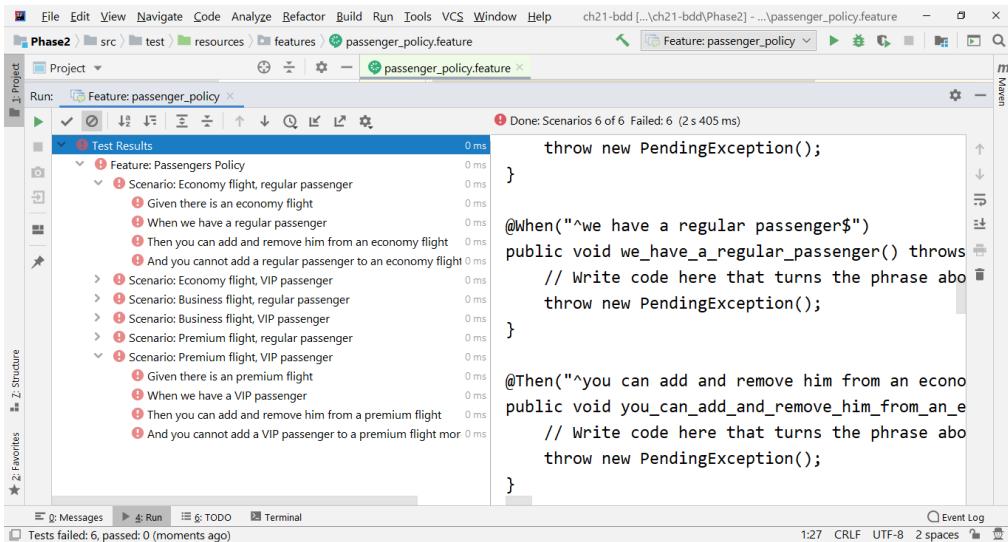


Figure 21.7 Getting the skeleton of the passengers policy feature by directly running the feature file – the annotated methods will be executed in order to verify the scenarios

John will now create a new Java class into the test/java folder, into the com.manning.junitbook.airport package. This class will be named PassengersPolicy and, for the beginning, it will contain the tests skeleton (listing 21.3). The execution of such a test will follow the scenarios described in the passengers_policy.feature file. For example, when executing the step:

Given there is an economy flight

the program will execute the method annotated with:

```
@Given("^there is an economy flight$")
```

Listing 21.3 The initial PassengersPolicy class

```
public class PassengerPolicy {
    @Given("^there is an economy flight$")
    public void there_is_an_economy_flight() throws Throwable {
        // Write code here that turns the phrase above into concrete actions
        throw new PendingException();
    }

    @When("^we have a regular passenger$")
    public void we_have_a_regular_passenger() throws Throwable {
        // Write code here that turns the phrase above into concrete actions
        throw new PendingException();
    }

    @Then("^you can add and remove him from an economy flight$")
    public void you_can_add_and_remove_him_from_an_economy_flight()
}
```

```

    throws Throwable {
        // Write code here that turns the phrase above into concrete actions #F
        throw new PendingException(); #F
    } #F
    [...] #F
}

```

In the listing above we are doing the following:

- The Cucumber plugin has generated a method annotated with `@Given("^there is an economy flight$")`, meaning that this method will be executed when the step "Given there is an economy flight" from the scenario will be executed (#A).
- A method stub has been generated by the Cucumber plugin to be implemented with the code addressing the step "Given there is an economy flight" from the scenario (#B).
- The Cucumber plugin has generated a method annotated with `@When("^we have a regular passenger$")`, meaning that this method will be executed when the step "When we have a regular passenger" from the scenario will be executed (#C).
- A method stub has been generated by the Cucumber plugin to be implemented with the code addressing the step "When we have a regular passenger" from the scenario (#D).
- The Cucumber plugin has generated a method annotated with `@Then("^you can add and remove him from an economy flight$")`, meaning that this method will be executed when the step "Then you can add and remove him from an economy flight" from the scenario will be executed (#E).
- A method stub has been generated by the Cucumber plugin to be implemented with the code addressing the step "Then you can add and remove him from an economy flight" from the scenario (#F).
- The rest of the methods are implemented in a similar way, we have covered the Given, When and Then steps of one scenario.

John follows the business logic of each step that has been defined and transposes it into the tests from listing 21.4 – the steps of the scenarios that need to be verified.

Listing 21.4 Implementing the business logic of the previously defined steps

```

public class PassengerPolicy {
    private Flight economyFlight; #A
    private Passenger mike; #A
    [...]
    @Given("^there is an economy flight$")
    public void there_is_an_economy_flight() throws Throwable { #B
        economyFlight = new EconomyFlight("1"); #B
    } #C
    @When("^we have a regular passenger$")
    public void we_have_a_regular_passenger() throws Throwable { #D
        #D
}

```

```

        mike = new Passenger("Mike", false); #E
    }

    @Then("^you can add and remove him from an economy flight$")
    public void you_can_add_and_remove_him_from_an_economy_flight()
        throws Throwable {
        assertAll("Verify all conditions for a regular passenger
                  and an economy flight",
                  () -> assertEquals("1", economyFlight.getId()), #G
                  () -> assertEquals(true, economyFlight.addPassenger(mike)), #G
                  () -> assertEquals(1,
                                      economyFlight.getPassengersSet().size()), #G
                  () ->
                      assertTrue(economyFlight.getPassengersSet().contains(mike)), #G
                  () -> assertEquals(true, economyFlight.removePassenger(mike)), #G
                  () -> assertEquals(0, economyFlight.getPassengersSet().size()) #G
        );
    }
    [...]
}

```

In the listing above we are doing the following:

- We declare the instance variables for the test, among which `economyFlight` and `mike` as a `Passenger` (#A).
- We write the method corresponding to the “Given there is an economy flight” business logic step (#B) by initializing the `economyFlight` (#C).
- We write the method corresponding to the “When we have a regular passenger” business logic step (#D) by initializing the regular passenger `mike` (#E).
- We write the method corresponding to the “Then you can add and remove him from an economy flight” business logic step (#F) by checking all the conditions by using the `assertAll` JUnit 5 method, which can now be read in a flow (#G).
- The rest of the methods are implemented in a similar way, we have covered the Given, When and Then steps of one scenario.

In order to run our Cucumber tests, we'll need a special class. The name of the class could be anything, we have chosen `CucumberTest` (listing 21.5).

Listing 21.5 The CucumberTest class

```

@RunWith(Cucumber.class) #A
@CucumberOptions(
    plugin = {"pretty"}, #B
    features = "classpath:features") #C
public class CucumberTest { #D

    /**
     * This class should be empty, step definitions should be in separate classes.
     */
}

```

In the listing above, we are doing the following:

- We have annotated this class with `@RunWith(Cucumber.class)` annotation (#A). Executing it as any JUnit test class will run all features found on the classpath in the same package. As there is no Cucumber JUnit 5 extension at the moment of writing this chapter, we use the JUnit 4 runner.
- The `@CucumberOptions` (#B) annotation provides the plugin option (#C), that is used to specify different formatting options for the output reports. Using "pretty", the Gherkin source will be printed with additional colors (fig. 21.8). Other plugin options include "html" and "json", but "pretty" is appropriate for us now. And the `features` option (#D) helps Cucumber to locate the feature file in the project folder structure. It will look for the features folder on the classpath - and remember that the `src/test/resources` folder is maintained by Maven on the classpath!

By running the tests, we see that the code coverage is 100% (fig. 21.8), so we have kept the existing test functionalities before moving to Cucumber.

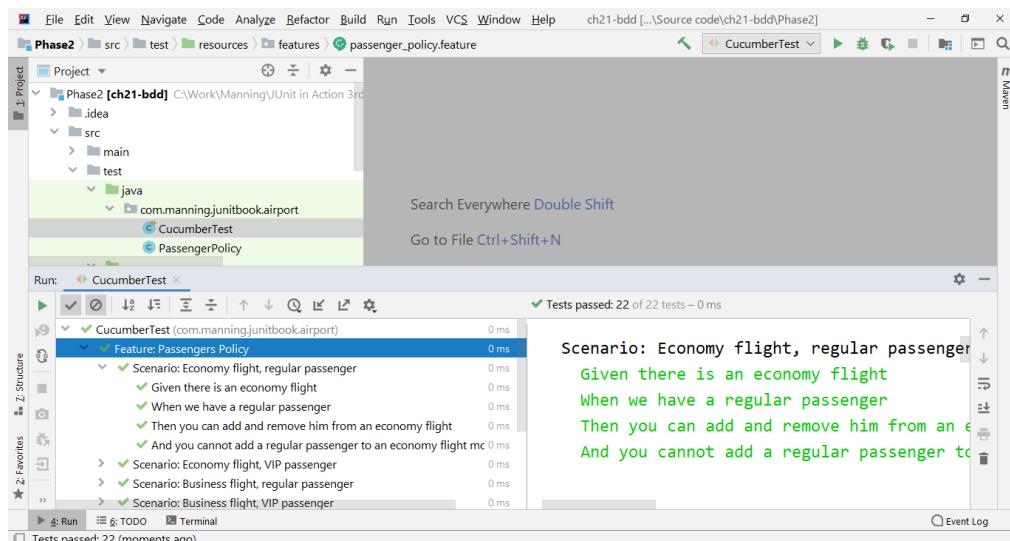


Figure 21.8 Running CucumberTest. The Gherkin source will be pretty-printed, the successful tests are displayed in green, the code coverage is 100%

There is another advantage to the move to BDD. We compare the length of the pre-Cucumber `AirportTest` class, which has 207 lines, with the one of the `PassengerPolicy` class, which has 157 lines. So, the testing code is now at only 75% of the pre-Cucumber size, yet it has the same 100% coverage. Where does this gain come from? Remember that the `AirportTest` file contained 7 classes, on 3 levels: `AirportTest` at the top level; one `EconomyFlightTest` and one `BusinessFlightTest` at the second level; and, at the third

level, two `RegularPassenger` and two `VipPassenger` classes. The code duplication is now really jumping to our attention, but that was the solution having only JUnit 5.

With Cucumber, each step is implemented only once, and if we have the same step in more than one scenario, we'll avoid the code duplication.

21.2.3 Adding a new feature with the help of Cucumber

John receives a new feature to implement, concerning the policy of bonus points that are awarded to the passenger.

The specifications about calculating the bonus points consider the mileage, meaning the distance that is traveled by each passenger. The bonus will be calculated for all flights of the passenger and it depends on a factor: the mileage will be divided by 10 for VIP passengers and by 20 for regular ones (fig 21.9).

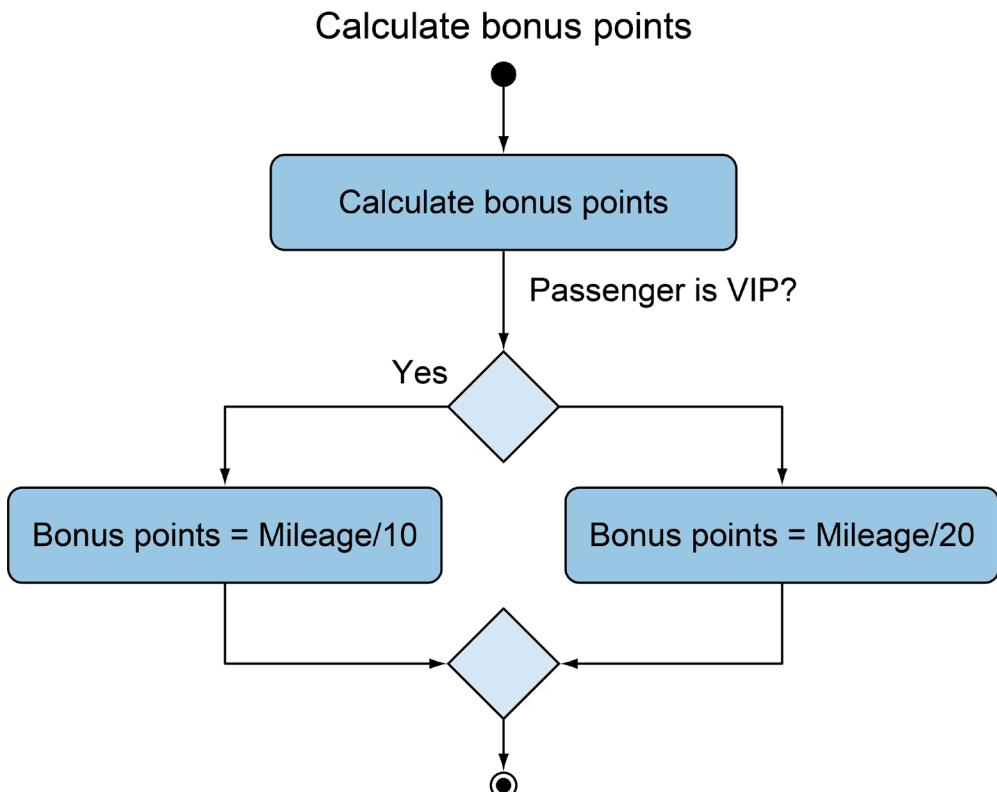


Figure 21.9 The business logic of awarding bonus points: the mileage will be divided by 10 for VIP passenger and by 20 for regular ones

John will move to the BDD scenarios, tests and implementation. He will define the scenarios of awarding the bonus points (listing 21.6) and generate the Cucumber tests that describe the scenarios. In the beginning, they are expected to fail. Then, he will effectively add the code that implements the bonus award, run the tests and expect them to be green.

Listing 21.6 The bonus_policy.feature file

Feature: Bonus Policy

The company follows a bonus policy, depending on the passenger type and on the mileage

Scenario Outline: Regular passenger bonus policy

#A

Given we have a regular passenger with a mileage

When the regular passenger travels <mileage1> and <mileage2> and <mileage3>

#B

Then the bonus points of the regular passenger should be <points>

#B

Examples:

mileage1	mileage2	mileage3	points
349	319	623	64
312	356	135	40
223	786	503	75
482	98	591	58
128	176	304	30

#C

#C

#C

#C

#C

#C

Scenario Outline: VIP passenger bonus policy

#A

Given we have a VIP passenger with a mileage

When the VIP passenger travels <mileage1> and <mileage2> and <mileage3>

#B

Then the bonus points of the VIP passenger should be <points>

#B

Examples:

mileage1	mileage2	mileage3	points
349	319	623	129
312	356	135	80
223	786	503	151
482	98	591	117
128	176	304	60

#C

#C

#C

#C

#C

In the listing above, we are using the following new things compared to the previous Cucumber feature that we have implemented:

- We introduce a new capability of Cucumber - Scenario Outline (#A). With Scenario Outline, values do not need to be hard-coded in the step definitions.
- Values are replaced with parameters as into the step-definition itself - you can see <mileage1>, <mileage2>, <mileage3> and <points> as parameters (#B).
- The effective values are defined in the Examples table, at the end of the Scenario Outline (#C). The first row in the first table defines the values of 3 mileages (349, 319, 623). Adding them and dividing them by 20 (the regular passenger factor), we get the integer part 64 (the number of bonus points). This successfully replaces the JUnit 5 parameterized tests, having the advantage that the values are kept inside the scenarios, and easy to be understood by everyone.

We will configure the way the feature is run. We need to go to Run -> Edit Configurations, and to set the following (fig. 21.10):

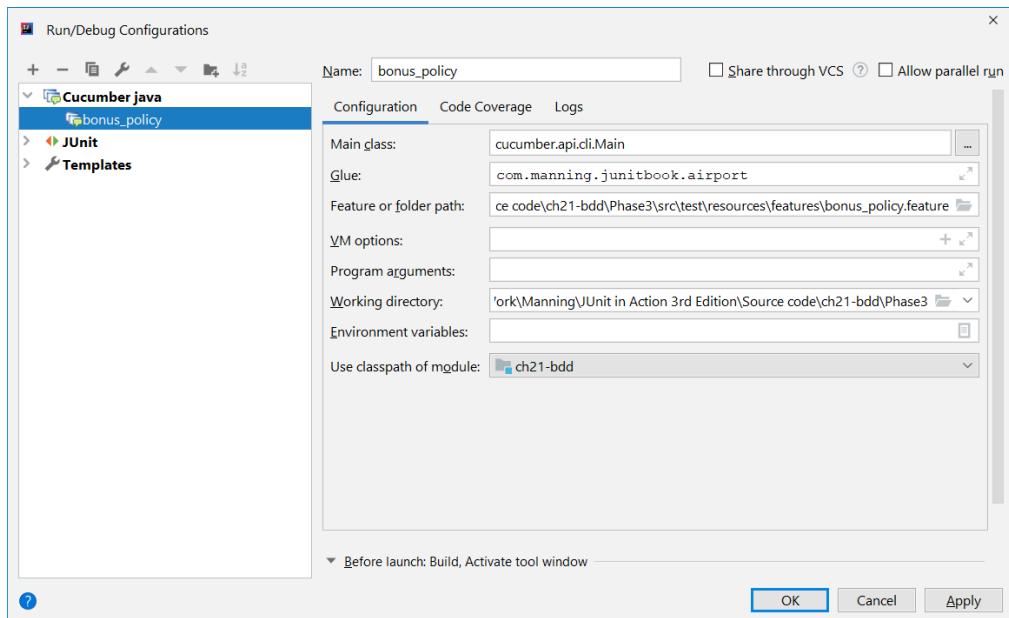


Figure 21.10 Setting the configuration for the new `bonus_policy.feature` by filling in the Main class, Glue, Feature or folder path, and Working directory

- Main class: `cucumber.api.cli.Main`
- Glue (the package where step definitions are stored): `com.manning.junitbook.airport`
- Feature or folder path: `test/resources/features/bonus_policy.feature` we have created
- Working directory: the project folder

Running the feature directly will generate the skeleton of the Java Cucumber tests (fig. 21.11).

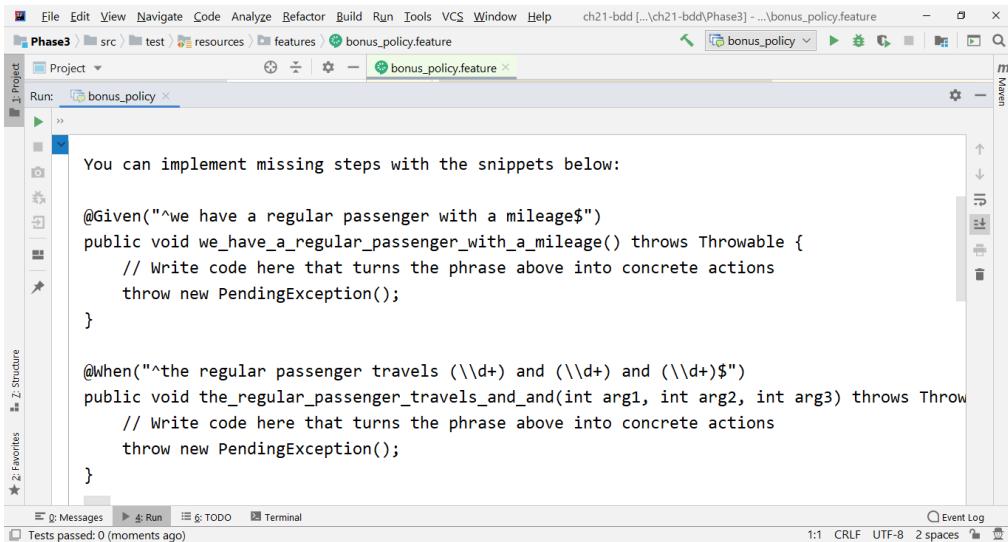


Figure 21.11 Getting the skeleton of the bonus policy feature by directly running the feature file

John will now create a new Java class into the test/java folder, into the com.manning.junitbook.airport package. This class will be named `BonusPolicy` and, for the beginning, it will contain the tests skeleton (listing 21.7). The execution of such a test will follow the scenarios described in the `bonus_policy.feature` file.

Listing 21.7 The initial BonusPolicy class

```
public class BonusPolicy {
    @Given("^we have a regular passenger with a mileage$")
    public void we_have_a_regular_passenger_with_a_mileage() #A
        throws Throwable { #B
    // Write code here that turns the phrase above into concrete actions #B
    throw new PendingException(); #B
}

@When("^the regular passenger travels (\d+) and (\d+) and (\d+)$")
public void the_regular_passenger_travels_and_and(
    int arg1, int arg2, int arg3) throws Throwable { #C
    // Write code here that turns the phrase above into concrete actions #D
    throw new PendingException(); #D
}

@Then("^the bonus points of the regular passenger should be (\d+)$") #E
public void the_bonus_points_of_the_regular_passenger_should_be(int arg1) #F
    throws Throwable { #F
    // Write code here that turns the phrase above into concrete actions #F
    throw new PendingException(); #F
}
[...]
```

```
}
```

In the listing above we are doing the following:

- The Cucumber plugin has generated a method annotated with `@Given("^we have a regular passenger with a mileage$")`, meaning that this method will be executed when the step "Given we have a regular passenger with a mileage" from the scenario will be executed (#A).
- A method stub has been generated by the Cucumber plugin to be implemented with the code addressing the step "Given we have a regular passenger with a mileage" from the scenario (#B).
- The Cucumber plugin has generated a method annotated with `@When("^the regular passenger travels (\\"d+) and (\\"d+) and (\\"d+)")`, meaning that this method will be executed when the step "When the regular passenger travels <mileage1> and <mileage2> and <mileage3>" from the scenario will be executed (#C).
- A method stub has been generated by the Cucumber plugin to be implemented with the code addressing the step "When the regular passenger travels <mileage1> and <mileage2> and <mileage3>" from the scenario (#D). This method has three parameters, corresponding to the three different mileages.
- The Cucumber plugin has generated a method annotated with `@Then("^the bonus points of the regular passenger should be (\\"d+)")`, meaning that this method will be executed when the step "Then the bonus points of the regular passenger should be <points>" from the scenario will be executed (#E).
- A method stub has been generated by the Cucumber plugin to be implemented with the code addressing the step "Then the bonus points of the regular passenger should be <points>" from the scenario (#F). This method has one parameter, corresponding to the points.
- The rest of the methods are implemented in a similar way, we have covered the Given, When and Then steps of one scenario.

Now, John will create the Mileage class, declaring the fields and the methods, but not implementing them yet. John needs to use the methods of this class for the tests, make these tests initially fail, then implement the methods and make the tests pass.

Listing 21.8 The Mileage class, with no implementation of the methods

```
public class Mileage {  
  
    public static final int VIP_FACTOR = 10; #A  
    public static final int REGULAR_FACTOR = 20; #A  
  
    private Map<Passenger, Integer> passengersMileageMap = new HashMap<>(); #B  
    private Map<Passenger, Integer> passengersPointsMap = new HashMap<>(); #B  
  
    public void addMileage(Passenger passenger, int miles) { #C
```

```

    }

    public void calculateGivenPoints() { #D
    }

}

```

In the listing above, we are doing the following:

- We have declared the `VIP_FACTOR` and `REGULAR_FACTOR` constants, corresponding to the factor by which we divide the mileage for each type of passenger in order to get the bonus points (#A).
- We have declared `passengersMileageMap` and `passengersPointsMap`, two maps having as key the passenger and keeping as value the mileage and the points for that passenger, respectively (#B).
- We have declared the `addMileage` method, which will populate the `passengersMileageMap` with the mileage for each passenger (#C). The method does not do anything, for now, it will be written later, to fix the tests.
- We have declared the `calculateGivenPoints` method, which will populate the `passengersPointsMap` with the bonus points for each passenger (#D). The method does not do anything, for now, it will be written later, to fix the tests.

John will now turn his attention to write the unimplemented tests from the `BonusPolicy` class, to follow the business logic of this feature (listing 21.9).

Listing 21.9 The business logic of the steps from BonusPolicy

```

public class BonusPolicy {
    private Passenger mike; #A
    private Mileage mileage; #A
    [...]

    @Given("^we have a regular passenger with a mileage$")
    public void we_have_a_regular_passenger_with_a_mileage() #B
        throws Throwable {
        mike = new Passenger("Mike", false); #C
        mileage = new Mileage(); #C
    }

    @When("^the regular passenger travels (\\d+) and (\\d+) and (\\d+)$") #D
    public void the_regular_passenger_travels_and_and(int mileage1, int
        mileage2, int mileage3) throws Throwable {
        mileage.addMileage(mike, mileage1); #E
        mileage.addMileage(mike, mileage2); #E
        mileage.addMileage(mike, mileage3); #E
    }

    @Then("^the bonus points of the regular passenger should be (\\d+)$") #F
    public void the_bonus_points_of_the_regular_passenger_should_be
        (int points) throws Throwable {
        mileage.calculateGivenPoints(); #G
    }
}

```

```

        assertEquals(points,
                     mileage.getPassengersPointsMap().get(mike).intValue());           #H
    }
}
}

```

In the listing above, we are doing the following:

- We declare the instance variables for the test, among which `mileage` and `mike` as a `Passenger` (#A).
- We write the method corresponding to the "Given we have a regular passenger with a mileage" business logic step (#B) by initializing the passenger and the `mileage` (#C).
- We write the method corresponding to the "When the regular passenger travels <mileage1> and <mileage2> and <mileage3>" business logic step (#D) by adding mileages to the regular passenger `mike` (#E).
- We write the method corresponding to the "Then the bonus points of the regular passenger should be <points>" business logic step (#F) by calculating the given points (#G) and checking that the calculated value is the expected one (#H).
- The rest of the methods are implemented in a similar way, we have covered the `Given`, `When` and `Then` steps of one scenario.

If we run the bonus points tests now they will fail (fig. 21.12), as the business logic is not yet implemented (the `addMileage` and `calculateGivenPoints` methods are empty, the business logic is implemented after the tests).

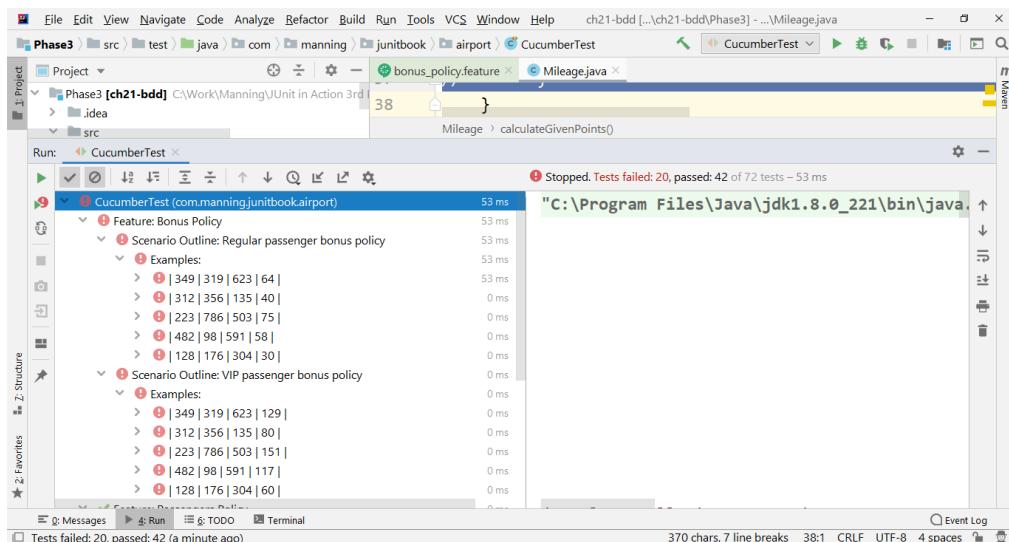


Figure 21.12 Running the bonus points tests before the business logic implementation will fail

John will move back to the implementation of the two remaining business logic methods from the Mileage class (`addMileage` and `calculateGivenPoints`), as shown in listing 21.10.

Listing 21.10 The implementation of business logic from the Mileage class

```
public void addMileage(Passenger passenger, int miles) {
    if (passengersMileageMap.containsKey(passenger)) { #A
        passengersMileageMap.put(passenger, #B
            passengersMileageMap.get(passenger) + miles); #B
    } else { #C
        passengersMileageMap.put(passenger, miles);
    }
}

public void calculateGivenPoints() {
    for (Passenger passenger : passengersMileageMap.keySet()) { #D
        if (passenger.isVip()) { #E
            passengersPointsMap.put(passenger, #E
                passengersMileageMap.get(passenger)/ VIP_FACTOR);
        } else { #F
            passengersPointsMap.put(passenger, #F
                passengersMileageMap.get(passenger)/ REGULAR_FACTOR);
        }
    }
}
```

In the listing above, we are doing the following:

- In the `addMileage` method, we check if the `passengersMileageMap` already contains a passenger (#A). If that passenger already exists, we add the mileage to him (#B), otherwise, we create a new entry into the map having that passenger as a key and the miles as the initial value (#C).
- In the `calculateGivenPoints` method, we browse the passengers set (#D) and, for each passenger, if he is a VIP, we calculate the bonus points by dividing the mileage to the VIP factor (#E). Otherwise, we calculate the bonus points by dividing the mileage to the regular factor (#F).

Running the bonus points tests now will be successful and nicely displayed, as shown in fig. 21.13.

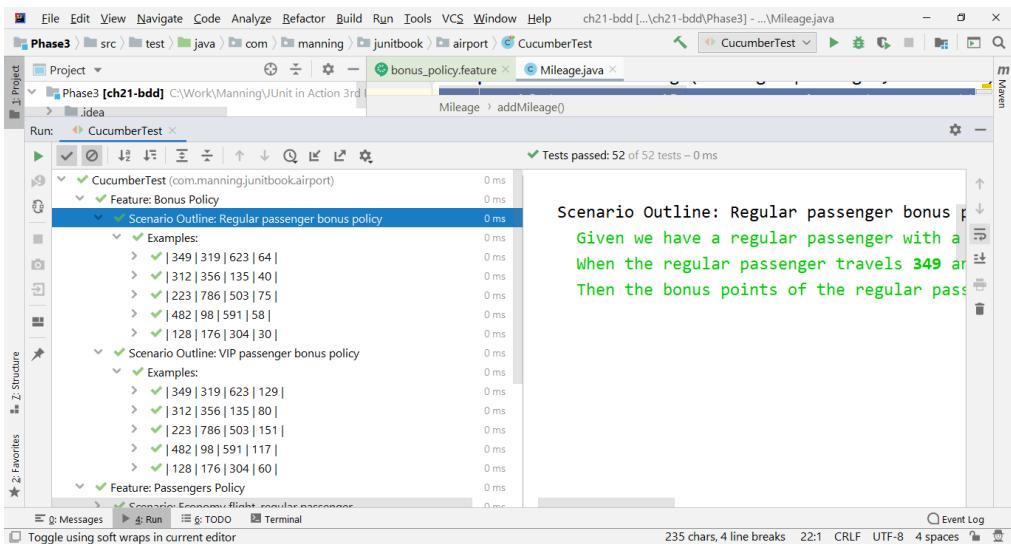


Figure 21.13 Running the bonus points tests after the business logic implementation will succeed

John has successfully implemented the bonus policy feature working BDD style, with JUnit 5 and Cucumber.

21.3 Working BDD with JBehave and JUnit 5

21.3.1 Introducing JBehave

There are a few alternatives in choosing a BDD framework. Besides the already introduced Cucumber, we'll also take a look at another very popular one, JBehave.

JBehave

JBehave is a Behavior Driven Development testing tool framework. As implementing the idea of Behavior Driven Development, JBehave allows us to write stories in plain text, to be understood by all persons involved in the project. Through the stories, we'll define scenarios that express the desired behavior.

Like other BDD frameworks, JBehave provides its terminology. We mention here:

- Story - covers one or more scenarios and represents an increment of business functionality that can be automatically executed.
- Scenario - a real-life situation to interact with the application.
- Step - this is defined using the classic BDD keywords: Given, When and Then.

21.3.2 Moving a TDD feature to JBehave

John would like to implement, with the help of JBehave, the same features, and tests that he has implemented with Cucumber. This will allow the possibility to make some comparisons between the two BDD frameworks and conclude which one to use.

John will first introduce the JBehave dependency into the Maven configuration (listing 21.11). He will first create a JBehave story, generate the tests skeleton and fill it in.

Listing 21.11 The JBehave dependency added to the pom.xml file

```
<dependency>
  <groupId>org.jbehave</groupId>
  <artifactId>jbehave-core</artifactId>
  <version>4.1</version>
</dependency>
```

Then John will install the plugins for IntelliJ. He goes to File -> Settings -> Plugins -> Browse Repositories, type JBehave, and choose JBehave Step Generator and JBehave Support (fig 21.14).

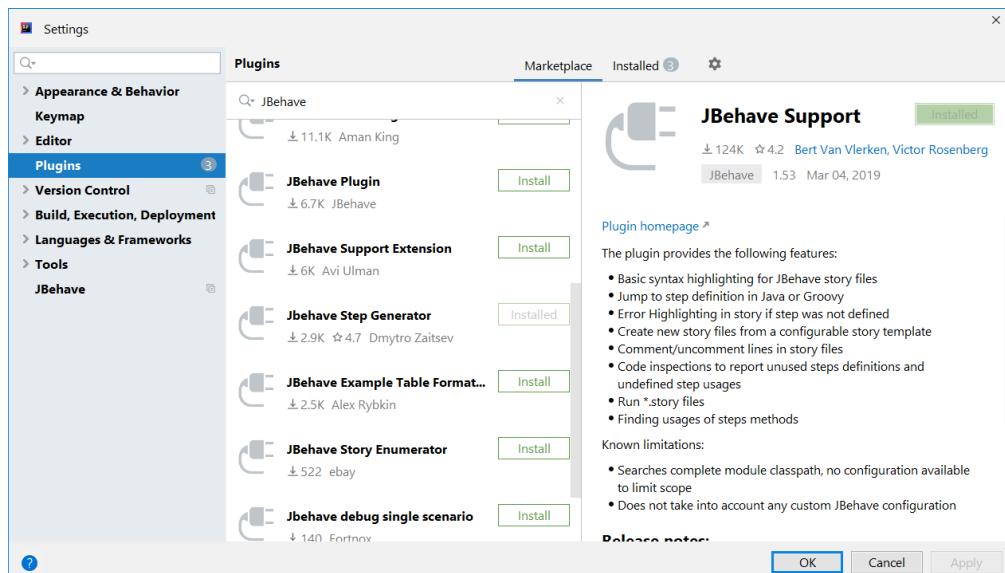


Figure 21.14 Installing the JBehave for Java plugins from the File -> Settings -> Plugins menu

John will start creating the story. He will follow the Maven standard folders structure and introduce the stories into the test/resources folder. He will create a folders structure com/manning/junitbook/airport and insert here the passengers_policy_story.story file. He will

also create, into the test folder, the com.manning.junitbook.airport package containing the PassengersPolicy class (fig. 21.15).

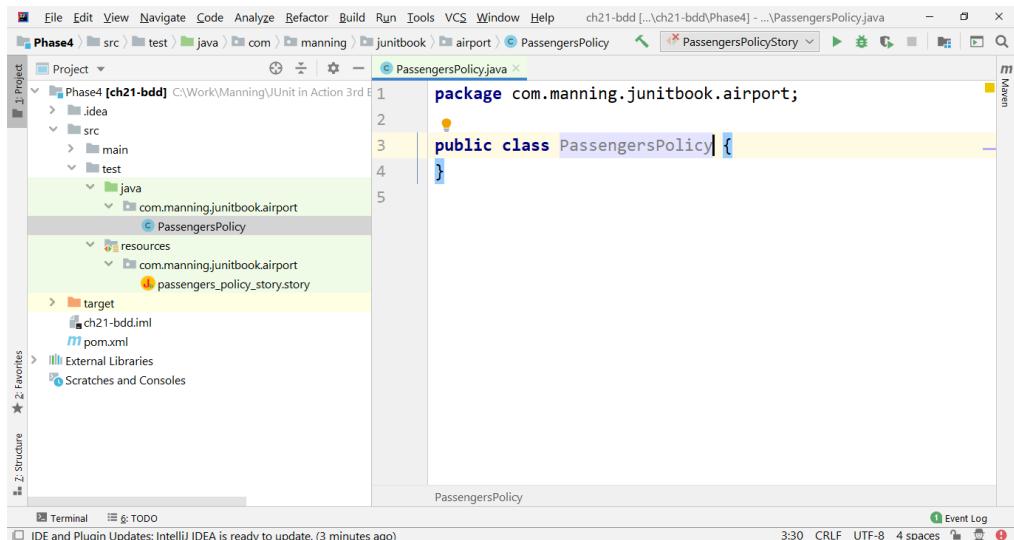


Figure 21.15 The newly introduced PassengersPolicy class corresponds to the story file to be found in the test/resources/com/manning/junitbook/airport folder

The story will contain meta-information about itself, the narrative (what it intends to do) and the scenarios (listing 21.12).

Listing 21.12 The passengers_policy_story.story file

```
Meta: Passengers Policy
The company follows a policy of adding and removing passengers,
depending on the passenger type and on the flight type

Narrative:
As a company
I want to be able to manage passengers and flights
So that the policies of the company are followed

Scenario: Economy flight, regular passenger
Given there is an economy flight
When we have a regular passenger
Then you can add and remove him from an economy flight
And you cannot add a regular passenger to an economy flight more than once

Scenario: Economy flight, VIP passenger
Given there is an economy flight
When we have a VIP passenger
Then you can add him but cannot remove him from an economy flight
And you cannot add a VIP passenger to an economy flight more than once
```

```

Scenario: Business flight, regular passenger
Given there is a business flight
When we have a regular passenger
Then you cannot add or remove him from a business flight

Scenario: Business flight, VIP passenger
Given there is a business flight
When we have a VIP passenger
Then you can add him but cannot remove him from a business flight
And you cannot add a VIP passenger to a business flight more than once

Scenario: Premium flight, regular passenger
Given there is a premium flight
When we have a regular passenger
Then you cannot add or remove him from a premium flight

Scenario: Premium flight, VIP passenger
Given there is a premium flight
When we have a VIP passenger
Then you can add and remove him from a premium flight
And you cannot add a VIP passenger to a premium flight more than once

```

In order to generate the steps into a Java file, John will place the cursor on any not yet created test step (they are underlined in red) and press Alt+Enter (fig. 21.16).

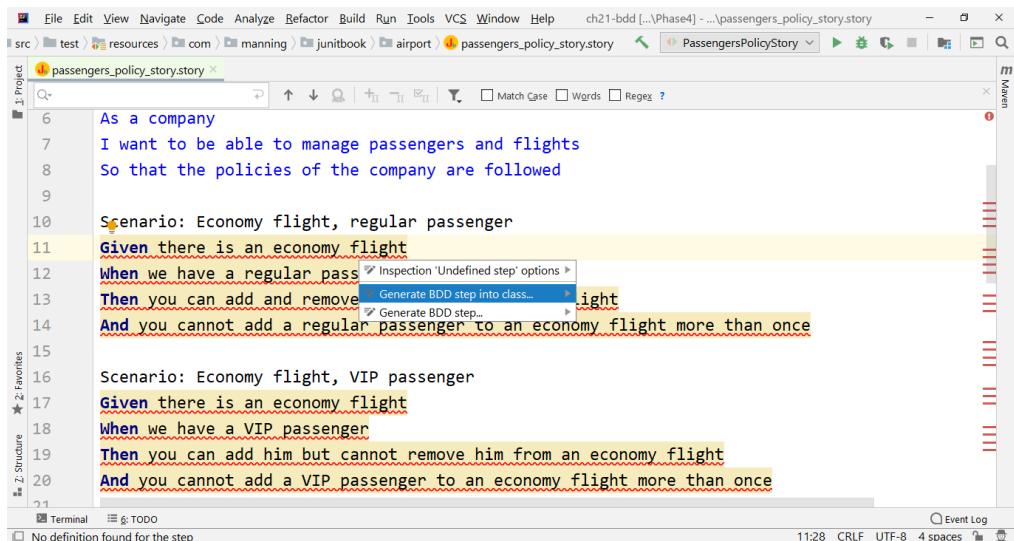


Figure 21.16 Pressing Alt+ Enter and generating the BDD steps into a class

He will generate all steps into the newly created `PassengersPolicy` class (fig. 21.17).

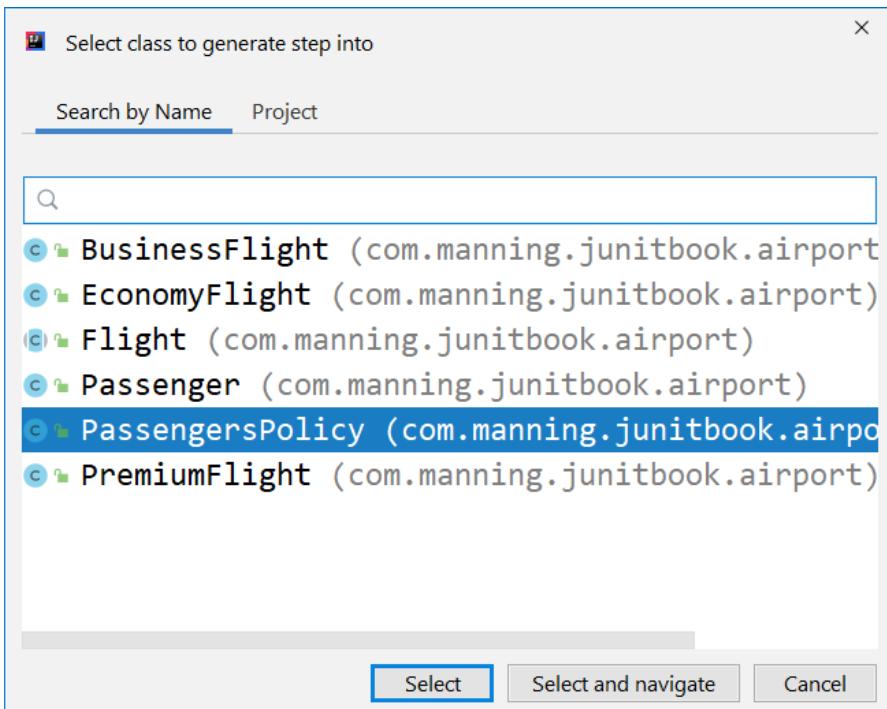


Figure 21.17 Choosing PassengersPolicy as the class to generate the steps of the story into

The skeleton will look as in listing 21.13, with the tests needing to be filled in.

Listing 21.13 The skeleton of the JBehave PassengersPolicy test

```
public class PassengersPolicy {  
    @Given("there is an economy flight")  
    public void givenThereIsAnEconomyFlight() {  
  
    }  
  
    @When("we have a regular passenger")  
    public void whenWeHaveARegularPassenger() {  
  
    }  
  
    @Then("you can add and remove him from an economy flight")  
    public void thenYouCanAddAndRemoveHimFromAnEconomyFlight() {  
  
    }  
    [...]  
}
```

John will now implement the tests according to the business logic, as in listing 21.14. He will write the code corresponding to each step defined through a method.

Listing 21.14 The implemented tests from PassengersPolicy

```
public class PassengersPolicy {  
    private Flight economyFlight; #A  
    private Passenger mike; #A  
    [...]  
  
    @Given("there is an economy flight") #B  
    public void givenThereIsAnEconomyFlight() {  
        economyFlight = new EconomyFlight("1"); #C  
    }  
  
    @When("we have a regular passenger") #D  
    public void whenWeHaveARegularPassenger() {  
        mike = new Passenger("Mike", false); #E  
    }  
  
    @Then("you can add and remove him from an economy flight") #F  
    public void thenYouCanAddAndRemoveHimFromAnEconomyFlight() {  
        assertAll("Verify all conditions for a regular passenger  
        and an economy flight", #G  
            () -> assertEquals("1", economyFlight.getId()), #G  
            () -> assertEquals(true, #G  
                economyFlight.addPassenger(mike)), #G  
            () -> assertEquals(1, #G  
                economyFlight.getPassengersSet().size()), #G  
            () -> assertEquals("Mike", new ArrayList<>( #G  
                economyFlight.  
                    getPassengersSet()).get(0).getName()), #G  
            () -> assertEquals(true, #G  
                economyFlight.removePassenger(mike)), #G  
            () -> assertEquals(0, #G  
                economyFlight.getPassengersSet().size()) #G  
        );  
    }  
    [...]  
}
```

In the listing above, we are doing the following:

- We declare the instance variables for the test, among which `economyFlight` and `mike` as a `Passenger` (#A).
- We write the method corresponding to the “Given there is an economy flight” business logic step (#B) by initializing the `economyFlight` (#C).
- We write the method corresponding to the “When we have a regular passenger” business logic step (#D) by initializing the regular passenger `mike` (#E).
- We write the method corresponding to the “Then you can add and remove him from an economy flight” business logic step (#F) by checking all the conditions by using the `assertAll` JUnit 5 method, which can now be read in a flow (#G).
- The rest of the methods are implemented in a similar way, we have covered the

Given, When and Then steps of one scenario.

In order to be able to run these tests, John will need a new special class that will represent the test configuration. He will name this class `PassengersPolicyStory` (listing 21.15).

Listing 21.15 The PassengersPolicyStory class

```
public class PassengersPolicyStory extends JUnitStory { #A

    @Override
    public Configuration configuration() { #B
        return new MostUsefulConfiguration()
            .useStoryReporterBuilder(
                new StoryReporterBuilder().
                    withFormats(Format.CONSOLE)); #D
    }

    @Override
    public InjectableStepsFactory stepsFactory() { #E
        return new InstanceStepsFactory(configuration(),
            new PassengersPolicy()); #F
    }
}
```

In the listing above, we are doing the following:

- We are declaring the `PassengersPolicyStory` class that is extending `JUnitStory` (#A). A JBehave story class must extend this `JUnitStory`.
- We override the `configuration` method (#B) and we tell that the configuration of the report is the one that works for the most situations that users are likely to encounter (#C), plus that the report will be displayed on the console (#D).
- We override the `stepsFactory` method (#E) and we tell that the steps definition is to be found in the `PassengersPolicy` class (#F).

The result of running these tests is shown in fig. 21.18. The tests are successfully running, the code coverage is 100%. However, the reporting capabilities of JBehave do not allow the same nice display as in the case of Cucumber.

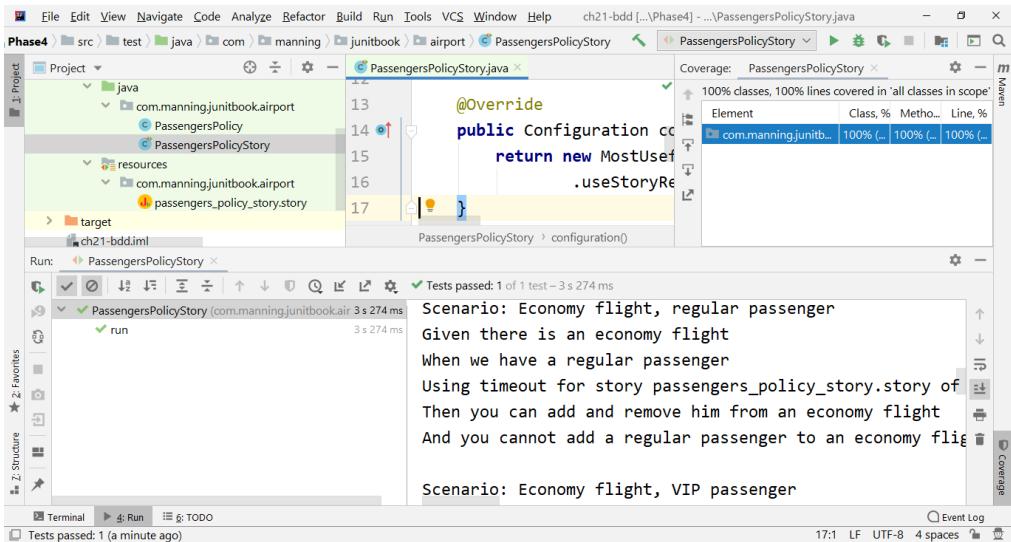


Figure 21.18 The JBehave passengers policy tests run successfully, the code coverage is 100%

We compare the length of the pre-BDD `AirportTest` class, which has 207 lines, with the one of this JBehave `PassengersPolicy` class, which has 157 lines (just like for the Cucumber version). The testing code is now at only 75% of the pre-BDD size, while we have shown that it has the same 100% coverage. Where does this gain come from? Remember that the `AirportTest` file contained 7 classes, on 3 levels: `AirportTest` at the top level; one `EconomyFlightTest` and one `BusinessFlightTest` at the second level; and, at the third level, two `RegularPassenger` and two `VipPassenger` classes. The code duplication is now really jumping to our attention, but that was the solution having only JUnit 5.

21.3.3 Adding a new feature with the help of JBehave

John would like to implement, with the help of JBehave, the same new feature concerning the policy of bonus points that are awarded to the passenger.

The specifications about calculating the bonus points consider the mileage, meaning the distance that is traveled by each passenger. The bonus will be calculated for all flights of the passenger and it depends on a factor: the mileage will be divided by 10 for VIP passenger and by 20 for regular ones.

John will define the scenarios of awarding the bonus points into the `bonus_policy_story.story` file (listing 21.16) and generate the JBehave tests that describe the scenarios. In the beginning, they are expected to fail.

Listing 21.16 The bonus_policy_story.story file

Meta: Bonus Policy

The company follows a bonus policy, depending on the passenger type and on the mileage

Narrative:

As a company

I want to be able to manage the bonus awarding

So that the policies of the company are followed

Scenario: Regular passenger bonus policy #A

Given we have a regular passenger with a mileage

When the regular passenger travels <mileage1> and <mileage2> and <mileage3> #B

Then the bonus points of the regular passenger should be <points> #B

Examples:

mileage1	mileage2	mileage3	points	#C
349	319	623	64	#C
312	356	135	40	#C
223	786	503	75	#C
482	98	591	58	#C
128	176	304	30	#C

Scenario: VIP passenger bonus policy #A

Given we have a VIP passenger with a mileage

When the VIP passenger travels <mileage1> and <mileage2> and <mileage3> #B

Then the bonus points of the VIP passenger should be <points> #B

Examples:

mileage1	mileage2	mileage3	points	#C
349	319	623	129	#C
312	356	135	80	#C
223	786	503	151	#C
482	98	591	117	#C
128	176	304	60	#C

In the listing above, we are doing the following:

- We introduce new scenarios for the bonus policy, using the Given, When and Then keywords (#A).
- Values are replaced with parameters as into the step-definition itself - you can see <mileage1>, <mileage2>, <mileage3> and <points> as parameters (#B).
- The effective values are defined in the Examples table, at the end of each Scenario (#C). The first row in the first table defines the values of 3 mileages (349, 319, 623). Adding them and dividing them by 20 (the regular passenger factor), we get the integer part 64 (the number of bonus points). This successfully replaces the JUnit 5 parameterized tests, having the advantage that the values are kept inside the scenarios, and easy to be understood by everyone.

John will create, into the test folder, the BonusPolicy class into the com.manning.junitbook.airport package (fig. 21.19).

```

package com.manning.junitbook.airport;

public class BonusPolicy {
}

```

Figure 21.19 The newly introduced BonusPolicy class corresponds to the story file to be found in the test/resources/com/manning/junitbook/airport folder

In order to generate the steps into a Java file, John will place the cursor on any not yet created step and press Alt+Enter (fig. 21.20).

```

Scenario: Regular passenger bonus policy
Given we have a regular passenger with a mileage
When the regular passenger travels <mileage>
Then the bonus points of the regular <points>

Examples:
| mileage1 | mileage2 | mileage3 | points |
| 349      | 319     | 623     | 64    |
| 312      | 356     | 135     | 40    |
| 223      | 786     | 503     | 75    |
| 482      | 98      | 591     | 58    |
| 128      | 176     | 304     | 30    |

Scenario: VIP passenger bonus policy
Given we have a VIP passenger with a mileage
When the VIP passenger travels <mileage1> and <mileage2> and <mileage3>
Then the bonus points of the VIP passenger <points>

```

Figure 21.20 Pressing Alt+ Enter and generating the BDD steps into a class

He will generate all steps into the newly created `BonusPolicy` class (fig. 21.21).

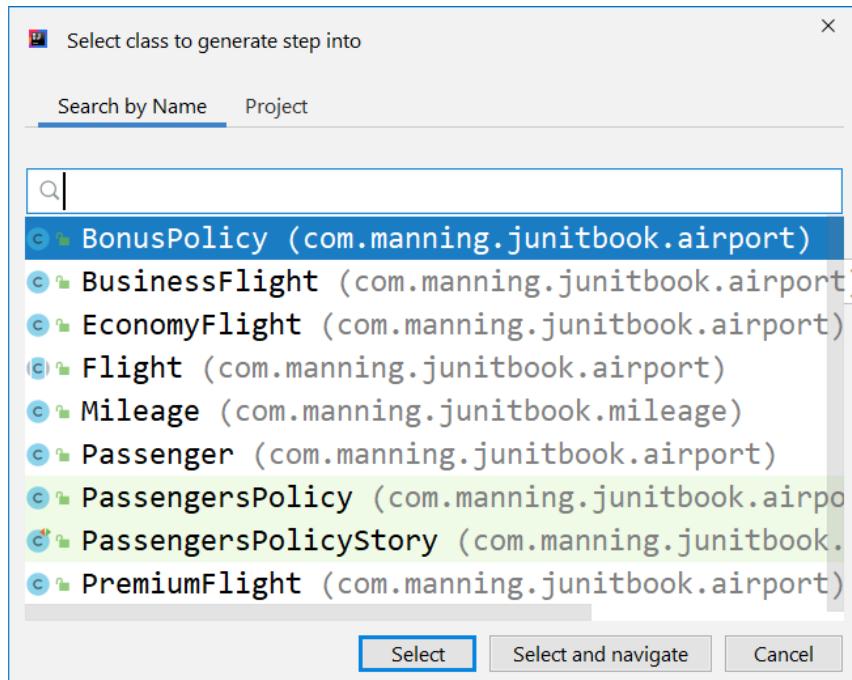


Figure 21.21 Choosing `BonusPolicy` as the class to generate the steps of the story into

The `BonusPolicy` class is shown in listing 21.17 with the tests skeleton filled in.

Listing 21.17 The skeleton of the JBehave `BonusPolicy` test

```
public class BonusPolicy {

    @Given("we have a regular passenger with a mileage") #A
    public void givenWeHaveARegularPassengerWithAMileage() {
    }

    @When("the regular passenger travels <mileage1> and
          <mileage2> and <mileage3>") #B #B
    public void whenTheRegularPassengerTravelsMileageAndMileageAndMileage(
        @Named("mileage1") int mileage1, @Named("mileage2") int mileage2,
        @Named("mileage3") int mileage3) {
    }

    @Then("the bonus points of the regular passenger should be <points>") #C
    public void thenTheBonusPointsOfTheRegularPassengerShouldBePoints(
        @Named("points") int points) {
}
```

```
    }
    [...]
}
```

In the listing above we are doing the following:

- The JBehave plugin has generated a method annotated with @Given("we have a regular passenger with a mileage"), meaning that this method will be executed when the step "Given we have a regular passenger with a mileage" from the scenario will be executed (#A).
- The JBehave plugin has generated a method annotated with @When("the regular passenger travels <mileage1> and <mileage2> and <mileage3>"), meaning that this method will be executed when the step "When the regular passenger travels <mileage1> and <mileage2> and <mileage3>" from the scenario will be executed (#B).
- The JBehave plugin has generated a method annotated with @Then("the bonus points of the regular passenger should be <points>"), meaning that this method will be executed when the step "Then the bonus points of the regular passenger should be <points>" from the scenario will be executed (#C).
- The rest of the methods generated by the JBehave plugin are similar, we have covered the Given, When and Then steps of one scenario.

Now, John will create the Mileage class, declaring the fields and the methods, but not implementing them yet (listing 21.18). John needs to use the methods of this class for the tests, make these tests initially fail, then implement the methods and make the tests pass.

Listing 21.18 The Mileage class, with no implementation of the methods

```
public class Mileage {

    public static final int VIP_FACTOR = 10;                      #A
    public static final int REGULAR_FACTOR = 20;                  #A

    private Map<Passenger, Integer> passengersMileageMap = new HashMap<>(); #B
    private Map<Passenger, Integer> passengersPointsMap = new HashMap<>(); #B

    public void addMileage(Passenger passenger, int miles) {        #C
    }

    public void calculateGivenPoints() {                            #D
    }

}
```

In the listing above, we are doing the following:

- We have declared the VIP_FACTOR and REGULAR_FACTOR constants, corresponding to the factor by which we divide the mileage for each type of passenger in order to get

the bonus points (#A).

- We have declared `passengersMileageMap` and `passengersPointsMap`, two maps having as key the passenger and keeping as value the mileage and the points for that passenger, respectively (#B).
- We have declared the `addMileage` method, which will populate the `passengersMileageMap` with the mileage for each passenger (#C). The method does not do anything, for now, it will be written later, to fix the tests.
- We have declared the `calculateGivenPoints` method, which will populate the `passengersPointsMap` with the bonus points for each passenger (#D). The method does not do anything, for now, it will be written later, to fix the tests.

John will now turn his attention to write the unimplemented tests from the `BonusPolicy` class, to follow the business logic of this feature (listing 21.19).

Listing 21.19 The business logic of the steps from `BonusPolicy`

```
public class BonusPolicy {  
    private Passenger mike; #A  
    private Mileage mileage; #A  
    [...]  
  
    @Given("we have a regular passenger with a mileage") #B  
    public void givenWeHaveARegularPassengerWithAMileage() {  
        mike = new Passenger("Mike", false); #C  
        mileage = new Mileage(); #C  
    }  
  
    @When("the regular passenger travels <mileage1> and <mileage2> and  
          <mileage3>") #D  
    public void the_regular_passenger_travels_and_and(@Named("mileage1")  
                                                    int mileage1, @Named("mileage2") int mileage2,  
                                                    @Named("mileage3") int mileage3) {  
        mileage.addMileage(mike, mileage1); #E  
        mileage.addMileage(mike, mileage2); #E  
        mileage.addMileage(mike, mileage3); #E  
    }  
  
    @Then("the bonus points of the regular passenger should be <points>") #F  
    public void the_bonus_points_of_the_regular_passenger_should_be  
        (@Named("points") int points) {  
        mileage.calculateGivenPoints(); #G  
        assertEquals(points, #H  
                    mileage.getPassengersPointsMap().get(mike).intValue()); #H  
    }  
    [...]  
}
```

In the listing above, we are doing the following:

- We declare the instance variables for the test, among which `mileage` and `mike` as a `Passenger` (#A).
- We write the method corresponding to the "Given we have a regular passenger with a mileage" business logic step (#B) by initializing the passenger

and the mileage (#C).

- We write the method corresponding to the “When the regular passenger travels <mileage1> and <mileage2> and <mileage3>” business logic step (#D) by adding mileages to the regular passenger mike (#E).
- We write the method corresponding to the “Then the bonus points of the regular passenger should be <points>” business logic step (#F) by calculating the given points (#G) and checking that the calculated value is the expected one (#H).
- The rest of the methods are implemented in a similar way, we have covered the Given, When and Then steps of one scenario.

In order to be able to run these tests, John will need a new special class that will represent the test configuration. He will name this class `BonusPolicyStory` (listing 21.20).

Listing 21.20 The BonusPolicyStory class

```
public class BonusPolicyStory extends JUnitStory { #A

    @Override
    public Configuration configuration() {
        return new MostUsefulConfiguration()
            .useStoryReporterBuilder(
                new StoryReporterBuilder().
                    withFormats(Format.CONSOLE)); #B #C #D #D #D
    }

    @Override
    public InjectableStepsFactory stepsFactory() {
        return new InstanceStepsFactory(configuration(),
            new BonusPolicy()); #E #F #F
    }
}
```

In the listing above, we are doing the following:

- We are declaring the `BonusPolicyStory` class that is extending `JUnitStory` (#A). A JBehave story class must extend this `JUnitStory`.
- We override the `configuration` method (#B) and we tell that the configuration of the report is the one that works for the most situations that users are likely to encounter (#C), plus that the report will be displayed on the console (#D).
- We override the `stepsFactory` method (#E) and we tell that the steps definition is to be found into the `BonusPolicy` class (#F).

If we run the bonus points tests now they will fail (fig. 21.22), as the business logic is not yet implemented (we have left the `addMileage` and `calculateGivenPoints` methods empty).

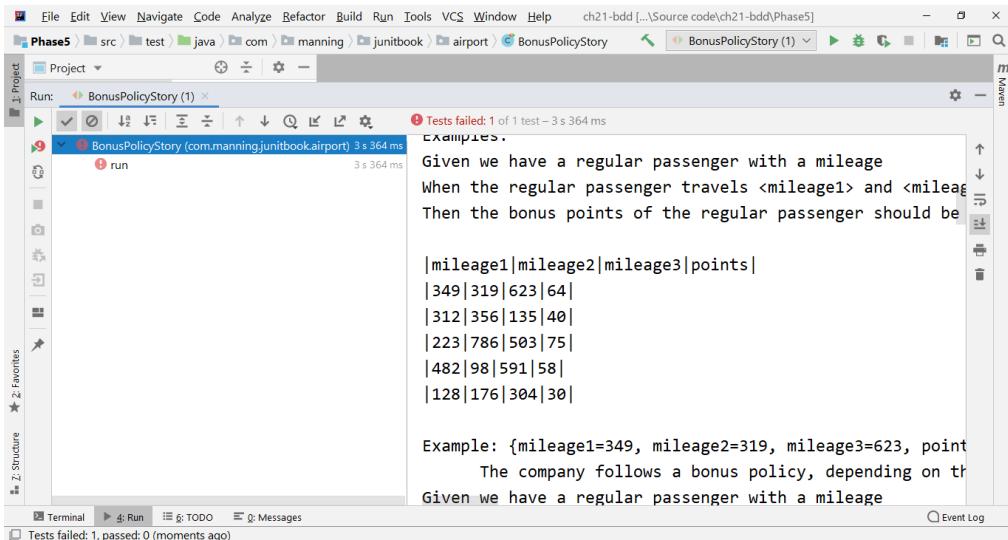


Figure 21.22 Running the JBehave bonus points tests before the business logic implementation will fail

John will move back to the implementation of the two remaining business logic methods from the `Mileage` class (`addMileage` and `calculateGivenPoints`), as shown in listing 21.21.

Listing 21.21 The implementation of business logic from the Mileage class

```
public void addMileage(Passenger passenger, int miles) {
    if (passengersMileageMap.containsKey(passenger)) {
        passengersMileageMap.put(passenger,
            passengersMileageMap.get(passenger) + miles);
    } else {
        passengersMileageMap.put(passenger, miles);
    }
}

public void calculateGivenPoints() {
    for (Passenger passenger : passengersMileageMap.keySet()) {
        if (passenger.isVip()) {
            passengersPointsMap.put(passenger,
                passengersMileageMap.get(passenger) / VIP_FACTOR);
        } else {
            passengersPointsMap.put(passenger,
                passengersMileageMap.get(passenger) / REGULAR_FACTOR);
        }
    }
}
```

In the listing above, we are doing the following:

- In the `addMileage` method, we check if the `passengersMileageMap` already

contains a passenger (#A). If that passenger already exists, we add the mileage to him (#B), otherwise, we create a new entry into the map having that passenger as a key and the miles as the initial value (#C).

- In the calculateGivenPoints method, we browse the passengers set (#C) and, for each passenger, if he is a VIP, we calculate the bonus points by dividing the mileage to the VIP factor (#D). Otherwise, we calculate the bonus points by dividing the mileage to the regular factor (#E).

Running the bonus points tests now will be successful, as shown in fig. 21.23.

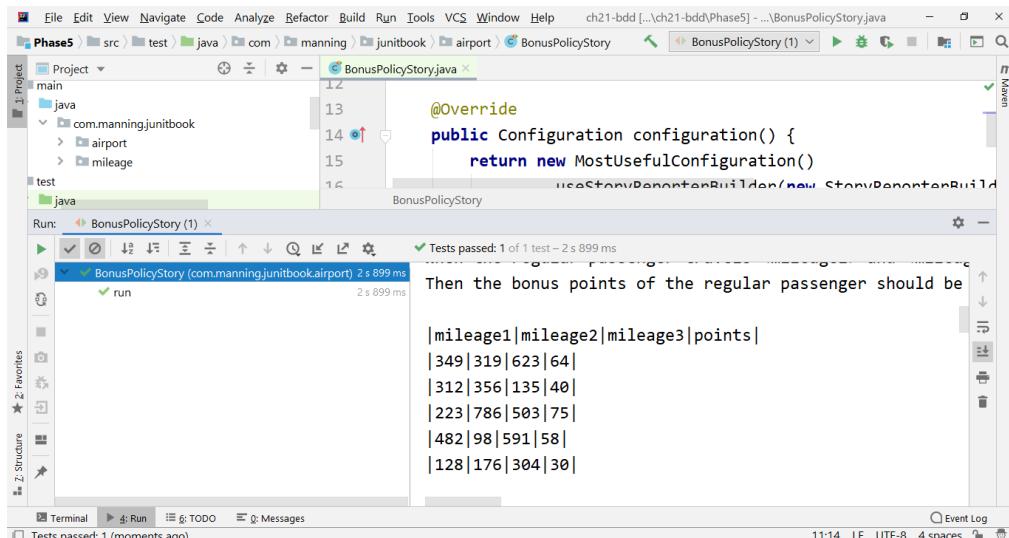


Figure 21.23 Running the JBehave bonus points tests after the business logic implementation will succeed

John has successfully implemented the bonus policy feature working BDD style, with JUnit 5 and JBehave.

21.4 Comparing Cucumber and JBehave

Cucumber and JBehave have similar approaches, supporting the Behaviour Driven Development ideas. Cucumber and JBehave are different frameworks but built on the same well defined BDD principles that we have emphasized.

They are based around features (Cucumber) or stories (JBehave). A feature is a collection of stories, expressed from the point of view of a specific project stakeholder.

You may notice the same BDD keywords Given, When, Then, but also some variations in names like "Scenario Outline" for Cucumber, or simply Scenario with Examples for JBehave.

The IntelliJ IDE support for both Cucumber and JBehave is provided through plugins that help from the steps generation in Java code up to checking the code coverage. The Cucumber plugin has been able to produce a nicer output, allowing us to follow the full testing hierarchy at a glance, and displaying everything with significant colors. More, it allowed us to directly run a test from the feature text file, which is easier to follow, especially by non-technical persons.

JBehave has reached its maturity phase some time ago, while the Cucumber codebase is still updated very frequently. At the time of writing this chapter, the Cucumber Github code displays tens of commits from within the previous 7 days. The JBehave Github code displays one single commit within the last 7 days.

Cucumber has also a more active community at the time of writing this chapter, the articles on blogs and forums are more recent and frequent. Consequently, this makes troubleshooting easier for developers.

In terms of code size compared to the pre-BDD situation, both Cucumber and JBehave have shown similar performances, reducing the size of the initial code in the same proportion.

Choosing one of these frameworks may be a matter of habit or preference - personal preference or project preference. We have intended to put them face to face in a very practical and comparable way so that you can eventually make your own choice.

The next chapter will be dedicated to building a test pyramid strategy, from the low level to the high level, and applying it in working with JUnit 5.

21.5 Summary

This chapter has covered the following:

- Introducing Behavior Driven Development, a software development technique that encourages teams to deliver software that matters, supporting the cooperation between stakeholders.
- Analyzing the benefits of Behavior Driven Development: addressing user needs, clarity, change support, automation support, focus on adding business value, costs reduction.
- Analyzing the challenges of Behavior Driven Development: it requires engagement and strong collaboration, interaction, direct communication, and constant feedback.
- Moving a TDD application to BDD, by moving the passengers policy business logic to be implemented with the help of both Cucumber and JBehave.
- Developing the business logic for the bonus policy with the help of Cucumber, by creating a separate feature, generating the skeleton of the testing code, writing the tests and implementing the code.
- Developing the business logic for the bonus policy with the help of JBehave, by creating a separate story, generating the skeleton of the testing code, writing the tests and implementing the code.
- Comparing Cucumber and JBehave in terms of approaching the BDD principles, ease of use, code size needed to implement some functionality.

22

Implementing a test pyramid strategy with JUnit 5

This chapter covers:

- Introducing the software testing levels
- Building unit tests for the components working in isolation
- Building integration tests for the units combined as a group
- Building system tests and looking at the complete software
- Building acceptance tests and making sure the software is compliant with the business requirements

The test pyramid is a way of thinking about different kinds of automated tests should be used to create a balanced portfolio. Its essential point is that you should have many more low-level UnitTests than high-level BroadStackTests running through a GUI.

- Martin Fowler

As we have discovered in the previous chapters, software testing has a few purposes. Tests will make us interact with the application and understand how it works. Testing helps us delivering software that meets expectations. It is a metric of the quality of the code and protects us against the possibility of introducing regressions. Consequently, effectively and systematically organizing the process of software testing is really important.

22.1 Software testing levels

Describing the levels of software testing, we'll enumerate the following (from the lowest to the highest):

- Unit testing - methods or classes (meaning individual units) are tested to determine whether they are working correctly.
- Integration testing - the individual software components are combined and tested together.
- System testing - testing is performed on a complete, full system, in order to evaluate the system compliance with the specification.
- Acceptance testing - an application is verified to satisfy the expectations of the end-user.

Low-level testing addresses individual components, it is concerned more with the details and less by the overview. High-level testing is more abstract, it verifies the overall goals and features of the system. It is more focused on the user – GUI interaction and how the system works as a whole.

If we ask ourselves what to test, we identify the following:

- Business logic - how, the program transposes the business rules from the real-world.
- Bad input values – e.g., we cannot assign a negative number of seats to a flight.
- Boundary conditions - extremes of some input domain, e.g. maximum, minimum. We may test on flights having 0 passengers or the maximum allowed passengers.
- Unexpected conditions - conditions that are not part of the normal operation of a program. A flight cannot change its origin once it has taken off.
- Invariants - expressions whose values do not change during program execution, e.g., the identifier of a person cannot change during the execution of the program.
- Regressions - bugs introduced in an existing system after upgrades- or patches.

We have introduced the different levels of tests, and they may be regarded as a pyramid (fig. 22.1).

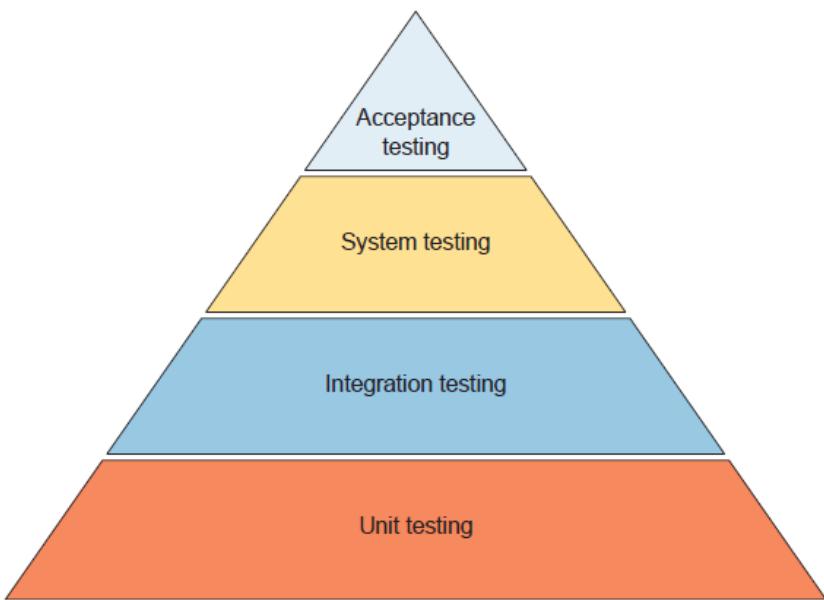


Figure 22.1 A pyramid possesses a larger ground (where the unit tests sit), while the higher testing level becomes smaller

- The unit testing is at its foundation. It focuses on each piece of software by testing it in isolating, to determine if it is according to expectations.
- Integration testing takes its input from already verified units, groups them in larger aggregates and executes integration tests on them.
- System testing requires no knowledge of the design or of the code but focuses on the functionality of the whole system.
- Acceptance testing use scenarios and test cases to check if the application satisfies the expectations of the end-user.

These levels represent a hierarchy built from simple to complex and also a view of the development process, from its beginning to the later phases.

22.2 Unit testing – our basic components work in isolation

Tested Data Systems Inc. is an outsourcing company creating software projects for different companies. We demonstrate in this chapter how Thomas, a developer in this company, will build a test pyramid strategy on a flights management application.

The application that Thomas is taking over is composed of two classes: Passenger (listing 22.1) and Flight (listing 22.2).

Listing 22.1 The Passenger class

```
public class Passenger {  
  
    private String identifier; #A  
    private String name; #A  
    private String countryCode; #A  
    private String ssnRegex = #B  
        "^(?![000|666][0-8][0-9]{2}-(?!00)[0-9]{2}-(?!0000)[0-9]{4}$"; #B  
    private String nonUsIdentifierRegex = #C  
        "^(?![000|666][9][0-9]{2}-(?!00)[0-9]{2}-(?!0000)[0-9]{4}$"; #C  
    private Pattern pattern; #D  
  
    public Passenger(String identifier, String name, String countryCode) { #E  
        pattern = countryCode.equals("US")? Pattern.compile(ssnRegex): #F  
            Pattern.compile(nonUsIdentifierRegex); #F  
        Matcher matcher = pattern.matcher(identifier); #G  
        if(!matcher.matches()) { #G  
            throw new RuntimeException("Invalid identifier"); #G  
        } #G  
  
        if(!Arrays.asList(Locale.getISOCountries()).contains(countryCode)) { #H  
            throw new RuntimeException("Invalid country code"); #H  
        } #H  
  
        this.identifier = identifier; #I  
        this.name = name; #I  
        this.countryCode = countryCode; #I  
    } #I  
  
    public String getIdentifier() { #J  
        return identifier; #J  
    } #J  
  
    public void setIdentifier(String identifier) { #K  
        pattern = countryCode.equals("US")? Pattern.compile(ssnRegex): #K  
            Pattern.compile(nonUsIdentifierRegex); #K  
        Matcher matcher = pattern.matcher(identifier); #K  
        if(!matcher.matches()) { #K  
            throw new RuntimeException("Invalid identifier"); #K  
        } #K  
  
        this.identifier = identifier; #K  
    } #K  
  
    public String getName() { #J  
        return name; #J  
    } #J  
  
    public void setName(String name) { #J  
        this.name = name; #J  
    } #J  
  
    public String getCountryCode() { #J  
        return countryCode; #J  
    } #J  
  
    public void setCountryCode(String countryCode) { #L  
    } #L
```

```

        if(!Arrays.asList(Locale.getISOCountries()).contains(countryCode)) { #L
            throw new RuntimeException("Invalid country code");
        } #L

        this.countryCode = countryCode; #L
    }

@Override #M
public String toString() { #M
    return "Passenger " + getName() + " with identifier: " #M
           + getIdentifier() + " from " + getCountryCode(); #M
}
} #M
}

```

In the listing above, we are doing the following:

- We declare the identifier, name and countryCode instance variables for a Passenger (#A).
- If the passenger is a US citizen, the identifier is the SSN (Social Security Number) itself. ssnRegex describes the regular expression to be followed by such an SSN. An SSN must conform to the following rules, the first three digits cannot be 000, 666, or between 900 and 999 (#B).
- If the passenger is not a US citizen, the identifier is generated inside the company, following some similar rules as above. nonUsIdentifierRegex will allow identifiers with the first digits only between 900 and 999 (#C). We also declare a pattern that will check if an identifier is following the rules (#D).
- In the constructor (#E), we create the pattern (#F), check if the identifier matches it (#G), check if the country code is valid (#H), and then we may construct the passenger (#I).
- We provide getters and setters for the fields (#J), checking the validity of the input data for an identifier that has to match the pattern (#K) and that the country code exists (#L).
- We also override the `toString` method, to include the name, the identifier and the country code of the passenger (#M).

Listing 22.2 The Flight class

```

public class Flight {

    private String flightNumber; #A
    private int seats; #A
    private int passengers; #A
    private String origin; #A
    private String destination; #A
    private boolean flying; #A
    private boolean takenOff; #A
    private boolean landed; #A

    private String flightNumberRegex = "[A-Z]{2}\\d{3,4}"; #B
    private Pattern pattern = Pattern.compile(flightNumberRegex); #C
}

```

```

public Flight(String flightNumber, int seats) { #D
    Matcher matcher = pattern.matcher(flightNumber); #D
    if(!matcher.matches()) { #D
        throw new RuntimeException("Invalid flight number"); #D
    }
    this.flightNumber = flightNumber; #E
    this.seats = seats; #E
    this.passengers = 0; #E
    this.flying = false; #E
    this.takenOff = false; #E
    this.landed = false; #E
}

public String getFlightNumber() { #F
    return flightNumber; #F
}

public int getSeats() { #F
    return seats; #F
}

public void setSeats(int seats) { #G
    if(passengers > seats) { #G
        throw new RuntimeException("Cannot reduce the number of seats #G
            under the number of existing passengers!"); #G
    }
    this.seats = seats; #G
}

public int getPassengers() { #F
    return passengers; #F
}

public String getOrigin() { #F
    return origin; #F
}

public void setOrigin(String origin) { #H
    if(takenOff){ #H
        throw new RuntimeException("Flight cannot change its origin #H
            any longer!"); #H
    }
    this.origin = origin; #H
}

public String getDestination() { #F
    return destination; #F
}

public void setDestination(String destination) { #I
    if(landed){ #I
        throw new RuntimeException("Flight cannot change its #I
            destination any longer!"); #I
    }
    this.destination = destination; #I
}

```

```

}

public boolean isFlying() { #F
    return flying; #F
}

public boolean isTakenOff() { #F
    return takenOff; #F
}

public boolean isLanded() { #F
    return landed; #F
}

@Override #J
public String toString() { #J
    return "Flight " + getFlightNumber() + " from " + getOrigin() + #J
        " to " + getDestination(); #J
}

public void addPassenger() { #K
    if(passengers >= seats) { #K
        throw new RuntimeException("Not enough seats!"); #K
    }
    passengers++; #K
}

public void takeOff() { #L
    System.out.println(this + " is taking off"); #L
    flying = true; #L
    takenOff = true; #L
}

public void land() { #M
    System.out.println(this + " is landing"); #M
    flying = false; #M
    landed = true; #M
}
}

```

In the listing above, we are doing the following:

- We declare the instance variables for a Flight (#A).
- The flight number needs to match a regular expression: a code for an airline service consists of a two-character airline designator and a 3 to 4 digit number (#B). We also declare a pattern that will check if a flight number is following the rules (#C).
- In the constructor, we check if the flight number matches the pattern (#D), and then we may construct the flight (#E).
- We provide getters and setters for the fields (#F), checking that we cannot have more passengers than seats (#G), we cannot change the origin after the plane has taken off (#H), and we cannot change the destination after the plane has landed (#I).
- We override the `toString` method to display the flight number, the origin and the destination (#J).

- When we add a passenger to the plane, we check to have enough seats (#K).
- When the plane is taking off, we print a message and change the state of the plane (#L). We also print a message and we change the state of the plane when the plane is landing (#M).

The functionality of the Passenger class is verified in the PassengerTest class in listing 22.3.

Listing 22.3 The PassengerTest class

```
public class PassengerTest {

    @Test
    public void testPassengerCreation() {
        Passenger passenger = new Passenger("123-45-6789",
                                             "John Smith", "US");
        assertNotNull(passenger);
    }

    @Test
    public void testNonUsPassengerCreation() {
        Passenger passenger = new Passenger("900-45-6789",
                                             "John Smith", "GB");
        assertNotNull(passenger);
    }

    @Test
    public void testInvalidSsn() {
        assertThrows(RuntimeException.class,
                    ()->{
                        Passenger passenger = new Passenger("123-456-789",
                                                             "John Smith", "US");
                    });
        assertThrows(RuntimeException.class,
                    ()->{
                        Passenger passenger = new Passenger("900-45-6789",
                                                             "John Smith", "US");
                    });
    }

    @Test
    public void testInvalidNonUsIdentifier() {
        assertThrows(RuntimeException.class,
                    ()->{
                        Passenger passenger = new Passenger("900-456-789",
                                                             "John Smith", "GB");
                    });
        assertThrows(RuntimeException.class,
                    ()->{
                        Passenger passenger = new Passenger("123-45-6789",
                                                             "John Smith", "GB");
                    });
    }

    @Test
    public void testInvalidCountryCode() {
        assertThrows(RuntimeException.class,
                    ()->{

```

```

        Passenger passenger = new Passenger("900-45-6789",      #E
                                              "John Smith", "GJ");
                                              #E
                                              #E
    });

@Test
public void testSetInvalidSsn() {                                #F
    assertThrows(RuntimeException.class,                         #F
                 ()->{                                         #F
                    Passenger passenger = new Passenger("123-45-6789",   #F
                                                 "John Smith", "US");   #F
                    passenger.setIdentifier("123-456-789");           #F
                });
}

@Test
public void testSetValidSsn() {                                #G
    Passenger passenger = new Passenger("123-45-6789",      #G
                                         "John Smith", "US");   #G
    passenger.setIdentifier("123-98-7654");                  #G
    assertEquals("123-98-7654", passenger.getIdentifier());   #G
}

@Test
public void testSetValidNonUsIdentifier() {                      #H
    Passenger passenger = new Passenger("900-45-6789",      #H
                                         "John Smith", "GB");   #H
    passenger.setIdentifier("900-98-7654");                  #H
    assertEquals("900-98-7654", passenger.getIdentifier());   #H
}

@Test
public void testSetInvalidCountryCode() {                        #I
    assertThrows(RuntimeException.class,                         #I
                 ()->{                                         #I
                    Passenger passenger = new Passenger("123-45-6789",   #I
                                                 "John Smith", "US");   #I
                    passenger.setCountryCode("GJ");                   #I
                });
}

@Test
public void testSetValidCountryCode() {                          #J
    Passenger passenger = new Passenger("123-45-6789",      #J
                                         "John Smith", "US");   #J
    passenger.setCountryCode("GB");                            #J
    assertEquals("GB", passenger.getCountryCode());          #J
}

@Test
public void testPassengerToString() {                           #K
    Passenger passenger = new Passenger("123-45-6789",      #K
                                         "John Smith", "US");   #K
    passenger.setName("John Brown");                         #K
    assertEquals("Passenger John Brown with identifier:     #K
                 123-45-6789 from US", passenger.toString());   #K
}

```

```
}
```

In the listing above, we are doing the following:

- We check the correct creation of a US passenger (#A) and of a non-US passenger (#B), with correct identifiers.
- We check that we cannot set an invalid identifier for a US citizen (#C) and for a non-US citizen (#D).
- We check that we cannot set an invalid country code (#E) and an invalid SSN (#F).
- We check that we can set a valid SSN for a US citizen (#G) and a valid identifier for a non-US citizen (#H).
- We check setting an invalid country code (#I) and a valid country code (#J).
- We check the behavior of the `toString` method (#K).

The functionality of the `Flight` class is verified in the `FlightTest` class in listing 22.4.

Listing 22.4 The FlightTest class

```
public class FlightTest {

    @Test
    public void testFlightCreation() {
        Flight flight = new Flight("AA123", 100); #A
        assertNotNull(flight); #A
    } #A

    @Test
    public void testInvalidFlightNumber() {
        assertThrows(RuntimeException.class, #B
            ()->{
                Flight flight = new Flight("AA12", 100); #B
            });
        assertThrows(RuntimeException.class, #B
            ()->{
                Flight flight = new Flight("AA12345", 100); #B
            });
    } #B

    @Test
    public void testValidFlightNumber() {
        Flight flight = new Flight("AA345", 100); #C
        assertNotNull(flight);
        flight = new Flight("AA3456", 100); #C
        assertNotNull(flight); #C
    } #C

    @Test
    public void testAddPassengers() {
        Flight flight = new Flight("AA1234", 50); #D
        flight.setOrigin("London");
        flight.setDestination("Bucharest");
        for(int i=0; i<flight.getSeats(); i++) {
            flight.addPassenger(); #D
        }
    } #D
}
```

```

        }
        assertEquals(50, flight.getPassengers()); #D
        assertThrows(RuntimeException.class, #D
            ()->{ #D
                flight.addPassenger(); #D
            });
        });

}

@Test
public void testSetInvalidSeats() { #E
    Flight flight = new Flight("AA1234", 50); #E
    flight.setOrigin("London"); #E
    flight.setDestination("Bucharest"); #E
    for(int i=0; i<flight.getSeats(); i++) { #E
        flight.addPassenger(); #E
    } #E
    assertEquals(50, flight.getPassengers()); #E
    assertThrows(RuntimeException.class, #E
        ()->{ #E
            flight.setSeats(49); #E
        });
}

@Test
public void testSetValidSeats() { #F
    Flight flight = new Flight("AA1234", 50); #F
    flight.setOrigin("London"); #F
    flight.setDestination("Bucharest"); #F
    for(int i=0; i<flight.getSeats(); i++) { #F
        flight.addPassenger(); #F
    } #F
    assertEquals(50, flight.getPassengers()); #F
    flight.setSeats(52); #F
    assertEquals(52, flight.getSeats()); #F
}

@Test
public void testChangeOrigin() { #G
    Flight flight = new Flight("AA1234", 50); #G
    flight.setOrigin("London"); #G
    flight.setDestination("Bucharest"); #G
    flight.takeOff(); #G
    assertEquals(true, flight.isFlying()); #G
    assertEquals(true, flight.isTakenOff()); #G
    assertEquals(false, flight.isLanded()); #G
    assertThrows(RuntimeException.class, #G
        ()->{ #G
            flight.setOrigin("Manchester"); #G
        });
}

@Test
public void testChangeDestination() { #H
    Flight flight = new Flight("AA1234", 50); #H
    flight.setOrigin("London"); #H
    flight.setDestination("Bucharest"); #H
    flight.takeOff(); #H
    flight.land(); #H
}

```

```

        assertThrows(RuntimeException.class, #H
            ()->{ #H
                flight.setDestination("Sibiu"); #H
            }); #H
    }

@Test #I
public void testLand() { #I
    Flight flight = new Flight("AA1234", 50); #I
    flight.setOrigin("London"); #I
    flight.setDestination("Bucharest"); #I
    flight.takeOff(); #I
    assertEquals(true, flight.isTakenOff()); #I
    assertEquals(false, flight.isLanded()); #I
    flight.land(); #J
    assertEquals(true, flight.isTakenOff()); #J
    assertEquals(true, flight.isLanded()); #J
    assertEquals(false, flight.isFlying()); #J
}

}

```

In the listing above, we are doing the following:

- We check a flight creation (#A).
- We check that we cannot set an invalid flight number (#B), but we can set a valid one (#C).
- We check that we can add passengers only within the limit of seats (#D).
- We check that we cannot set the number of seats below the number of passengers (#E), but we can set it above the number of passengers (#F).
- We test that we cannot change the origin after the plane has taken off (#G) and that we cannot change the destination after the plane has landed (#H).
- We check that the plane has changed its state after having taken off (#I) and has changed its state again after landing (#J).

The result of running the unit tests for the Passenger and Flight classes is successful and the code coverage is 100%, as shown in fig. 22.2.

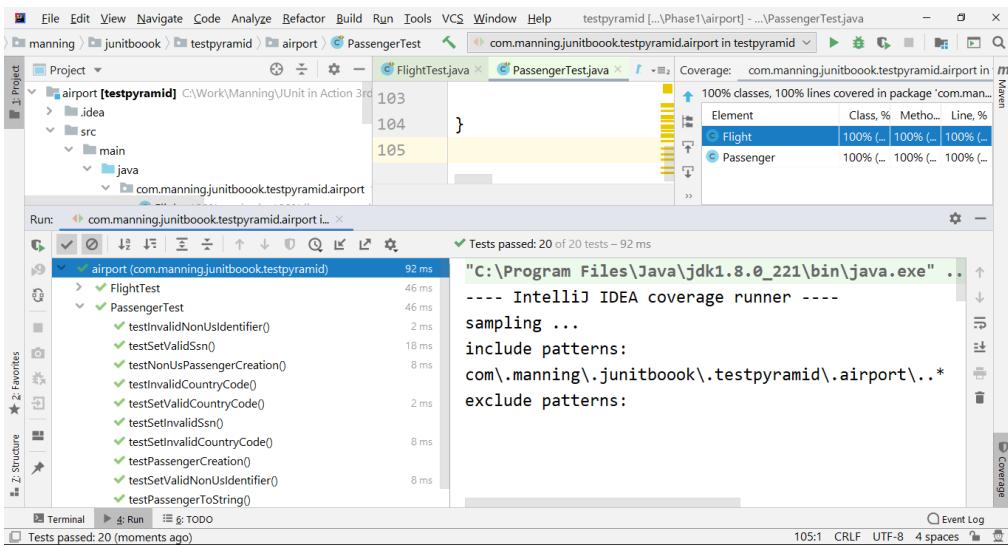


Figure 22.2 The unit tests for Passenger and Flight run successfully, the correct behavior of the individual classes is tested

Thomas has successfully created the application consisting of the `Passenger` and `Flight` classes and has checked that each class is working fine individually. Using JUnit 5 tests, he has checked: The passengers' identifiers and country codes restrictions

- The flights' numbers restrictions, that need to start with two letters and be followed by three or four digits
- The bad input values, e.g. a flight with a negative number of seats.
- The boundary conditions, e.g. we cannot add more passengers than available seats.

From here, Thomas will move further to integrate the functionality of the two classes.

22.3 Integration testing – units combined as a group

Integration testing is a level of software testing where individual units are combined, in order to check the interaction between them. The fact that units work fine in isolation does not necessarily mean that they also work fine together.

Thomas will try now to investigate how the two classes, `Passenger` and `Flight`, cooperate. They represent two different units and, in order to cooperate, they need to expose appropriate interfaces (APIs). However, interfaces can have defects that will prevent their interactions, as missing methods or methods not receiving the appropriate type of arguments.

When we analyze the current interfaces, we find that the passenger should be effectively added to the flight and removed from the flight. Now, we are only modifying the number of passengers. The current interface is missing an `addPassenger` method, but also a

`removePassenger` method. The `Flight` class should be more integrated with `Passenger`, and keep the set of the recorded passengers.

We show the changes to the `Flight` class in listing 22.5.

Listing 22.5 The Flight class

```
public class Flight {  
  
    Set<Passenger> passengers = new HashSet<>(); #A  
  
    [...]  
    public boolean addPassenger(Passenger passenger) { #B  
        if(passengers.size() >= seats) { #C  
            throw new RuntimeException( #C  
                "Cannot add more passengers than the capacity of the flight!"); #C  
        } #C  
        return passengers.add(passenger); #D  
    }  
  
    public boolean removePassenger(Passenger passenger) { #E  
        return passengers.remove(passenger); #E  
    }  
  
    [...]  
}
```

The class has experienced the following changes:

- A field `passengers` — to keep the set of passengers (#A).
- A modified `addPassenger` method (#B), that will check if there is enough place (#C) and then effectively add the passenger to the set of passengers (#D).
- A `removePassenger` method that will remove the passenger from the passengers set (#E).

To do the integration testing, Thomas decides to use Arquillian - a testing framework for Java that leverages JUnit to execute test cases against a Java container. We introduced Arquillian in chapter 9 and we will revisit it now, in the context of the integration testing.

Arquillian does not have a JUnit 5 extension at this time, but it is very popular and has been largely adopted in projects up to JUnit 4. It greatly eases the task of managing containers, deployments, and framework initializations.

Arquillian tests Java EE applications. Using it in our examples requires some basic knowledge of CDI (Contexts and Dependency Injection), a Java EE standard for the inversion of control design pattern.

ShrinkWrap is an external dependency used with Arquillian and a simple way to create archives in Java. Using the fluent ShrinkWrap API, developers at Tested Data Systems can assemble .jar, .war, and .ear files to be deployed directly by Arquillian during testing. Such files are archives that may contain all classes needed to run an application. ShrinkWrap helps define the deployments and the descriptors to be loaded to the Java container being tested against.

Thomas will use a list of 50 passengers on the flight, to be described by identifier, name, and country. The list is stored in a CSV file (listing 22.6).

Listing 22.6 The flights_information.csv file

```
123-45-6789; John Smith; US
900-45-6789; Jane Underwood; GB
123-45-6790; James Perkins; US
900-45-6790; Mary Calderon; GB
123-45-6791; Noah Graves; US
900-45-6791; Jake Chavez; GB
123-45-6792; Oliver Aguilar; US
900-45-6792; Emma Mccann; GB
123-45-6793; Margaret Knight; US
900-45-6793; Amelia Curry; GB
123-45-6794; Jack Vaughn; US
900-45-6794; Liam Lewis; GB
123-45-6795; Olivia Reyes; US
900-45-6795; Samantha Poole; GB
123-45-6796; Patricia Jordan; US
900-45-6796; Robert Sherman; GB
123-45-6797; Mason Burton; US
900-45-6797; Harry Christensen; GB
123-45-6798; Jennifer Mills; US
900-45-6798; Sophia Graham; GB
123-45-6799; Bethany King; US
900-45-6799; Isla Taylor; GB
123-45-6800; Jacob Tucker; US
900-45-6800; Michael Jenkins; GB
123-45-6801; Emily Johnson; US
900-45-6801; Elizabeth Berry; GB
123-45-6802; Isabella Carpenter; US
900-45-6802; William Fields; GB
123-45-6803; Charlie Lord; US
900-45-6803; Joanne Castaneda; GB
123-45-6804; Ava Daniel; US
900-45-6804; Linda Wise; GB
123-45-6805; Thomas French; US
900-45-6805; Joe Wyatt; GB
123-45-6806; David Byrne; US
900-45-6806; Megan Austin; GB
123-45-6807; Mia Ward; US
900-45-6807; Barbara Mac; GB
123-45-6808; George Burns; US
900-45-6808; Richard Moody; GB
123-45-6809; Victoria Montgomery; US
900-45-6809; Susan Todd; GB
123-45-6810; Joseph Parker; US
900-45-6810; Alexander Alexander; GB
123-45-6811; Jessica Pacheco; US
900-45-6811; William Schneider; GB
123-45-6812; Damian Reid; US
900-45-6812; Daniel Hart; GB
123-45-6813; Thomas Wright; US
900-45-6813; Charles Bradley; GB
```

Listing 22.7 implements the `FlightUtilBuilder` class, which parses the CSV file and populates the flight with the corresponding passengers. Thus, the code brings the information from an external file to the memory of the application.

Listing 22.7 The FlightBuilderUtil class

```
public class FlightBuilderUtil {

    public static Flight buildFlightFromCsv() throws IOException {
        Flight flight = new Flight("AA1234", 50);                      #A
        flight.setOrigin("London");                                     #A
        flight.setDestination("Bucharest");                                #A

        try(BufferedReader reader =
            new BufferedReader(new FileReader(
                "src/test/resources/flights_information.csv"))){          #B
            {
                String line = null;
                do {
                    line = reader.readLine();                                #C
                    if (line != null) {
                        String[] passengerString = line.toString().split(";"); #D
                        Passenger passenger =                                #E
                            new Passenger(passengerString[0].trim(),           #E
                                              passengerString[1].trim(),           #E
                                              passengerString[1].trim());         #E
                        flight.addPassenger(passenger);                         #F
                    }
                } while (line != null);
            }
            return flight;                                              #G
        }
    }
}
```

In this listing:

- We create a flight and set its origin and destination (#A).
- We open the CSV file to parse (#B).
- We read line by line (#C), split each line (#D), create a passenger based on the information that has been read (#E), and add the passenger to the flight (#F).
- We return the fully populated flight from the method (#G).

So far, all classes that have been implemented for the development of the flight and passenger management tasks are pure Java classes; no particular framework and technology have been used so far. As a testing framework that executes test cases against a Java container, Arquillian requires some understanding of notions related to Java EE and CDI. As it is a widespread framework for integration testing, we have decided to use it and we'll try to explain the most important ideas so that you can quickly adopt it within your projects.

Arquillian abstracts the container or application startup logic and it deploys the application to the targeted run time (an application server, embedded or managed) to execute test cases.

Listing 22.8 shows the dependencies that we need to add to the Maven pom.xml configuration file to work with Arquillian.

Listing 22.8 Required pom.xml dependencies

```
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.jboss.arquillian</groupId>                      #A
            <artifactId>arquillian-bom</artifactId>                      #A
            <version>1.4.0.Final</version>
            <scope>import</scope>
            <type>pom</type>
        </dependency>
    </dependencies>
</dependencyManagement>
<dependencies>
    <dependency>
        <groupId>org.jboss.spec</groupId>                                #B
        <artifactId>jboss-javae-7.0</artifactId>                          #B
        <version>1.0.3.Final</version>
        <type>pom</type>
        <scope>provided</scope>
    </dependency>
    <dependency>
        <groupId>org.junit.vintage</groupId>                               #C
        <artifactId>junit-vintage-engine</artifactId>                     #C
        <version>5.4.2</version>
        <scope>test</scope>
    </dependency>
    <dependency>
        <groupId>org.jboss.arquillian.junit</groupId>                      #D
        <artifactId>arquillian-junit-container</artifactId>                 #D
        <scope>test</scope>
    </dependency>
    <dependency>
        <groupId>org.jboss.arquillian.container</groupId>                  #E
        <artifactId>arquillian-weld-ee-embedded-1.1</artifactId>           #E
        <version>1.0.0.CR9</version>
        <scope>test</scope>
    </dependency>
    <dependency>
        <groupId>org.jboss.weld</groupId>                                    #F
        <artifactId>weld-core</artifactId>                                 #F
        <version>2.3.5.Final</version>
        <scope>test</scope>
    </dependency>
</dependencies>
```

This listing adds:

- The Arquillian API dependency (#A).
- The Java EE 7 API dependency (#B).
- The JUnit Vintage Engine dependency (#C). As mentioned earlier, at least for the moment, Arquillian is not yet integrated with JUnit 5. Because Arquillian lacks a JUnit 5 extension, we'll have to use the JUnit 4 dependencies and annotations to run our tests.

- The Arquillian JUnit integration dependency (#D).
- The container adapter dependencies (#E and #F). To execute our tests against a container, we must include the dependencies that correspond to that container. This requirement demonstrates one of the strengths of Arquillian: it abstracts the container from the unit tests and is not tightly coupled to specific tools that implement in-container unit testing.

An Arquillian test, as implemented in listing 22.9, looks just like a unit test, with some additions. The test is named `FlightWithPassengersTest` to show the goal of the integration testing between the two classes.

Listing 22.9 The FlightWithPassengersTest class

```
@RunWith(Arquillian.class) #A
public class FlightWithPassengersTest {

    @Deployment #B
    public static JavaArchive createDeployment() { #B
        return ShrinkWrap.create(JavaArchive.class) #B
            .addClasses(Passenger.class, Flight.class) #B
            .addAsManifestResource(EmptyAsset.INSTANCE, "beans.xml"); #B
    }

    @Inject #C
    Flight flight; #C

    @Test(expected = RuntimeException.class) #D
    public void testNumberOfSeatsCannotBeExceeded() throws IOException { #D
        assertEquals(20, flight.getNumberOfPassengers());
        flight.addPassenger(new Passenger("1247890", "Michael Johnson"));
    }

    @Test #E
    public void testAddRemovePassengers() throws IOException { #E
        flight.setSeats(21);
        Passenger additionalPassenger =
            new Passenger("1247890", "Michael Johnson");
        flight.addPassenger(additionalPassenger);
        assertEquals(21, flight.getNumberOfPassengers());
        flight.removePassenger(additionalPassenger);
        assertEquals(20, flight.getNumberOfPassengers());
        assertEquals(21, flight.getSeats());
    }
}
```

As this listing shows, an Arquillian test case must have three things:

- A `@RunWith(Arquillian.class)` annotation on the class (#A). The `@RunWith` annotation tells JUnit to use Arquillian as the test controller.
- A public static method annotated with `@Deployment` that returns a ShrinkWrap archive (#B). The purpose of the test archive is to isolate the classes and resources that the test needs. The archive is defined with ShrinkWrap. The micro deployment strategy lets us focus on precisely the classes we want to test. As a result, the test remains very

lean and manageable. For the moment, we have included only the Passenger and the Flight classes. We try to inject a Flight object as a class member, using the CDI @Inject annotation (#C). The @Inject annotation allows us to define injection points inside classes. In this case, @Inject instructs CDI to inject into the test a field of type reference to a Flight object.

- At least one method annotated with @Test (#D and #E). Arquillian looks for a public static method annotated with the @Deployment annotation to retrieve the test archive. Then each @Test annotated method is run inside the container environment.

When the ShrinkWrap archive is deployed to the server, it becomes a real archive. The container has no knowledge that the archive was packaged by ShrinkWrap.

We provided the infrastructure for using Arquillian in our project, now we'll run our integration tests with its help! If we run the tests now, we are going to get an error (figure 22.3).

```
java.lang.RuntimeException: WELD-001408: Unsatisfied dependencies for type Flight with qualifiers @Default
@Field [red] @Inject com.manning.junitbook.testpyramid.airport.FlightWithPassengersTest.flight
@mid.airport.FlightWithPassengersTest.flight(FightWithPassengersTest.java:0)
```

Figure 22.3 The result of running FlightWithPassengersTest – there is no default dependency for type Flight

The error says Unsatisfied dependencies for type Flight with qualifiers @Default. It means that the container is trying to inject the dependency, as it has been instructed through the CDI @Inject annotation, but it is unsatisfied. Why? What has Thomas missed? The Flight class provides only a constructor with arguments, and it has no default constructor to be used by the container for the creation of the object. The container does not know how to invoke the constructor with parameters and which parameters to pass to it to create the Flight object that must be injected.

What is the solution in this case? Java EE offers the producer methods that are designed to inject objects that require custom initialization (listing 22.10). The solution fixes the issue and is easy to put into practice, even by a junior developer.

Listing 22.10 The FlightProducer class

```
public class FlightProducer {  
  
    @Produces  
    public Flight createFlight() throws IOException {  
        return FlightBuilderUtil.buildFlightFromCsv();  
    }  
}
```

In this listing, we create the `FlightProducer` class with the `createFlight` method, which invokes `FlightBuilderUtil.buildFlightFromCsv()`. We can use this method to inject objects that require custom initialization, and in this case, we are injecting a flight that has been configured based on the CSV file. We annotated the `createFlight` method with `@Produces`, which is also a Java EE annotation. Then the container will automatically invoke this method to create the configured flight; then it injects the method into the `Flight` field, annotated with `@Inject` from the `FlightWithPassengersTest` class.

Listing 22.11 adds the `FlightProducer` class to the ShrinkWrap archive.

Listing 22.11 The modified deployment method from the FlightWithPassengersTest class

```
@Deployment
public static JavaArchive createDeployment() {
    return ShrinkWrap.create(JavaArchive.class)
        .addClasses(Passenger.class, Flight.class,
                    FlightProducer.class)
        .addAsManifestResource(EmptyAsset.INSTANCE, "beans.xml");
}
```

If we run the tests now, they will be green and the code coverage will be 100%. The container injected the correctly configured flight (figure 22.4).

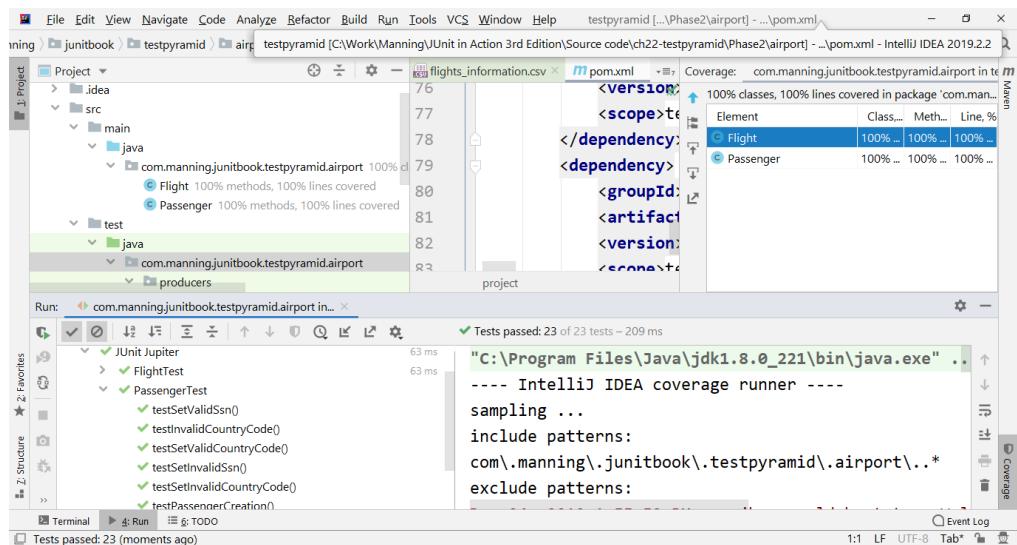


Figure 22.4 Successfully executing the integration tests from `FlightWithPassengersTest`. The code coverage is 100%, after fully configuring Arquillian

Thomas has successfully created the integration layer of the test pyramid and will move to the next level.

22.4 System testing – looking at the complete software

System testing tests the entire system, in order to evaluate the system compliance with the specification and to detect inconsistencies between units that are integrated together.

Mock objects (see chapter 8) can simulate the behavior of complex, real objects and are therefore useful when a real object (for example, some Depended-on component) is impractical to incorporate into a test, or it is impossible - at least for the moment - as the Depended-on component is not yet available (fig. 22.5). For example, our system may depend on some device providing outside conditions measurement (temperature, humidity). The results offered by this device influence our tests and are non-deterministic – we simply cannot decide the meteorological conditions that we need at a given time.

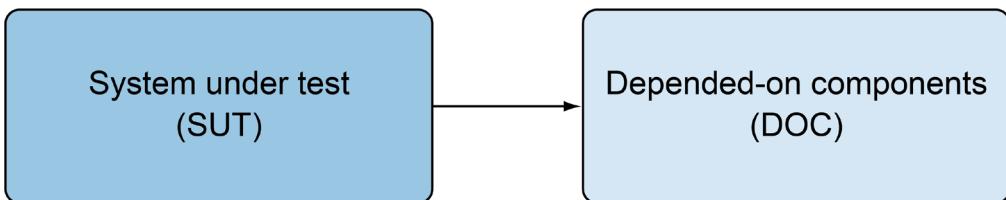


Figure 22.5 The System Under Test (SUT) depends on some component (DOC) that is not available from the beginning of the system development

When developing a program, we may need to create mock objects that simulate the behavior of complex, real objects in order to achieve our testing goals. Common uses of mock objects include communicating with an external or internal service that is not yet available. These kinds of services may not be fully available, or they might be maintained by different teams which makes accessing them slow or difficult. That's why a test double is handy, it stops our own tests from waiting for the availability of that service. We would like to make these mock objects an accurate representation of the external service. It is important that the Depended-On Component keeps its contract – the expected behavior and an API that our system may use.

When the Depended-On Component is created in parallel by some different team, it is useful to use a consumer-driven contract approach. This means that the provider must follow some API or some behavior, as the consumer is expecting.

22.4.1 Testing with a mock external dependency

Thomas has left the flights management application at the level of integration testing. There is a new feature to be introduced that will require awarding some bonus points to the passengers, depending on the distances they have traveled with the company. The bonus policy is simple: a passenger will get 1 bonus point for every 10 kilometers that he or she has traveled.

The management of the bonus policy is externalized to some other team, that will provide the `DistancesManager` class. So far, Thomas knows that the interface forms the contract between the consumer and the provider. We are following a consumer-driven contract.

We know that the API of the `DistancesManager` provides the following methods:

- `getPassengersDistancesMap` provides a map that has the passenger as key and the traveled distance as value
- `getPassengersPointsMap` provides a map that has the passenger as key and the bonus points as value
- `addDistance` adds a traveled distance to the passenger
- `calculateGivenPoints` calculates the bonus points for a passenger

We do not know the implementations so far, so we'll provide dummy ones (listing 22.12) and we are going to mock the behavior of the class.

Listing 22.12 The dummy implementation of the `DistancesManager` class

```
public class DistancesManager {  
  
    public Map<Passenger, Integer> getPassengersDistancesMap() {  
        return null;  
    }  
  
    public Map<Passenger, Integer> getPassengersPointsMap() {  
        return null;  
    }  
  
    public void addDistance(Passenger passenger, int distance) {}  
  
    public void calculateGivenPoints() {}  
  
}
```

We have provided one flight description within one CSV file, but this is not enough. We need passengers that have participated in more than one flight, to make sure that our calculation is consistent. So, we will introduce two more CSV files describing two other flights, which have passengers in common with the first flight (listings 22.13 and 22.14 show some partial lists).

Listing 22.13 The `flights_information2.csv` file

```
123-45-6789; John Smith; US  
900-45-6789; Jane Underwood; GB  
123-45-6790; James Perkins; US  
[...]
```

Listing 22.14 The `flights_information3.csv` file

```
123-45-6790; James Perkins; US  
900-45-6790; Mary Calderon; GB  
123-45-6792; Oliver Aguilar; US  
[...]
```

To know that we have the same passenger on different flights, we have to override the `equals` and `hashCode` methods into the `Passenger` class (listing 22.15). We'll know that two passengers are the same if they have the same identifier.

Listing 22.15 The overridden equals and hashCode methods into the Passenger class

```
public class Passenger {  
    [...]  
    @Override  
    public boolean equals(Object o) {  
        if (this == o) return true;  
        if (o == null || getClass() != o.getClass()) return false;  
        Passenger passenger = (Passenger) o;  
        return Objects.equals(identifier, passenger.identifier);  
    }  
  
    @Override  
    public int hashCode() {  
        return Objects.hash(identifier);  
    }  
}
```

In order to distinguish between different flights, Thomas will introduce the `FlightNumber` annotation, which will receive the number as a parameter (listing 22.16).

Listing 22.16 The FlightNumber annotation

```
@Qualifier #A  
@Retention(RUNTIME) #B  
@Target({FIELD, METHOD}) #C  
public @interface FlightNumber {  
    String number(); #A  
}
```

In this listing:

- We create the `FlightNumber` annotation (#A), the information it has provided on the annotated elements will be kept during runtime (#B).
- This annotation may be applied to fields and methods (#C) and it has a number parameter (#D).

In the `FlightProducer` class, we'll annotate the `createFlight` method with the new `FlightNumber` annotation, having "AA1234" as an argument, in order to give an identity to the flight (listing 22.17).

Listing 22.17 The modified FlightProducer class

```
public class FlightProducer {  
  
    @Produces  
    @FlightNumber(number= "AA1234")  
    public Flight createFlight() throws IOException {
```

```

        return FlightBuilderUtil.buildFlightFromCsv("AA1234",
                50,"src/test/resources/flights_information.csv");
    }
}

```

Thomas will also change the `FlightWithPassengersTest` class, to annotate the injected flight and to write a test for the distances manager (listing 22.18).

Listing 22.18 The modified `FlightWithPassengersTest` class

```

@RunWith(Arquillian.class)
public class FlightWithPassengersTest {
    [...]

    @Inject
    @FlightNumber(number= "AA1234")                                     #A
    Flight flight;

    @Mock
    DistancesManager distancesManager;                                  #B
    #B

    @Rule
    public MockitoRule mockitoRule = MockitoJUnit.rule();               #C
    #C

    private static Map<Passenger, Integer> passengersPointsMap =      #D
        new HashMap<>();                                              #D

    @BeforeClass
    public static void setUp() {                                         #E
        passengersPointsMap.put(new Passenger("900-45-6809",           #E
                "Susan Todd", "GB"), 210);                                #E
        passengersPointsMap.put(new Passenger("900-45-6797",           #E
                "Harry Christensen", "GB"), 420);                            #E
        passengersPointsMap.put(new Passenger("123-45-6799",           #E
                "Bethany King", "US"), 630);                                #E
    }
    [...]

    @Test
    public void testFlightsDistances() {                                    #F
        when(distancesManager.getPassengersPointsMap()).            #G
            thenReturn(passengersPointsMap);                           #G

        assertEquals(210, distancesManager.getPassengersPointsMap().    #H
                get(new Passenger("900-45-6809", "Susan Todd", "GB")).longValue()); #H
        assertEquals(420, distancesManager.getPassengersPointsMap().    #H
                get(new Passenger("900-45-6797", "Harry Christensen", "GB")).longValue()); #H
        assertEquals(630, distancesManager.getPassengersPointsMap().    #H
                get(new Passenger("123-45-6799", "Bethany King", "US")).longValue()); #H
    }
}

```

In the listing above, we are doing the following:

- We annotate the `flight` field with the `FlightNumber` annotation (#A), to give an

identity to the injected flight.

- We provide a mock implementation of the `DistancesManager` object and annotate this one with `@Mock (#B)`.
- We declare a `MockitoRule @Rule` annotated object, as it is needed to allow the initialization of mocks annotated with `@Mock (#C)`. Remember that we are using Arquillian which is not compatible with JUnit 5, and we need to use the rules from JUnit 4.
- We declare a `passengersPointsMap` to keep the bonus points for the passengers (#D) and we are populating it with some expected values (#E).
- We create the `testFlightsDistances (#F)`, where we instruct the mock object to return `passengersPointsMap` when we call `distancesManager.getPassengersPointsMap () (#G)`. Then, we check the bonus points are the expected ones (#H). For now, we have only inserted some data into the map and we check that we are correctly retrieving it. However, we have defined a tests skeleton and we are expecting new functionalities from the provider side!

22.4.2 Testing with a partially implemented external dependency

From the provider side, Thomas will receive the partial implementation of the `DistancesManager` class (listing 22.19).

Listing 22.19 The modified implementation of the `DistancesManager` class

```
public class DistancesManager {  
  
    private static final int DISTANCE_FACTOR = 10; #A  
  
    private Map<Passenger, Integer> passengersDistancesMap = new HashMap<>(); #B  
    private Map<Passenger, Integer> passengersPointsMap = new HashMap<>(); #B  
  
    public Map<Passenger, Integer> getPassengersDistancesMap() { #B  
        return Collections.unmodifiableMap(passengersDistancesMap); #B  
    }  
  
    public Map<Passenger, Integer> getPassengersPointsMap() { #B  
        return Collections.unmodifiableMap(passengersPointsMap); #B  
    }  
  
    public void addDistance(Passenger passenger, int distance) { #C  
    }  
  
    public void calculateGivenPoints() { #D  
        for (Passenger passenger : getPassengersDistancesMap().keySet()) { #D  
            passengersPointsMap.put(passenger, #D  
                getPassengersDistancesMap().get(passenger)/ DISTANCE_FACTOR); #D  
        }  
    }  
}
```

In the listing above, we are doing the following:

- We define the `DISTANCE_FACTOR` constant to divide the distances through, in order to get the number of bonus points (#A).
- We keep a `passengersDistancesMap` and a `passengersPointsMap` and provide getters for them (#B).
- The `addDistance` method is still not implemented (#C), while the `calculateGivenPoints` method has an implementation that browses the `passengersDistancesMap` and divides the distances by the `DISTANCE_FACTOR` to populate the `passengersPointsMap` (#D).

In the real applications, you may receive some fully implemented packages or classes, while another part is still under construction, but it follows the agreed contract. To simplify, we reduced our situation to a simple class with four methods, from which only one does not have an implementation. What is important is that the API contract is respected.

How will this change our tests on the consumer side? We know how to get the bonus based on the distance, so we won't keep the `passengersPointsMap`, but a `passengersDistancesMap`.

The changes to `FlightWithPassengersTest` will look as in listing 22.20.

Listing 22.20 The modified FlightWithPassengersTest class

```
@RunWith(Arquillian.class)
public class FlightWithPassengersTest {
    [...]

    @Spy
    DistancesManager distancesManager;                                #A
    #A

    private static Map<Passenger, Integer> passengersDistancesMap =   #B
        new HashMap<>();                                         #B

    @BeforeClass
    public static void setUp() {                                       #C
        #C
        passengersDistancesMap.put(new Passenger("900-45-6809",           #C
            "Susan Todd", "GB"), 2100);                                #C
        passengersDistancesMap.put(new Passenger("900-45-6797",           #C
            "Harry Christensen", "GB"), 4200);                            #C
        passengersDistancesMap.put(new Passenger("123-45-6799",           #C
            "Bethany King", "US"), 6300);                                #C
    }

    [...]

    @Test
    public void testFlightsDistances() {
        when(distancesManager.getPassengersDistancesMap()).           #D
            thenReturn(passengersDistancesMap);                         #D

        distancesManager.calculateGivenPoints();                        #E

        assertEquals(210, distancesManager.getPassengersPointsMap().      #F
            get(new Passenger("900-45-6809", "Susan Todd", "GB")).longValue()); #F
    }
}
```

```

        assertEquals(420, distancesManager.getPassengersPointsMap().
            get(new Passenger("900-45-6797", "Harry Christensen", "GB")) #F
            .longValue()); #F
        assertEquals(630, distancesManager.getPassengersPointsMap()
            .get(new Passenger("123-45-6799", "Bethany King", "US")) #F
            .longValue()); #F
    }
}

```

In the listing above, we are doing the following:

- We change the annotation of the `DistancesManager` object to `@Spy (#A)`. With the previous `@Mock` annotation, we are mocking the whole `distancesManager` object. In order to tell that we would like to mock only some methods and to keep the functionality of other ones, we replace the `@Mock` annotation with the `@Spy` one.
- We initialize the `passengersDistancesMap (#B)` and we populate it before the execution of the tests (#C).
- Into the `testFlightsDistances` we instruct the `distancesManager` object to return `passengersDistancesMap` when we call `distancesManager.getPassengersDistancesMap () (#D)`. Then, we call the already implemented `calculateGivenPoints (#E)` and check the bonus points are the expected ones after making this calculus (#F).

If we run the tests now, they will be green. The code coverage is still not 100%, as we are still waiting for the implementation of a method from `DistancesManager`, in order to be able to test everything (fig. 22.6).

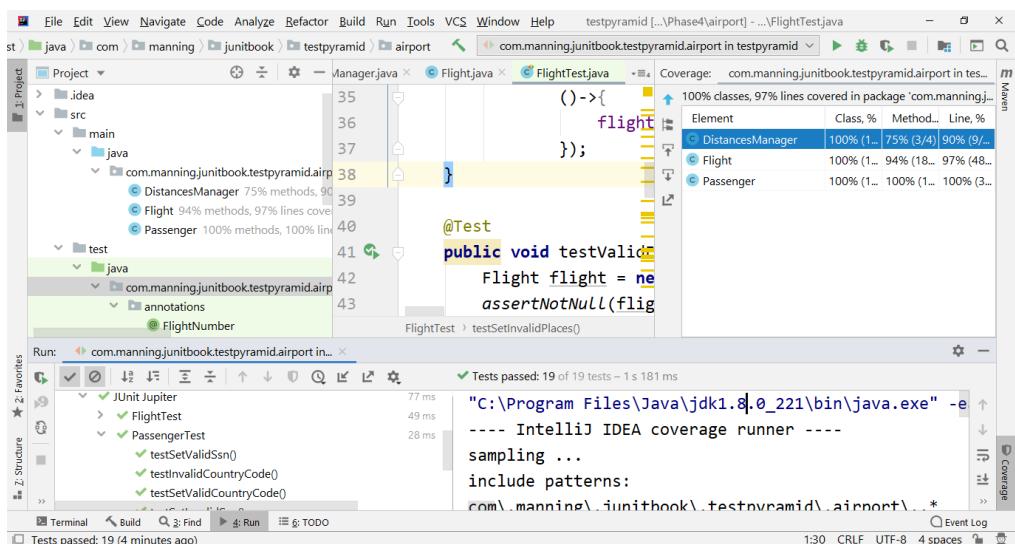


Figure 22.6 The `PassengerTest`, `FlightTest`, and `FlightWithPassengersTest` run successfully with a partial implementation of the `DistancesManager` class. The code coverage is not yet 100%, as some functionality is

still missing

22.4.3 Testing with the fully implemented external dependency

On the consumer side, the real system is now fully available. Thomas will have to test the functionality against this real provider service. The full implementation of the DistancesManager class has introduced the addDistance method (listing 22.21).

Listing 22.21 The modified implementation of the DistancesManager class

```
public class DistancesManager {  
    [...]  
  
    public void addDistance(Passenger passenger, int distance) {  
        if (passengersDistancesMap.containsKey(passenger)) { #A  
            passengersDistancesMap.put(passenger,  
                passengersDistancesMap.get(passenger) + distance); #B  
        } else {  
            passengersDistancesMap.put(passenger, distance); #C  
        }  
    }  
}
```

In the listing above, we are doing the following:

- In the addDistance method, we check if the passengersDistanceMap already contains a passenger (#A).
- If that passenger already exists, we add the distance to him (#B), otherwise we create a new entry into the map having that passenger as a key and the distance as the initial value (#C).

As we are going to use the real passengers set for each flight and populate the passengersDistancesMap based on their information, we'll introduce the getPassengers method into the Flight class (listing 22.22).

Listing 22.22 The modified implementation of the Flight class

```
public class Flight {  
    [...]  
    private Set<Passenger> passengers = new HashSet<Passenger>(); #A  
  
    public Set<Passenger> getPassengers() { #B  
        return Collections.unmodifiableSet(passengers); #B  
    }  
    [...]  
}
```

In the listing above, we are doing the following:

- We make passengers private (#A) and we expose it through a getter (#B).

Thomas will move back to the test that was previously mocking a behavior and introduce real behavior. He will remove the Mockito dependencies from the pom.xml file, as they are no longer needed. He will also remove the initialization of the passengersDistancesMap before executing the tests. The changes to the FlightWithPassengersTest are shown in listing 22.23.

Listing 22.23 The modified FlightWithPassengersTest class

```
@RunWith(Arquillian.class)
public class FlightWithPassengersTest {
    @Deployment
    public static JavaArchive createDeployment() {
        return ShrinkWrap.create(JavaArchive.class)
            .addClasses(Passenger.class, Flight.class, FlightProducer.class,
                        DistancesManager.class)                                     #A
            .addAsManifestResource(EmptyAsset.INSTANCE, "beans.xml");
    }

    @Inject
    @FlightNumber(number= "AA1234")                                     #B
    Flight flight;                                                       #B
    Flight flight2;                                                      #B

    @Inject
    @FlightNumber(number= "AA1235")                                     #B
    Flight flight3;                                                     #B

    @Inject
    @FlightNumber(number= "AA1236")                                     #B
    Flight flight4;                                                    #B

    @Inject
    DistancesManager distancesManager;                                  #C
    [...]

    @Test
    public void testFlightsDistances() {

        for (Passenger passenger : flight.getPassengers()) {           #D
            distancesManager.addDistance(passenger, flight.getDistance()); #D
        }

        for (Passenger passenger : flight2.getPassengers()) {          #D
            distancesManager.addDistance(passenger, flight2.getDistance()); #D
        }

        for (Passenger passenger : flight3.getPassengers()) {          #D
            distancesManager.addDistance(passenger, flight3.getDistance()); #D
        }

        distancesManager.calculateGivenPoints();                         #E

        assertEquals(210, distancesManager.getPassengersPointsMap()
            .get(new Passenger("900-45-6809", "Susan Todd", "GB"))
            .longValue());                                              #F
        assertEquals(420, distancesManager.getPassengersPointsMap()
            .get(new Passenger("900-45-6797", "Harry Christensen", "GB"))
            .longValue());                                             #F
    }
}
```

```

        assertEquals(630, distancesManager.getPassengersPointsMap()
            .get(new Passenger("123-45-6799", "Bethany King", "US"))
            .longValue());
    }
}

```

In the listing above, we are doing the following:

- We add the `DistancesManager.class` to the ShrikWrap archive (#A) so that the `DistancesManager` class will be able to be injected into the test.
- We inject three flights into the test, annotate and differentiate them with the `@FlightNumber` annotation, having different arguments (#B). We also inject a `DistancesManager` field (#C).
- We change the `testFlightsDistance` test to browse all passengers from the three flights and add the distances that they have traveled to the `distancesManager` (#D). Based on the traveled distances we calculate the bonus given points (#E) and check the bonus points are the expected ones after making this calculus (#F).

If we run the tests now, they will be green and the code coverage will be 100% (fig. 22.7).

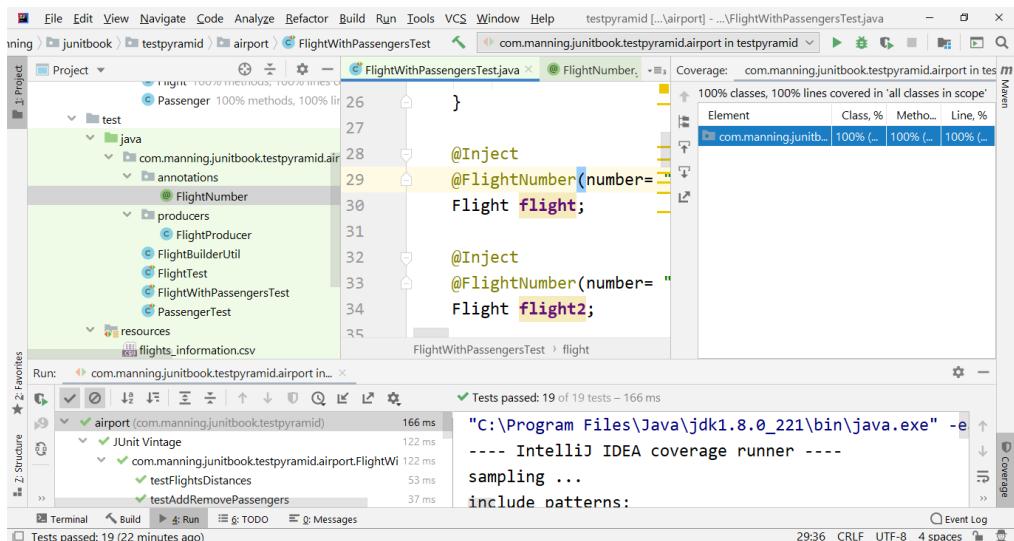


Figure 22.7 The `PassengerTest`, `FlightTest`, and `FlightWithPassengersTest` run successfully with a full implementation of the `DistancesManager` class and the code coverage is 100%

Thomas has successfully tested the whole system using a consumer-driven contract and moving from a mock implementation of the external functionality to using the real one. He is ready to move now to the last step in building the tests pyramid strategy: acceptance testing.

22.5 Acceptance testing – compliance with the business requirements

Acceptance testing is the level of software testing where a system is tested for compliance with the business requirements. Once the system testing has been fulfilled, acceptance testing is executed to confirm that the software is ready for delivery and to satisfy the needs of the end-users.

In chapter 21 we discussed the working features that give business value to the software, and the communication challenges within a project between the customer, the business analyst, the developer, and the tester.

We have emphasized that acceptance criteria may be expressed as scenarios, in a way that can be automated later. The keywords are "Given", "When", "Then". More exactly:

Given <a context>

When <an action occurs>

Then <expect a result>

Thomas needs to implement a new feature in the application. This feature concerns the company policy regarding adding and removing passengers to flights. Because the number of seats and the type of passenger must be considered, the company defines the following policy: in case of constraints, regular passengers may be removed from a flight and added to another one, while VIP passengers cannot be removed from a flight.

This is the business logic that the application must be compliant with in order to satisfy the end-user. For fulfilling the acceptance testing, Thomas will use Cucumber, the acceptance testing framework that we used in chapter 21. Cucumber describes the application scenarios in plain English text, using Gherkin as language. Cucumber is easy to read and understand by stakeholders and allows automation.

In order to start working with Cucumber, Thomas will first introduce the additional dependencies that are needed at the level of the Maven pom.xml. These ones are cucumber-java and cucumber-junit (listing 22.24).

Listing 22.24 The Cucumber dependencies into the Maven pom.xml file

```
<dependency>
    <groupId>info.cukes</groupId>
    <artifactId>cucumber-java</artifactId>
    <version>1.2.5</version>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>info.cukes</groupId>
    <artifactId>cucumber-junit</artifactId>
    <version>1.2.5</version>
    <scope>test</scope>
</dependency>
```

Thomas will effectively introduce the new feature by working TDD/BDD style, as we have demonstrated in chapters 20 and 21. This means that he will first write the acceptance tests in Cucumber. Then, he will write the tests in Java. Then, he will write the code that will fix the tests and consequently provide the feature implementation.

Thomas will introduce the 2 types of passengers: regular ones and VIPs. This will change the Passenger class by adding a boolean `vip` field, together with a getter and a setter (listing 22.25).

Listing 22.25 The modified Passenger class

```
public class Passenger {  
    [...]  
    private boolean vip;  
  
    public boolean isVip() {  
        return vip;  
    }  
  
    public void setVip(boolean vip) {  
        this.vip = vip;  
    }  
    [...]  
}
```

Then, to build the scenarios that will express the acceptance criteria, Thomas will create the `passengers_policy.feature` file into the new `test/resources/features` folder. He reuses the 3 CSV flight files that have been used for the system integration testing. The scenarios that Thomas defines are presented into listing 22.26 and can be read in natural language. You may review the capabilities of Cucumber by revisiting chapter 21.

Listing 22.26 The `passengers_policy.feature` file

Feature: Passengers Policy

The company follows a policy of adding and removing passengers, depending on the passenger type

Scenario Outline: Flight with regular passengers

Given there is a flight having number "<flightNumber>" and <seats> seats with passengers defined into "<file>"

When we have regular passengers

Then you can remove them from the flight

And add them to another flight

Examples:

flightNumber	seats	file
AA1234	50	flights_information.csv
AA1235	50	flights_information2.csv
AA1236	50	flights_information3.csv

Scenario Outline: Flight with VIP passengers

Given there is a flight having number "<flightNumber>" and <seats> seats with passengers defined into "<file>"

When we have VIP passengers

Then you cannot remove them from the flight

Examples:

flightNumber	seats	file
AA1234	50	flights_information.csv
AA1235	50	flights_information2.csv
AA1236	50	flights_information3.csv

In order to generate the skeleton of the Java tests, Thomas will make sure that the Cucumber and Gherkin plugins are installed, by accessing the File -> Settings -> Plugins menu (fig. 22.8 and 22.9).

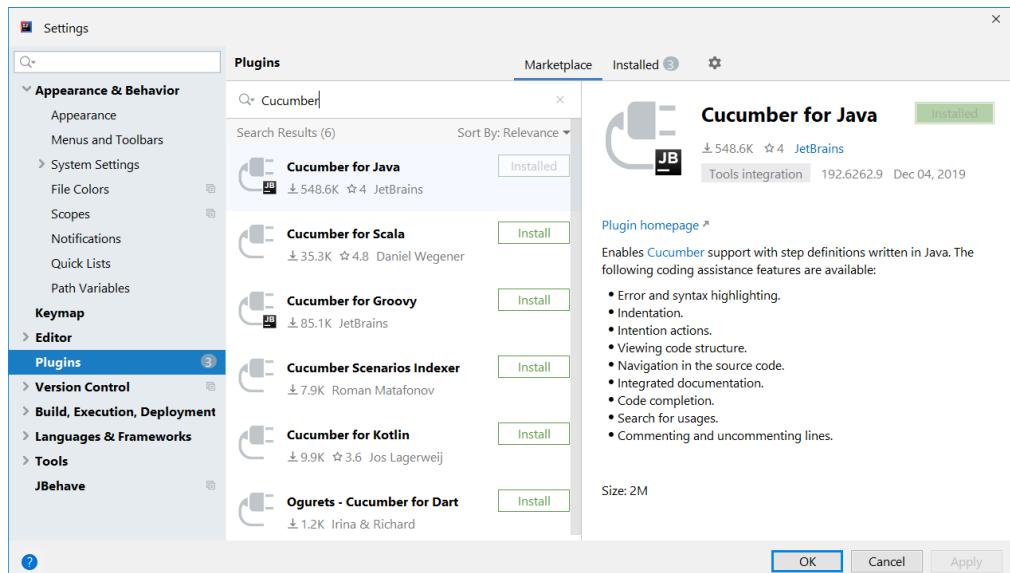


Figure 22.8 Installing the Cucumber for Java plugin from the File -> Settings -> Plugins menu

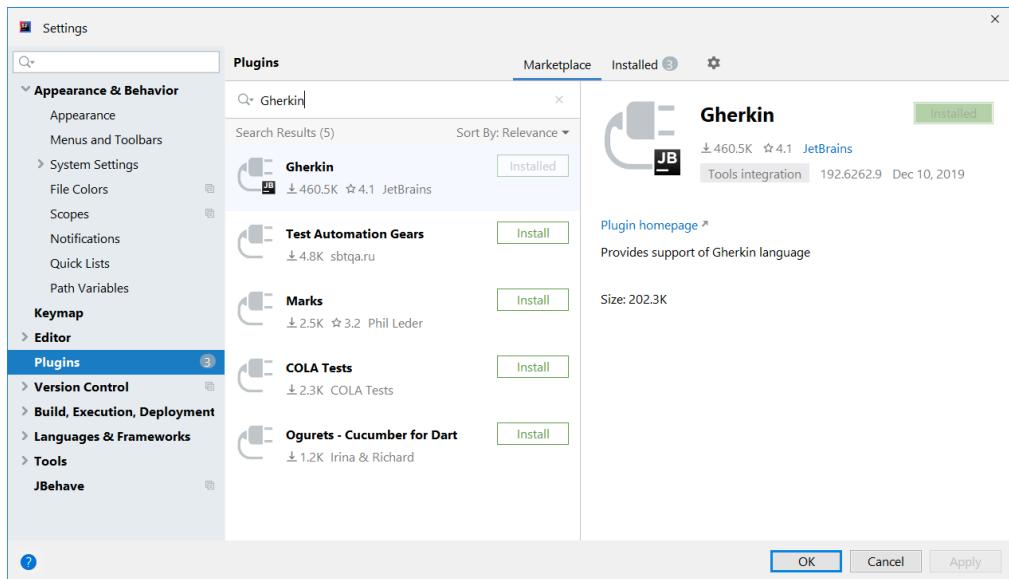


Figure 22.9 Installing the Gherkin plugin from the File -> Settings -> Plugins menu

Then, he must configure the way the feature is run by going to Run -> Edit Configurations, and make a few settings (fig. 22.10).

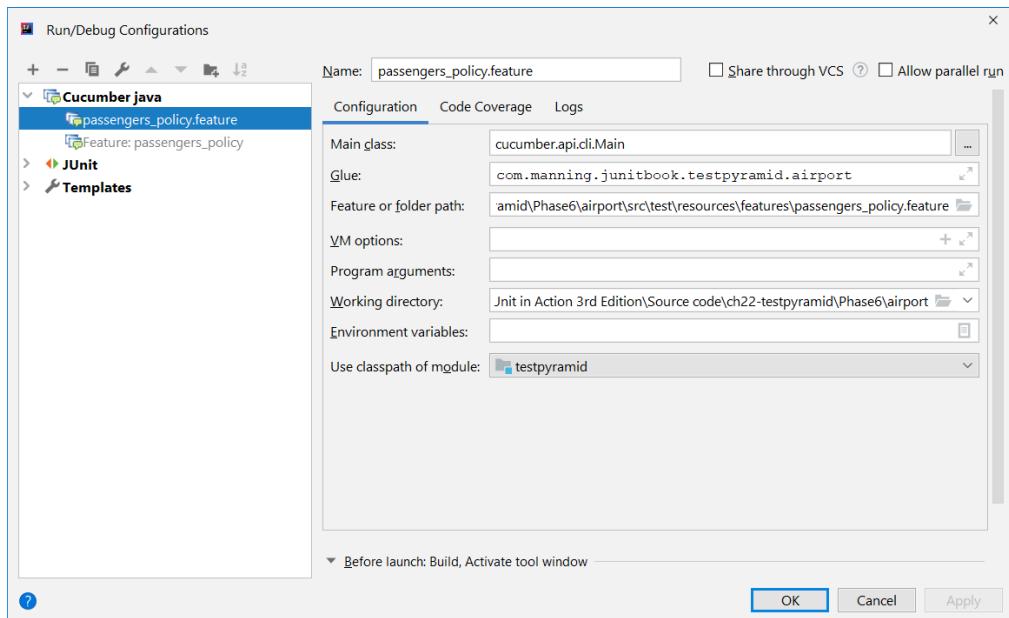


Figure 22.10 Setting the feature configuration by filling in the Main class, Glue, Feature or folder path, and Working directory

- Main class: cucumber.api.cli.Main
- Glue (the package where step definitions are stored): com.manning.junitbook.testpyramid.airport
- Feature or folder path: the newly created file test/resources/features/passengers_policy.feature
- Working directory: the project folder

After making this configuration, Thomas will be able to directly run the feature file by right-clicking on it (fig 22.11) and he will get the skeleton of the Java tests (fig. 22.12).

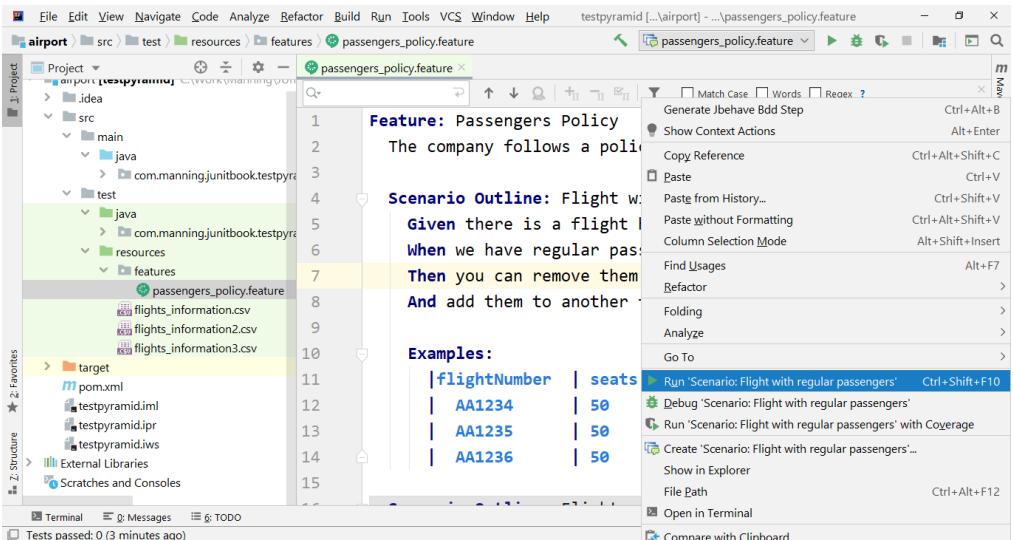


Figure 22.11 Directly running the passengers_policy.feature file by right-clicking on the feature file

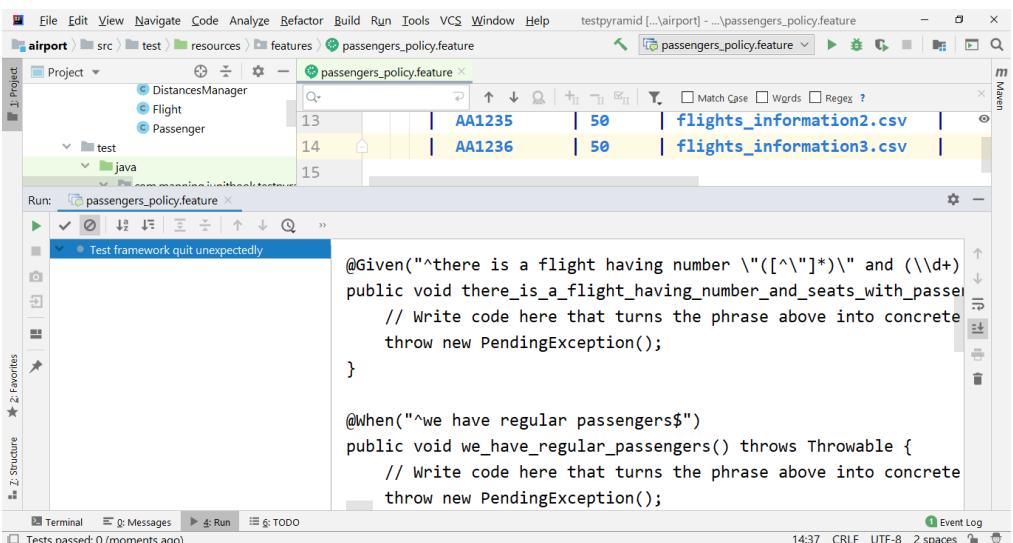


Figure 22.12 Generating the tests skeleton of the passengers policy by directly running the feature file

Thomas will create the PassengersPolicy file, that will contain the skeleton of the tests to be executed (listing 22.27).

Listing 22.27 The skeleton of the PassengersPolicy test

```
public class PassengersPolicy {  
    @Given("^there is a flight having number \"([^\"]*)\" and (\\d+) seats  
          with passengers defined into \"([^\"]*)\"$")  
    public void there_is_a_flight_having_number_and_seats_with_passengers_defined_into(  
        String arg1, int arg2, String arg3) throws Throwable {  
        // Write code here that turns the phrase above into concrete actions  
        throw new PendingException();  
    }  
  
    @When("^we have regular passengers$")  
    public void we_have_regular_passengers() throws Throwable {  
        // Write code here that turns the phrase above into concrete actions  
        throw new PendingException();  
    }  
  
    @Then("^you can remove them from the flight$")  
    public void you_can_remove_them_from_the_flight() throws Throwable {  
        // Write code here that turns the phrase above into concrete actions  
        throw new PendingException();  
    }  
  
    @Then("^add them to another flight$")  
    public void add_them_to_another_flight() throws Throwable {  
        // Write code here that turns the phrase above into concrete actions  
        throw new PendingException();  
    }  
  
    @When("^we have VIP passengers$")  
    public void we_have_VIP_passengers() throws Throwable {  
        // Write code here that turns the phrase above into concrete actions  
        throw new PendingException();  
    }  
  
    @Then("^you cannot remove them from the flight$")  
    public void you_cannot_remove_them_from_the_flight() throws Throwable {  
        // Write code here that turns the phrase above into concrete actions  
        throw new PendingException();  
    }  
}
```

In order to run the Cucumber tests, Thomas will need a special class. The name of the class could be anything, he has chosen `CucumberTest` (listing 22.28).

Listing 22.28 The CucumberTest class

```
@RunWith(Cucumber.class) #A  
@CucumberOptions(#B  
    plugin = {"pretty"},  
    features = "classpath:features") #C  
public class CucumberTest {  
  
    /** #D  
     * This class should be empty, step definitions should be in separate classes.  
     */
```

```
}
```

In the listing above, we are doing the following:

- We have annotated this class with `@RunWith(Cucumber.class)` annotation (#A). Executing it as any JUnit test class will run all features found on the classpath in the same package. As there is no Cucumber JUnit 5 extension at the moment of writing this chapter, we use the JUnit 4 runner.
- The `@CucumberOptions` (#B) annotation provides the plugin option (#C), that is used to specify different formatting options for the output reports. Using "pretty", the Gherkin source will be printed with additional colors. And the `features` option (#D) helps Cucumber to locate the feature file in the project folder structure. It will look for the features folder on the classpath - and remember that the `src/test/resources` folder is maintained by Maven on the classpath!

Thomas will now turn back to the `PassengersPolicy` class and write the tests to check the functionality of the passengers policy feature to be introduced (listing 22.29).

Listing 22.29 The PassengersPolicy test class

```
public class PassengersPolicy {  
    private Flight flight; #A  
    private List<Passenger> regularPassengers = new ArrayList<>(); #B  
    private List<Passenger> vipPassengers = new ArrayList<>(); #B  
    private Flight anotherFlight = new Flight("AA7890", 48); #C  
  
    @Given("^there is a flight having number \"([^\"]*)\" and (\\\d+)  
          seats with passengers defined into \"([^\"]*)\"$")  
    public void there_is_a_flight_having_number_and_seats_with_passengers_defined_into(  
        String flightNumber, int seats, String fileName) throws Throwable {  
        flight = FlightBuilderUtil.buildFlightFromCsv(flightNumber, #D  
            seats, "src/test/resources/" + fileName); #D  
    }  
  
    @When("^we have regular passengers$")  
    public void we_have_regular_passengers() {  
        for (Passenger passenger: flight.getPassengers()) { #E  
            if (!passenger.isVip()) { #E  
                regularPassengers.add(passenger); #E  
            }  
        }  
    }  
  
    @Then("^you can remove them from the flight$")  
    public void you_can_remove_them_from_the_flight() {  
        for(Passenger passenger: regularPassengers) { #F  
            assertTrue(flight.removePassenger(passenger)); #F  
        }  
    }  
  
    @Then("^add them to another flight$")  
    public void add_them_to_another_flight() {  
        for(Passenger passenger: regularPassengers) { #G
```

```

        assertTrue(anotherFlight.addPassenger(passenger)); #G
    }

}

@When("^we have VIP passengers$")
public void we_have_VIP_passengers() {
    for (Passenger passenger: flight.getPassengers()) { #H
        if (passenger.isVip()) { #H
            vipPassengers.add(passenger); #H
        }
    }
}

@Then("^you cannot remove them from the flight$")
public void you_cannot_remove_them_from_the_flight(){ #I
    for(Passenger passenger: vipPassengers) { #I
        assertFalse(flight.removePassenger(passenger)); #I
    }
}
}

```

In the listing above, we are doing the following:

- We define as fields the flight that the passengers are moved from (#A), the list of regular and VIP passengers (#B) and the second flight that the passengers are moved to (#C).
- The step labeled as "Given there is a flight having number "<flightNumber>" and <seats> seats with passengers defined into "<file>"" will initialize the flight from the CSV file (#D).
- We browse the list of all passengers and add the regular ones to their list (#E), check that we can remove them from a flight (#F) and add them to another flight (#G).
- We browse the list of all passengers and add the VIP ones to their list (#H) and check that we cannot remove them from a flight (#I).

Running the tests now by executing `CucumberTest`, Thomas will obtain the result shown in figure 22.13. Only the tests concerning VIP passengers fail, so Thomas will know that he has to change the code only regarding this type of passenger.

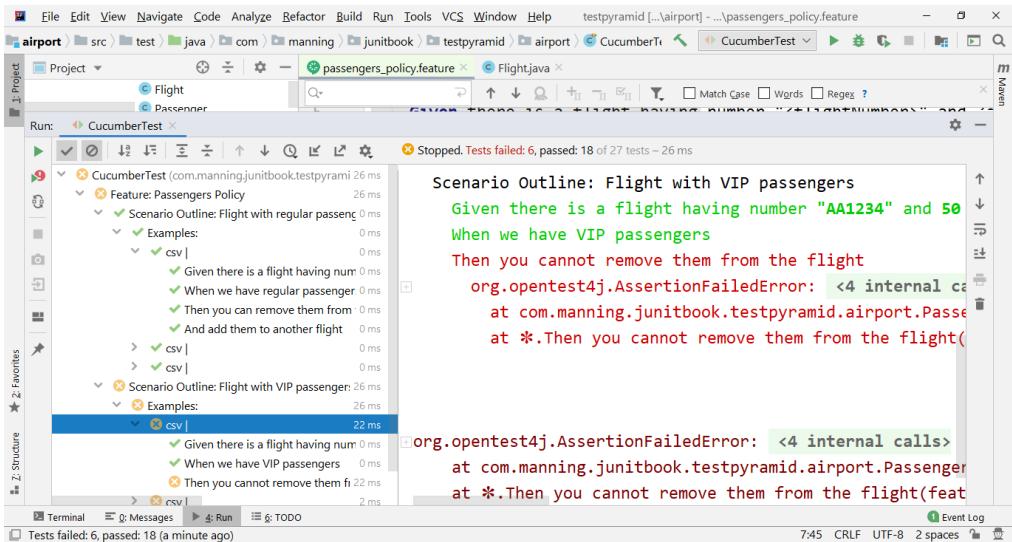


Figure 22.13 Running the newly introduced PassengersPolicy tests will result in failure only for the VIP passengers

Thomas will change the `addPassenger` method from the `Flight` class, as shown in listing 22.30 – he will introduce the condition that a VIP passenger cannot be removed from a flight (#A).

Listing 22.30 The modified Flight class

```
public class Flight {
    [...]
    public boolean removePassenger(Passenger passenger) {
        if(passenger.isVip()) {
            return false;
        }
        return passengers.remove(passenger);
    }
}
```

Running the tests now by executing `CucumberTest`, Thomas will obtain the result shown in figure 22.14 – all tests will be successfully executed.

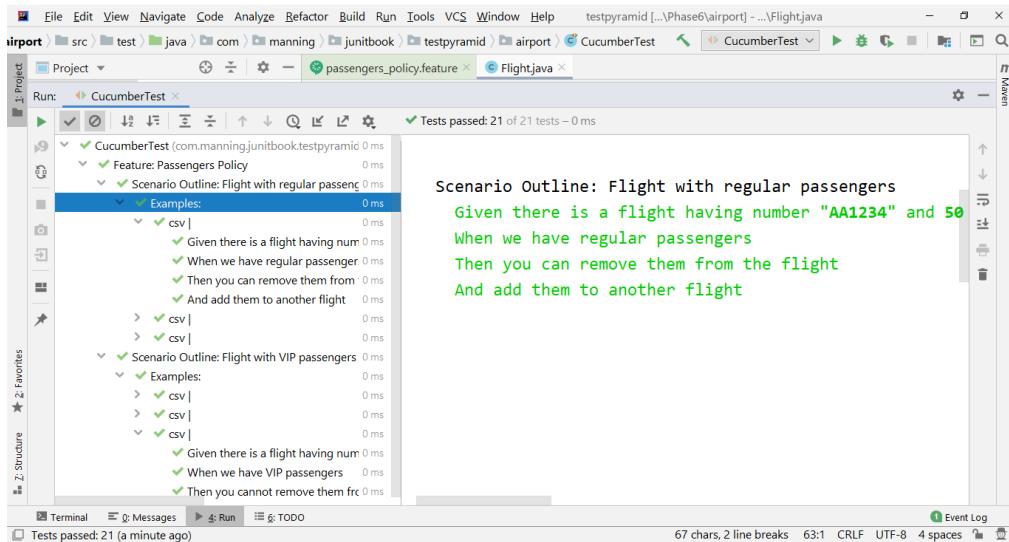


Figure 22.14 Successfully running the newly introduced PassengersPolicy tests after writing the business logic

In order to check the code coverage, Thomas will execute all tests that form the test pyramid. The code coverage is 100% (fig. 22.15).

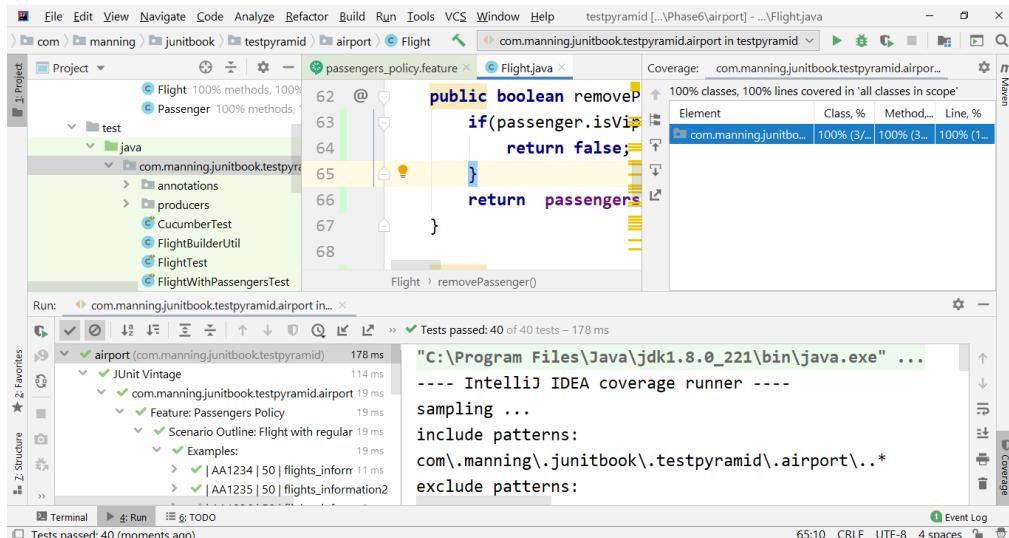


Figure 22.15 The code coverage is 100% after implementing the whole test pyramid

So, Thomas has successfully implemented a test pyramid for the flights management application, covering unit tests, integration tests, system tests, and acceptance tests.

22.6 Summary

This chapter has covered the following:

- Introducing the software testing levels (unit, integration, system, and acceptance) and the concept of test pyramid (from the lower levels to the higher ones).
- Analyzing what software tests should verify: business logic, bad input values, boundary conditions, unexpected conditions, invariants, regressions.
- Developing unit tests for two classes (`Passenger` and `Flight`) and testing: values restrictions; bad input values; boundary conditions.
- Developing tests to verify the integration between two classes using the Arquillian framework and testing how these classes interact (how to add and remove passengers to a flight).
- Developing system tests using a consumer-driven contract and moving from a mock implementation of an external functionality (the calculus of the bonus points awarded to passengers) to using the real one.
- Developing acceptance tests for a new feature to satisfy external policies, such as the passengers policy of the company, with the help of JUnit 5 and Cucumber.