`InternetExplorerDriver`) that are injected one by one into the parameterized tests.

- He defines two JUnit 5 parameterized tests, for which `getBrowserVersions` provides the test arguments ❸. This means the parameterized method is executed a number of times equal to the size of the collection returned by the `getBrowserVersions` method.
- John starts each parameterized test by keeping a reference to the web driver inside the test and initializing a `Homepage` variable, passing its constructor the web driver as an argument ❹.
- The first test opens the form authentication, logs in with the valid username and password, and checks that the login is successful ❺.
- The second test opens the form authentication, logs in with an invalid username and password, and checks that the login is unsuccessful ❻.
- After executing each test, the `quit` method of the web driver is executed, which closes all the open browser windows. The driver instance becomes garbage collectible ❼.

John has now implemented two scenarios: successful and unsuccessful logins to the website. And he has done more than this: he has created, through the `Homepage` and `LoginPage` classes, the basics of a testing library that other developers who join the project will be able to use. In listing 15.21, the calls to the testing methods flow one after the other; the test scenario is easy for any newcomer to understand. This will really speed up the development of Selenium tests at Tested Data Systems!

15.6 HtmlUnit vs. Selenium

Let's recap the similarities and differences of HtmlUnit and Selenium. Both are free and open source, and both of their versions at the time of writing (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, whereas Selenium drives a real web browser process.

HtmlUnit's benefits are that, because it's a headless web browser, test execution is faster. It also provides its own domain-specific set of assertions.

Selenium's benefits are that the API is simpler and drives native browsers, which guarantees that the behavior of tests is as close as possible to interaction with real users. Selenium also supports multiple languages and provides a "visual effect" for executing larger scenarios by running the real browsers—a reason why it has been preferred for testing authentication functionality in the real world.

In general, use HtmlUnit when your application is independent of OS features and browser-specific implementations not accounted for by HtmlUnit. And use Selenium when you require validation of specific browsers and OSs, 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 today: Spring.

Summary

This chapter has covered the following:

- Examining presentation-layer testing
- Using HtmlUnit, an open source, headless browser tool
- Using Selenium, an open source tool that drives various browsers programmatically
- Testing HTML assertions (`assertTitleEquals`, `assertTextPresent`, `notNull`) and the functionality of different web browsers, and creating standalone tests using HtmlUnit
- Testing forms, web navigation and frames, and websites developed with JavaScript using HtmlUnit
- Testing the operation of searching on Google, selecting a link, and navigating on it using Selenium and different web browsers
- Creating and testing a website authentication scenario using Selenium

16 Testing Spring applications

This chapter covers

- Understanding dependency injection
- Building and testing a Spring application
- Using `SpringExtension` for JUnit Jupiter
- Testing Spring application features 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—open source set of frameworks for creating Java applications. It is not dedicated to any particular layer, which means it can be used at the level of any layer of a Java application. 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 *library* is a collection of classes or functions that enables code reuse: we can use code created by other developers. Usually, a library is dedicated to a domain-specific

area. For example, there are libraries of computer graphics that let us quickly build three-dimensional scenes and display them on the computer screen.

On the other hand, 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 specific software. A framework supports the development of software applications by providing a working paradigm. The key difference between a library and a framework is *inversion of control* (IoC). When you call a method from a library, you are in control. But with a framework, control is inverted: the framework calls *you* (see figure 16.1). You must follow the paradigm provided by the framework and fill out your own code. The framework defines a skeleton, and you insert the features to fill out that skeleton. Your code is under the control of the framework, and the framework calls it. This way, you can focus on implementing the business logic rather than on the design.

Rod Johnson created Spring in 2003, beginning with his book *Expert One-on-One J2EE Design and Development*. The basic idea behind Spring is to simplify the traditional approach to designing enterprise applications.

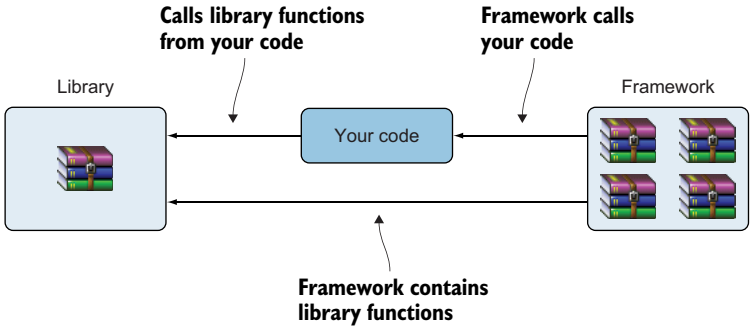


Figure 16.1 Your code calls a library. A framework calls your code.

The Spring Framework provides a comprehensive infrastructure for developing Java applications. It handles the infrastructure so that you can focus on your application and enables you 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. The objects in an application have dependencies on each other. Java cannot organize the building blocks of an application and instead leaves that task to developers and architects. You can use design patterns (factory, builder, proxy, decorator, and so on) to compose classes and objects, but this burden is on the side of the developer.

Spring implements various design patterns. The Spring Framework dependency injection (also known as IoC) pattern provides a means of composing disparate components into a fully working application.

We'll start by demonstrating the traditional approach, where dependencies need to be managed by the developer at the level of the code. You will find the first examples in the ch16-traditional folder in the source code for this chapter. In our example flight-management application developed at Tested Data Systems, each passenger who is tracked comes from a country. In figure 16.2, a `Passenger` object is dependent on a `Country` object. We initialize this dependency directly in listings 16.1 and 16.2.

Figure 16.2 A passenger is from a country, and the `Passenger` object is dependent on a `Country` object.

Listing 16.1 `Country` class

```
public class Country {
    private String name;
    private String codeName;

    public Country(String name, String codeName) {
        this.name = name;
        this.codeName = codeName;
    }

    public String getName() {
        return name;
    }

    public String getCodeName() {
        return codeName;
    }
}
```

In this listing:

- We create a `name` field together with a getter for it (1) and a `codeName` field with a getter for it (2).
- We create a constructor that initializes the `name` and `codeName` fields (3).

Listing 16.2 `Passenger` class

```
public class Passenger {
    private String name;
    private Country country;

    public Passenger(String name) {
        this.name = name;
        this.country = new Country("USA", "US");
    }

    public String getName() {
        return name;
    }
```

```
    public Country getCountry() {
        return country;
    }                                          2

}
```

In this listing:

- We create a `name` field together with a getter for it (1) and a `country` field and with a getter for it (2).
- We create a constructor that initializes the `name` and `country` fields (3). The country is effectively initialized in the `Passenger` constructor, meaning there is a tight coupling between the country and the passenger.

With this approach, we are in the general situation shown in figure 16.3. This approach has several 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.

Figure 16.3 The direct general dependency between objects A and B (`Passenger` and `Country` in the previous example)

These shortcomings favor a new approach: dependency injection.

With the dependency injection (DI) approach, objects are inserted into a container that injects the dependencies when it creates the object. This process is fundamentally the inverse of the traditional one: hence, the name *inversion of control*. Martin Fowler suggested the name *dependency injection* because it better reflects the essence of the pattern (https://www.martinfowler.com/articles/injection.html). The basic idea is to eliminate the dependency of application components on a certain implementation and to delegate to the container the rights to control class instantiation.

Moving to this new approach for the previous example, we will eliminate the direct dependency between the objects and rewrite the `Passenger` class. The next sources are found in the ch16-spring-junit4 folder from this chapter.

Listing 16.3 `Passenger` class, eliminating the tight coupling with `Country`

```
public class Passenger {
    private String name;
    private Country country;

    public Passenger(String name) {
        this.name = name;                      1
    }
```

```
    public String getName() {
        return name;
    }

    public Country getCountry() {
        return country;
    }

    public void setCountry(Country country) {
         this.country = country;
    }
}
```

The change in the code is at the level of its constructor: it no longer creates the dependent country itself (1). The country can be set through the `setCountry` method (2). As shown in figure 16.4, direct dependency has been removed.

Figure 16.4 The `Passenger` and `Country` classes with no direct dependency

XML is still the traditional way to configure Spring applications. It has the advantage of being nonintrusive—this means your code will be unaware of the fact that it is being used by a framework and is not mixed with external dependencies. In addition, when the configuration changes, the code does not need to be recompiled. This can be beneficial for testers who have less technical knowledge.

To be comprehensive, in this chapter we'll show alternatives for configuring Spring, and we'll start with XML—it is easier to understand and to follow, at least at the beginning. We'll delegate the management of dependencies to the container, which will be instructed how to do this through the configuration information in the application-context.xml file in listing 16.4. The Spring Framework will use this file to create and manage objects and their dependencies.

NOTE Objects under the control of the Spring Framework container are generally called *beans*. According to the definition in the Spring Framework documentation, "a bean is an object from the backbone of the application, managed by a Spring IoC container."

Listing 16.4 application-context.xml configuration information

```
<bean id="passenger" class="com.manning.junitbook.spring.Passenger">
    <constructor-arg name="name" value="John Smith"/>
    <property name="country" ref="country"/>
</bean>

<bean id="country" class="com.manning.junitbook.spring.Country">
    <constructor-arg name="name" value="USA"/>
    <constructor-arg name="codeName" value="US"/>
</bean>
```

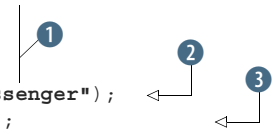
In this listing:

- We declare the `passenger` bean belonging to the class `com.manning.junitbook.spring.Passenger` (1). We initialize it by passing to it `"John Smith"` as the argument of the constructor (2), and we set the country by passing the reference to the `country` bean to the `setCountry` setter (3). Then, we close the bean definition (4).
- We declare the `country` bean belonging to the class `com.manning.junitbook.spring.Country` (5). We initialize it by passing it `"USA"` and `"US"` as arguments of the constructor (6). Then we close the bean definition (7).

To access the beans created by the container, we do the following.

Listing 16.5 Accessing the beans defined in the application-context.xml file

```
ClassPathXmlApplicationContext context =
     new ClassPathXmlApplicationContext(
                "classpath:application-context.xml");
Passenger passenger = (Passenger) context.getBean("passenger");
Country country = (Country) context.getBean("country");
```



In this listing:

- We create a `context` variable of type `ClassPathXmlApplicationContext` and initialize it to point to the application-context.xml file from the classpath (1). The class `ClassPathXmlApplicationContext` belongs to the Spring Framework; a little later, we'll show how to introduce the dependency that it belongs to in the Maven pom.xml file.
- We request the `passenger` bean from the container (2), and then we request the `country` bean (3). From now on, we can use the `passenger` and `country` variables in our program.

The following are the advantages of this approach:

- When we request an object of any type, the container will return it.
- `Passenger` and `Country` are independent of each other and do not depend on any outer libraries—they are POJOs.
- The application-context.xml file documents the system and object dependencies.
- The container controls the lifetime of created objects.
- This approach facilitates the reuse of classes and components.
- The code is cleaner (classes do not initiate auxiliary objects).
- 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; we do not need to recompile the Java source files. It is recommended that you insert in the DI (IoC) container any objects for which the implementation may change.

Thus we have taken a step between the traditional approach, where the dependencies are in the code, to the DI (IoC) approach, where objects do not know anything about each other (figure 16.5).

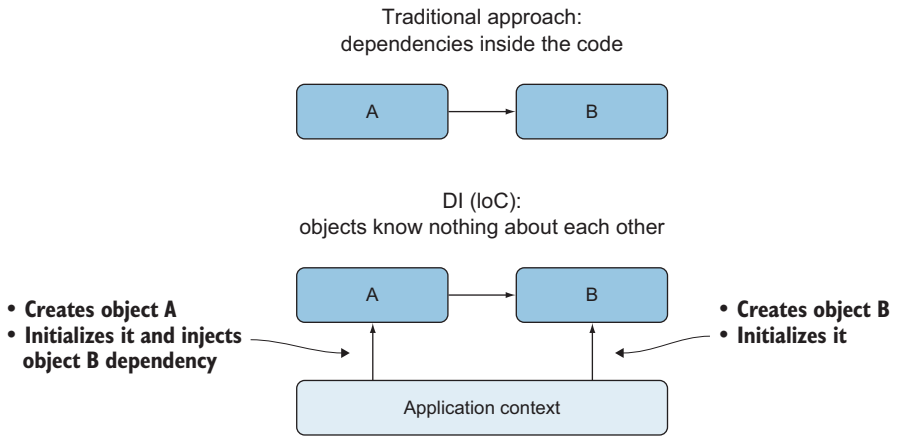


Figure 16.5 Moving from the traditional approach to DI (IoC)

In general, the work of a Spring container can be represented as shown in figure 16.6. When we instantiate and initiate the container, the application classes are combined with metadata (container configuration); as output, we get a fully configured, ready-to-work application.

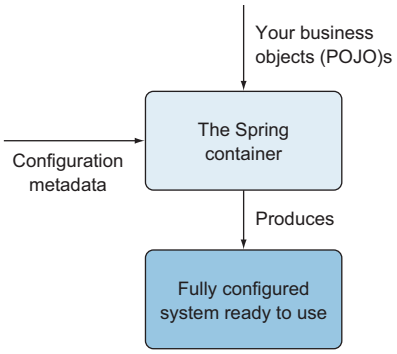


Figure 16.6 The functionality of the Spring container combines the POJOs and configuration metadata.

16.3 Using and testing a Spring application

As we mentioned earlier, due to the advantages of the Spring Framework, Tested Data Systems has introduced it into some of the company's projects, including the flight-management application. The introduction of Spring took place before the company began using JUnit 5, so we'll first look at how that took place.

16.3.1 Creating the Spring context programmatically

A few years ago, Ada was responsible for initially moving the flight-management application to Spring 4 and testing it using JUnit 4. It will be useful to see how Ada accomplished this task, because at some point you may need to work on an application using Spring 4 and/or JUnit 4 and continue its development or migrate it to Spring 5 and JUnit 5. Therefore, we closely examine what such an application looks like.

The first thing Ada had to do to work with Spring 4 and JUnit 4 was to introduce the needed dependencies in the Maven pom.xml file.

Listing 16.6 Spring 4 and JUnit 4 dependencies in pom.xml

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>4.2.5.RELEASE</version>
</dependency>

<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-test</artifactId>
    <version>4.2.5.RELEASE</version>
</dependency>

<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.12</version>
</dependency>
```

(1) (2) (3)

In this listing:

- Ada adds the `spring-context` Maven dependency (1). This is necessary in order to use classes such as `ClassPathXmlApplicationContext` that are needed to load the application context or the `@Autowired` annotation that we'll introduce later.
- She adds the `spring-test` Maven dependency (2). This is needed in order to use the `SpringJUnit4ClassRunner` runner (we'll demonstrate its use).
- She also adds the JUnit 4.12 dependency (3). Remember, Spring was added to the flight-management application a few years ago, before JUnit 5 existed.

The first test Ada wrote at that time loaded a passenger from the Spring container and verified its correctness.

Listing 16.7 First unit test for a Spring application

```
[...]
public class SimpleAppTest {

   private static final String APPLICATION_CONTEXT_XML_FILE_NAME =
           "classpath:application-context.xml";

   private ClassPathXmlApplicationContext context;

   private Passenger expectedPassenger;

   @Before
   public void setUp() {
```

(1) (2) (3) (4)

```
        context = new ClassPathXmlApplicationContext(
              APPLICATION_CONTEXT_XML_FILE_NAME);
        expectedPassenger = getExpectedPassenger();
    }

    @Test
    public void testInitPassenger() {
        Passenger passenger = (Passenger) context.getBean("passenger");
        assertEquals(expectedPassenger, passenger);
    }
}
```

In this listing:

- Ada creates a `SimpleAppTest` class (1). It defines the string to access the application-context.xml file on the classpath (2), the Spring context (3), and the `expectedPassenger` (4) to be constructed programmatically and compared with the one extracted from the Spring context.
- Before the execution of each test, she creates the context based on the string to access the application-context.xml file on the classpath (5) and creates the `expectedPassenger` programmatically (6).
- In the test, she gets the `passenger` bean from the context (7) and compares it with the one constructed programmatically (8). The configuration of the `passenger` bean from the Spring context is from listing 16.4.

To make an accurate comparison between the passenger extracted from the container and the one constructed programmatically, Ada overrides the `equals` and `hashCode` methods from the `Passenger` and `Country` classes (listings 16.8 and 16.9).

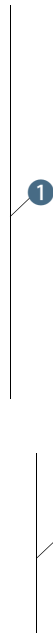
Listing 16.8 Overridden `equals` and `hashCode` methods from the `Passenger` class

```
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 name.equals(passenger.name) &&
               Objects.equals(country, passenger.country);
    }

    @Override
    public int hashCode() {
        return Objects.hash(name, country);
    }
}
```

In this listing, Ada constructs the `Passenger` methods `equals` (1) and `hashCode` (2) based on the `name` and `country` fields.

Listing 16.9 Overridden `equals` and `hashCode` methods from the `Country` class

```
public class Country {
    [...]
    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;

        Country country = (Country) o;

        if (codeName != null ?
           !codeName.equals(country.codeName) :
            country.codeName != null) return false;
        if (name != null ?
           !name.equals(country.name) :
           country.name != null) return false;

        return true;
    }

    @Override
    public int hashCode() {
        int result = 0;
        result = 31 * result + (name != null ? name.hashCode() : 0);
        result = 31 * result + (codeName != null ?
                 codeName.hashCode() : 0);
        return result;
    }
}
```

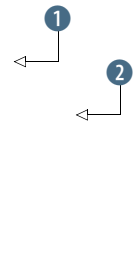


In this listing, Ada constructs the `Country` methods `equals` 1 and `hashCode` 2 based on the `name` and `codeName` fields.

The expected passenger is constructed programmatically through the `getExpectedPassenger` method from the `PassengerUtil` class.

Listing 16.10 `PassengerUtil` class

```
public class PassengerUtil {

    public static Passenger getExpectedPassenger() {
        Passenger passenger = new Passenger("John Smith");

        Country country = new Country("USA", "US");
        passenger.setCountry(country);

        return passenger;
    }
}
```



In the `getExpectedPassenger` method, Ada first creates the passenger `"John Smith"` 1 and the country `"USA"` 2 and sets that as the country of the passenger 3. She returns this passenger at the end of the method 4.

Running the newly created `SimpleAppTest` is successful, as shown in figure 16.7. Ada has verified that the bean extracted from the Spring container is equal to the beans constructed programmatically.

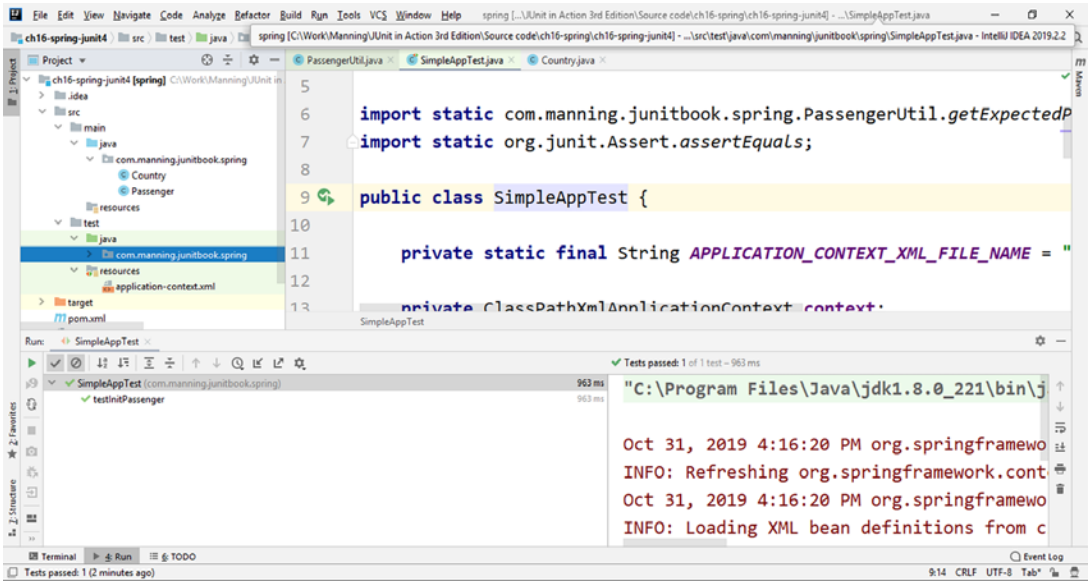


Figure 16.7 The result of running `SimpleAppTest`

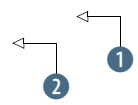
16.3.2 Using the Spring TestContext framework

The Spring TestContext Framework offers unit and integration testing support independent of the testing framework in use: 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.

Ada decided to refactor the initial test to use the capabilities of the TestContext framework.

Listing 16.11 `SpringAppTest` class

```
[...]
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration("classpath:application-context.xml")
public class SpringAppTest {

   @Autowired
   private Passenger passenger;
   private Passenger expectedPassenger;
```



```
    @Before
    public void setUp() {
        expectedPassenger = getExpectedPassenger();
    }

    @Test
    public void testInitPassenger() {
        assertEquals(expectedPassenger, passenger);
    }

}
```

In this listing:

- Ada runs the test with the help of `SpringJUnit4ClassRunner` (1). She also annotates the test class to look for the context configuration in the application-context.xml file from the classpath (2). These annotations belong to the `spring-test` dependency and serve to replace the programmatic creation of the context, as we saw in listing 16.7. Through these annotations, she takes care of creating the context from the application-context.xml file from the classpath and injects the defined beans into the test.
- She declares a field of type `Passenger` and annotates it as `@Autowired` (3). Spring will automatically look in the container and try to autowire the declared `passenger` field to a unique injected bean of the same type, declared in the container. It is important that there is a single bean of type `Passenger` in the container; otherwise, there will be ambiguity, and the test will fail and throw an `UnsatisfiedDependencyException`.

`SpringAppTest` is shorter than `SimpleAppTest` because Ada has pushed even more control onto the Spring Framework by using the `@RunWith` and `@ContextConfiguration` annotations that help to initialize the context automatically.

The result of running the newly created `SimpleAppTest` and `SpringAppTest` is successful, as shown in figure 16.8. Ada has verified in two ways that the bean extracted from the Spring container is equal to a bean constructed programmatically.

16.4 Using SpringExtension for JUnit Jupiter

`SpringExtension`, introduced in Spring 5, is used to integrate Spring TestContext with a JUnit 5 Jupiter test. `SpringExtension` is used with the JUnit 5 Jupiter `@ExtendWith` annotation.

Continuing her work on developing and testing the Spring-based flight-management application, Ada will first migrate to JUnit 5, following the steps we presented in chapter 4. Then, she will continue to add new features to the application that will now be tested with JUnit 5. You will find these examples in this chapter's ch16-spring-junit5 folder.

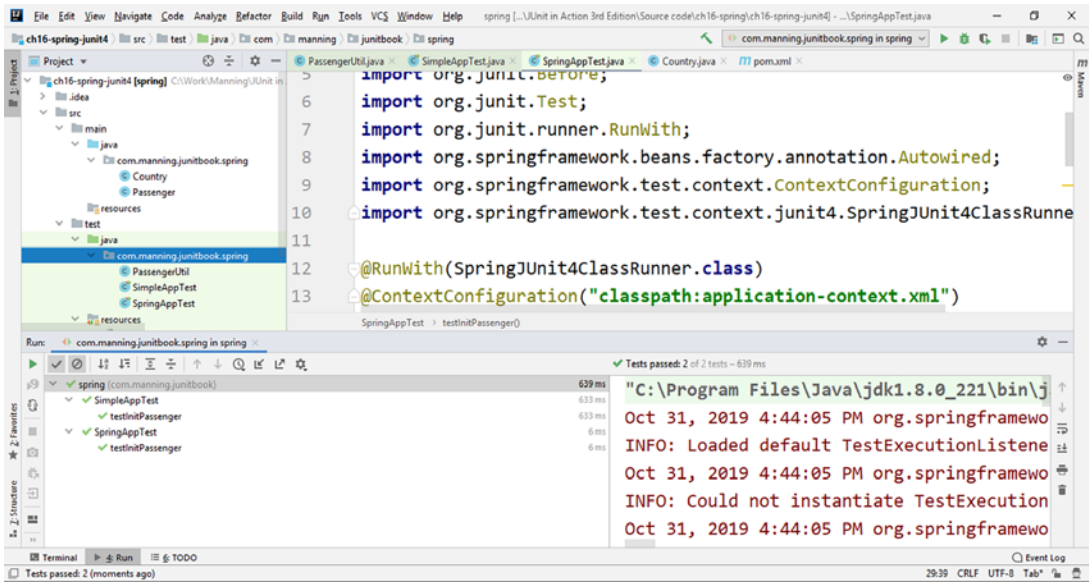


Figure 16.8 `SimpleAppTest` and `SpringAppTest` are successfully executed as Spring JUnit 4 tests.

The first thing Ada must do is replace the Spring 4 and JUnit 4 dependencies from the pom.xml file with the Spring 5 and JUnit 5 dependencies.

Listing 16.12 pom.xml file with Spring 5 and JUnit 5 dependencies

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>5.2.0.RELEASE</version>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-test</artifactId>
    <version>5.2.0.RELEASE</version>
</dependency>
<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>
```

1
2
3
4

In this listing:

- Ada introduces the `spring-context` (1) and `spring-test` (2) 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` class and the `@ContextConfiguration` annotation. Ada has replaced the same dependencies from version 4.2.5.RELEASE.
- She introduces the `junit-jupiter-api` (3) and `junit-jupiter-engine` (4) dependencies that are needed to test the application with JUnit 5. She replaces the previously used JUnit 4.12 dependency, as we did for migration processes between JUnit 4 and JUnit 5 (see chapter 4).

Ada then migrates the code from using Spring 4 and JUnit 4 to using Spring 5 and JUnit 5.

Listing 16.13 `SpringAppTest` class

```
[...]
@ExtendWith(SpringExtension.class)
@ContextConfiguration("classpath:application-context.xml")
public class SpringAppTest {

   @Autowired
   private Passenger passenger;
   private Passenger expectedPassenger;

   @BeforeEach
   public void setUp() {
      expectedPassenger = getExpectedPassenger();
   }

   @Test
   public void testInitPassenger() {
      assertEquals(expectedPassenger, passenger);
      System.out.println(passenger);
   }
}
```

In this listing:

- Ada extends the test with the help of `SpringExtension` (1). She also annotates the test class to look for the context configuration in the application-context.xml file from the classpath (2). These annotations belong to the `spring-test` dependency (version 5.2.0.RELEASE this time) and serve to replace the work that has been done by JUnit 4 runners. Through these annotations, Ada takes care of creating the context from the application-context.xml file from the classpath and injects the defined beans into the test. For more details about JUnit 5 extensions, see chapter 14.
- She keeps the field of type `Passenger` and annotates it with `@Autowired` (3). Spring will automatically look in the container and try to autowire the declared

`passenger` field to a unique injected bean of the same type, declared in the container. It is important that there is a single bean of type `Passenger` in the container; otherwise, there will be ambiguity, and the test will fail and throw an `UnsatisfiedDependencyException`.

- She replaces the JUnit 4 `@Before` annotation with the JUnit 5 `@BeforeEach` annotation ❹. This is not specific to 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` is successful, as shown in figure 16.9. Ada has verified the correct migration from working with Spring 4 and JUnit 4 to Spring 5 and JUnit 5. Now she is ready to continue her work and add new features to the Spring flight-management application.

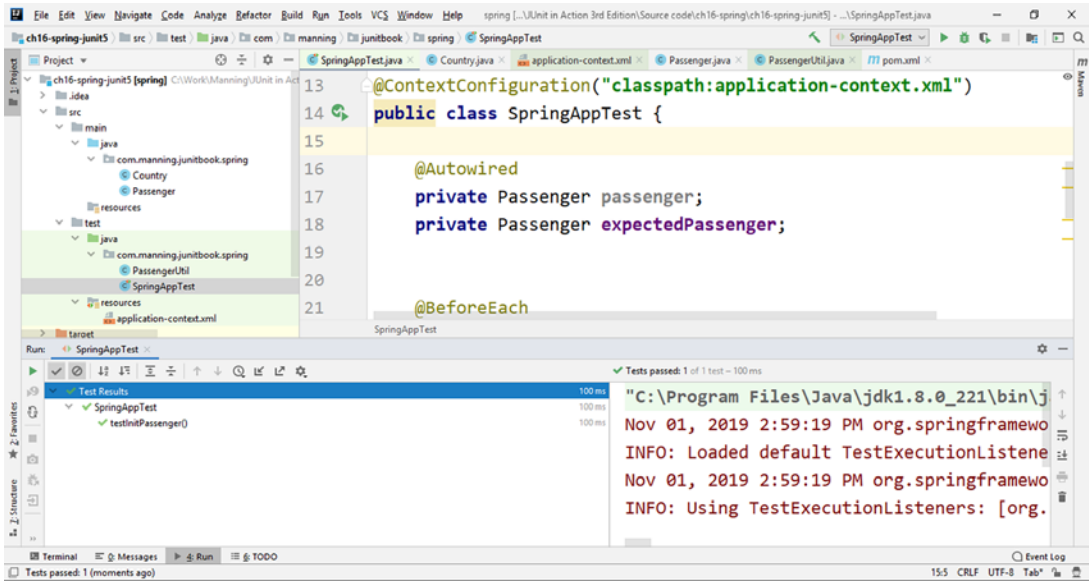


Figure 16.9 `SpringAppTest` **successfully executed as a Spring JUnit 5 test**

16.5 Adding a new feature and testing it with JUnit 5

Next, we will follow Ada as she adds a new feature to the current Spring 5 flight-management application and tests 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, and implementing the observer pattern with the help of Spring 5–defined events and listeners.

NOTE We started with an XML-based configuration as a gentle introduction. But the Spring Framework is a broad topic, and we are mostly discussing

things related to testing these applications with JUnit 5. For a comprehensive look at this subject, which also shows the configuration possibilities in detail, we recommend *Spring in Action* by Craig Walls (Manning, www.manning.com/books/spring-in-action-sixth-edition).

The feature that Ada has been assigned to implement requires tracking the registration of passengers through the registration manager of the flight-management system. The customer requires that whenever a new passenger is registered, the registration manager must answer with a confirmation. This functionality follows the idea of the observer pattern (figure 16.10), which we are going to examine and implement. You will find these examples in the chapter's ch16-spring-junit5-new-feature folder.

DEFINITION *Observer pattern*—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 in which the observers are interested.

Such an observer can 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.

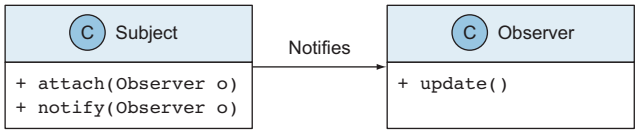


Figure 16.10 The subject notifies the observer about events of interest to it.

In this case, Ada identifies 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 is the event that is generated on the registration-manager side. This event must be broadcast to the observers (listeners) that are interested.
- The registration listener is the observer that will receive the registration event and confirm the registration by setting the passenger as registered in the system and by printing a message.

The functionality of this concrete registration system is shown in figure 16.11. Ada starts working on it by changing the `Passenger` class to be aware of the registration status.

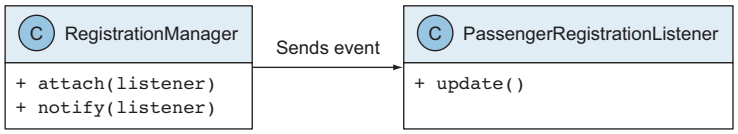


Figure 16.11 The registration manager informs the listener about registration events.

Listing 16.14 Modified `Passenger` class

```
public class Passenger {
    [...]
    private boolean isRegistered;

    [...]
    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 &&
                Objects.equals(name, passenger.name) &&
                Objects.equals(country, passenger.country);
    }

    @Override
    public int hashCode() {
        return Objects.hash(name, country, isRegistered);
    }

}
```

1
2
3
4

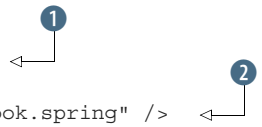
In this listing:

- Ada adds the `isRegistered` field to the `Passenger` class together with the getter and setter for it 1.
- She modifies the `toString` 2, `equals` 3, and `hashCode` 4 methods to also take into consideration the newly introduced `isRegistered` field.

Ada also modifies the application-context.xml file (listing 16.15). Declaring data beans in the XML configuration file is suitable for testing purposes, as in this case. The XML is easy to change, the code does not need to be recompiled, and we can have different configurations in different environments without touching the code.

Listing 16.15 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"/>
</bean>
<context:component-scan base-package="com.manning.junitbook.spring" />
```



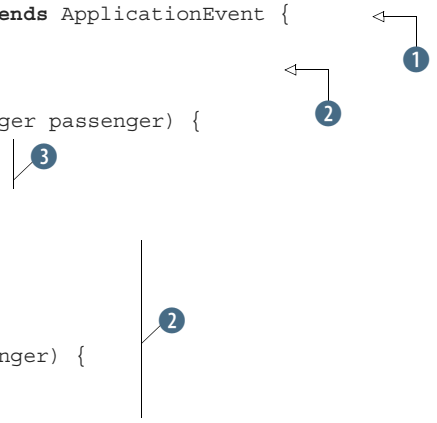
In this listing:

- Ada adds the `isRegistered` field to the initialization of the `passenger` bean (1).
- She inserts the directive to require Spring to also scan the package `com.manning.junitbook.spring` to find components (2). As a result, in addition to the beans defined in the XML, other beans will be defined in the code from the indicated base package, using annotations.

Ada creates the `PassengerRegistrationEvent` class to define the custom event that is happening in the registration system.

Listing 16.16 `PassengerRegistrationEvent` class

```
public class PassengerRegistrationEvent extends ApplicationEvent {

   private Passenger passenger;

   public PassengerRegistrationEvent(Passenger passenger) {
      super(passenger);
      this.passenger = passenger;
   }

   public Passenger getPassenger() {
      return passenger;
   }

   public void setPassenger(Passenger passenger) {
      this.passenger = passenger;
   }
}
```



In this listing:

- Ada defines the `PassengerRegistrationEvent` class that extends `ApplicationEvent` (1). `ApplicationEvent` is the Spring abstract class to be extended by all application events.
- She keeps a reference to the `Passenger` object whose registration generated the event, together with a getter and a setter on it (2).
- In the `PassengerRegistrationEvent`, she calls the constructor of the superclass `ApplicationEvent` that receives as an argument the source of the event: the passenger (3).

We have mentioned that there are alternative ways to create Spring beans. The XML approach is 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 appropriate for functional beans that generally do not change.

Ada creates the `RegistrationManager` class to serve as an event generator.

Listing 16.17 `RegistrationManager` class

```
[...]
@Service                                                          (1)
public class RegistrationManager implements ApplicationContextAware {   (2)
   private ApplicationContext applicationContext;

   public ApplicationContext getApplicationContext() {            (3)
      return applicationContext;
   }

   @Override
   public void setApplicationContext(ApplicationContext applicationContext)
                                       throws BeansException {
      this.applicationContext = applicationContext;               (4)
   }

}
```

In this listing:

- Ada annotates the class with the `@Service` annotation (1). This means Spring will automatically create a bean of the type of this class. Remember that in the application-context.xml file, Ada inserted the `component-scan` directive with `base-package="com.manning.junitbook.spring"`, so Spring will look for annotation-defined beans in that package.
- The `RegistrationManager` class implements the `ApplicationContextAware` interface (2). As a result, `RegistrationManager` will have a reference to the application context that it will use to publish events.
- In the class, Ada keeps a reference to the application context and a getter for it (3).
- The `setApplicationContext` method inherited from `ApplicationContextAware` initializes the field `applicationContext` as a reference to the `applicationContext` injected by Spring as an argument of this method (4).

Ada creates the `PassengerRegistrationListener` class to serve as an observer of passenger registration events.

Listing 16.18 `PassengerRegistrationListener` class

```
[...]
@Service                                                          (1)
public class PassengerRegistrationListener {
```

```
    @EventListener
    public void confirmRegistration(PassengerRegistrationEvent
                                    passengerRegistrationEvent) {
        passengerRegistrationEvent.getPassenger().setIsRegistered(true);
        System.out.println("Confirming the registration
                           for the passenger: "
                + passengerRegistrationEvent.getPassenger());
    }
}
```

In this listing:

- Ada annotates the class with `@Service` (1). As a result, Spring will automatically create a bean of the type of this class. Remember that, in the application-context.xml file, she inserted the `component-scan` directive with `base-package="com.manning.junitbook.spring"`, so Spring will look for annotation-defined beans in that package.
- The `confirmRegistration` method receives as an argument a `PassengerRegistrationEvent` and is annotated with `@EventListener` (2). Thus Spring will automatically register this method as a listener (observer) for `PassengerRegistrationEvent` type events. Whenever such an event occurs, this method will be executed.
- In the method, once the passenger registration event is received, Ada confirms the registration by setting the passenger as registered in the system and printing a message (3).

Ada finally creates the `RegistrationTest` class to verify the behavior of the code she has implemented.

Listing 16.19 `RegistrationTest` class

```
[...]
@ExtendWith(SpringExtension.class)
@ContextConfiguration("classpath:application-context.xml")
public class RegistrationTest {

    @Autowired
    private Passenger passenger;

    @Autowired
    private RegistrationManager registrationManager;

    @Test
    public void testPersonRegistration() {
        registrationManager.getApplicationContext()
            .publishEvent(new PassengerRegistrationEvent(passenger));
        assertTrue(passenger.isRegistered());
    }

}
```

In this listing:

- Ada extends the test with the help of `SpringExtension` (1). She also annotates the test class to look for the context configuration in the application-context.xml file from the classpath (2). Through these annotations, she takes care of creating the context from the application-context.xml file from the classpath and injecting the defined beans into the test.
- She declares a field of type `Passenger` and a field of type `RegistrationManager` and annotates them as `@Autowired` (3). Spring will automatically look in the container and try to autowire the declared `Passenger` and `RegistrationManager` fields to unique injected beans of the same type, declared in the container. It is important that there is a single bean of each of these types in the container; otherwise, there will be ambiguity, and the test will fail and throw an `UnsatisfiedDependencyException`. Ada declares a `Passenger` bean in the application-context.xml file. And she annotates `RegistrationManager` with `@Service`, so Spring will automatically create a bean of the type of this class.
- In the test method, Ada uses the `RegistrationManager` field to publish an event of type `PassengerRegistrationEvent` with the help of the reference to the application context it is holding (4). Then she checks that the registration status of the passenger has really changed (5).

The result of running the newly created `RegistrationTest` is successful, as shown in figure 16.12. Ada has implemented and tested the passengers' registration feature using the capabilities of JUnit 5 and Spring 5.

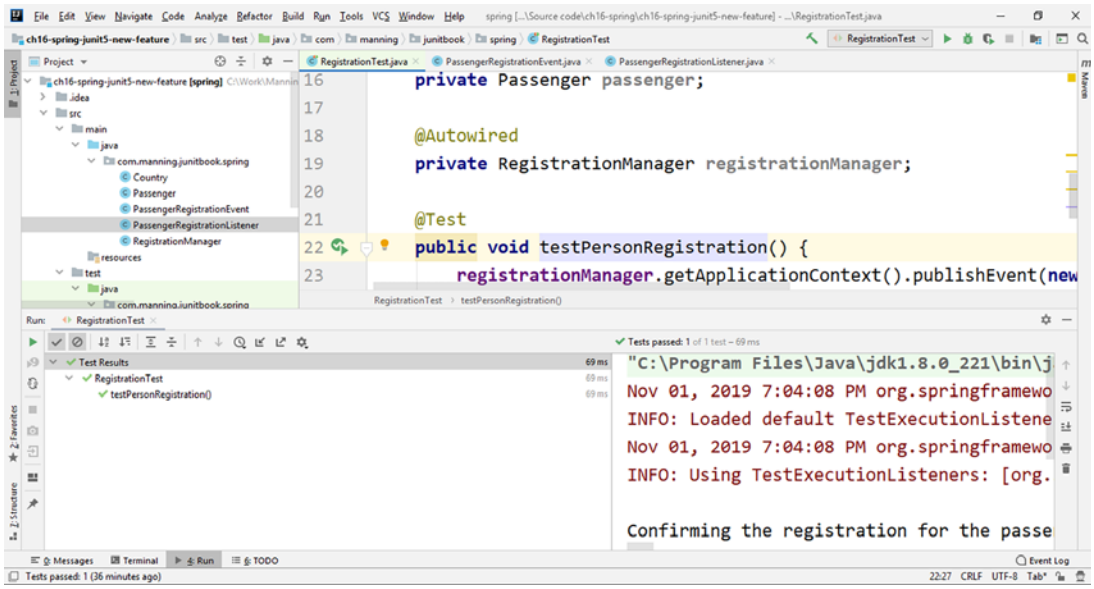


Figure 16.12 `RegistrationTest` **successfully executes as a Spring JUnit 5 test.**

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.

Summary

This chapter has covered the following:

- Introducing the Spring Framework, a widely used, lightweight, flexible, universal framework for creating Java applications.
- Demonstrating the dependency injection (inversion of control) pattern that is the basis of Spring. Objects are inserted into a container that injects 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 DI (IoC) container.
- Using and testing a Spring application both by creating the Spring context programmatically and by using the Spring TestContext framework.
- Using `SpringExtension` for JUnit Jupiter to create JUnit 5 tests for a Spring application.
- Developing a new feature for a Spring application using beans created through annotations, implementing the observer pattern with the help of Spring 5–defined events and listeners, and testing the application with JUnit 5.

17 Testing Spring Boot applications

This chapter covers

- Creating a project with Spring Initializr
- Moving a Spring application tested with JUnit 5 to Spring Boot
- Implementing a test-specific Spring Boot configuration
- Testing a Spring Boot application feature 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 that are ready to run immediately. Spring Boot is an extension of the Spring Framework that strongly reduces the initial Spring application's configuration: a Spring Boot application is preconfigured and given dependencies to outside libraries so that we can start using it. Most Spring Boot applications need very little (or no) Spring configuration.

DEFINITION *Convention over configuration*—A software design principle adopted by various frameworks to reduce the number of configuration actions a programmer using that framework needs to perform. The programmer only needs to specify the nonstandard configuration of the application.

17.1 Introducing Spring Boot

A shortcoming of the Spring Framework is that it takes some time to make an application ready for production. The configuration is time consuming and can be a little overwhelming for new developers.

Spring Boot is built on top of the Spring Framework as an extension of it. It strongly supports developers through a convention-over-configuration approach that can help with the rapid creation of an application. As most Spring Boot applications need little Spring configuration, you can focus on the business logic instead of infrastructure and configuration. The business logic is the part of the program focusing on the business rules that determine the way the application works.

In this chapter, we will examine the following Spring Boot features:

- Creating standalone Spring applications
- Automatically configuring Spring when possible
- Embedding web servers like Tomcat and Jetty directly (no need to deploy war files)
- Providing preconfigured Maven project object models (POMs)

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 applications, we will use Spring Initializr to generate a project with only what we need to start quickly. To do this, we go to https://start.spring.io/ and insert the configuration data of our new project on the web page. Spring Boot generates a skeleton for the application, and we transfer into it the business logic for tracking passenger registration through the registration manager of the example flight-management system we implemented in chapter 16.

As shown in figure 17.1, we choose to create a new Java project managed by Maven as a build tool. The group name is `com.manning.junitbook`, and the artifact ID is `spring-boot`. The Spring Boot version at the time of writing this chapter was 2.2.2; your version while studying this chapter may be higher. The application to be generated is found in the ch17-spring-boot-initializr folder of the source code for this chapter.

In figure 17.2, we provide more configuration for the newly created project. We leave the description as "Demo project for Spring Boot" and the packaging as .jar. We are given the chance to add new dependencies (Web, Security, JPA, Actuator, Devtools). We are not adding anything for the moment, as our first goal is to move the Spring application from chapter 16 to Spring Boot, but you should be aware of this possibility: with a few clicks, you can obtain the needed dependencies directly in the Maven pom.xml file.

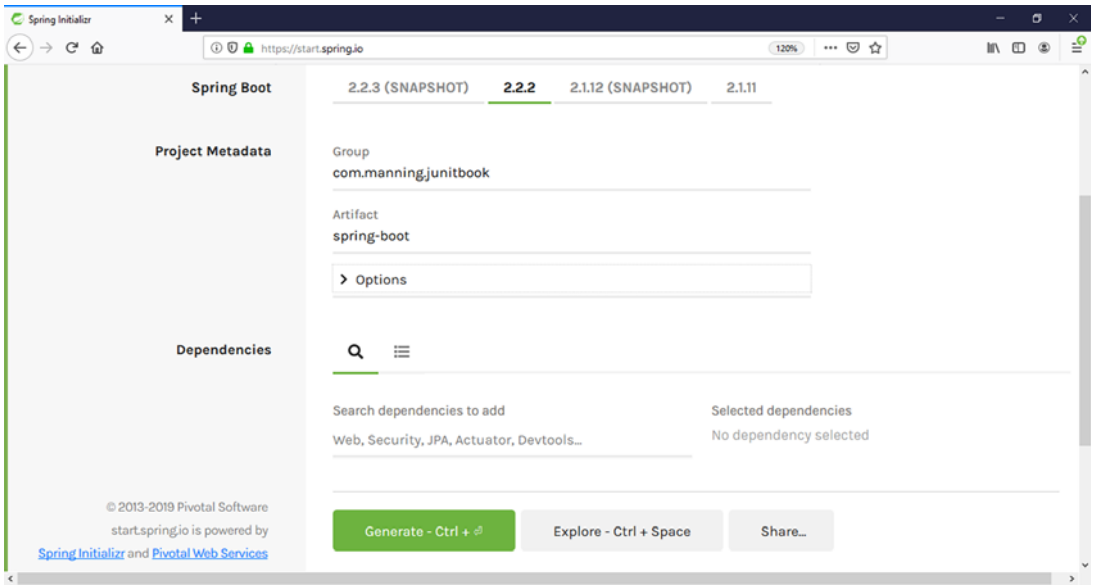


Figure 17.1 Creating a new Spring Boot project with the help of Spring Initializr. The project is managed by Maven as the build tool, the group name is `com.manning.junitbook`, and the artifact ID is `spring-boot`.

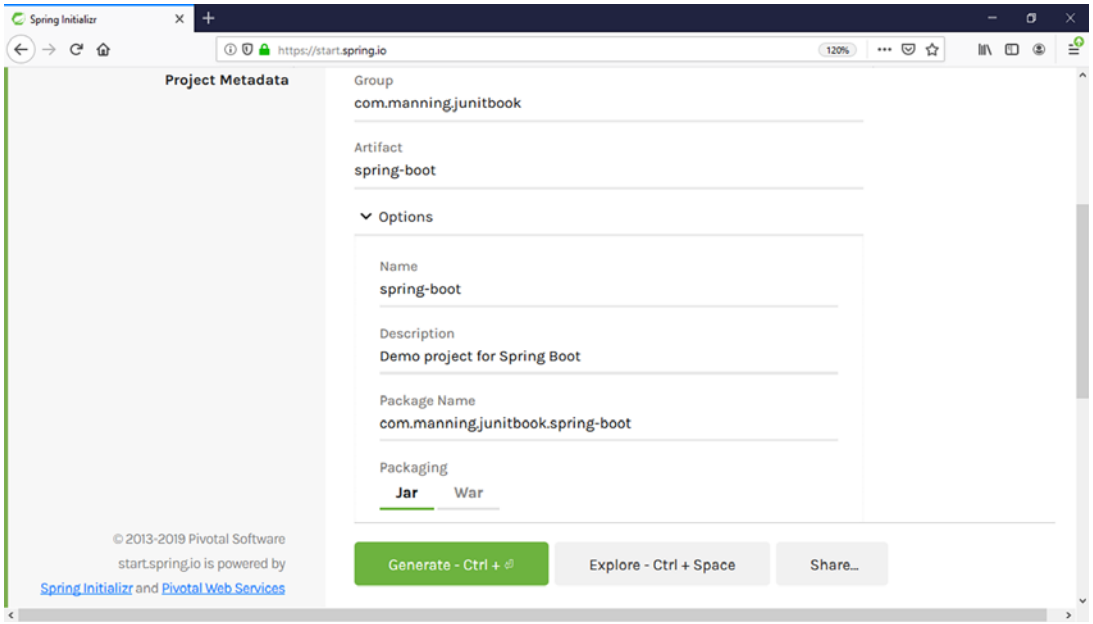


Figure 17.2 Available options when creating a new Spring Boot project with Spring Initializr: the description is "Demo project for Spring Boot," and the packaging is .jar. We can also add new dependencies (Web, Security, JPA, Actuator, Devtools).

When we click the Explore – Ctrl + Space button, we 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). We can click Download the ZIP or, on the previous screen, Generate - Ctrl +, and we'll get the archive containing the project.

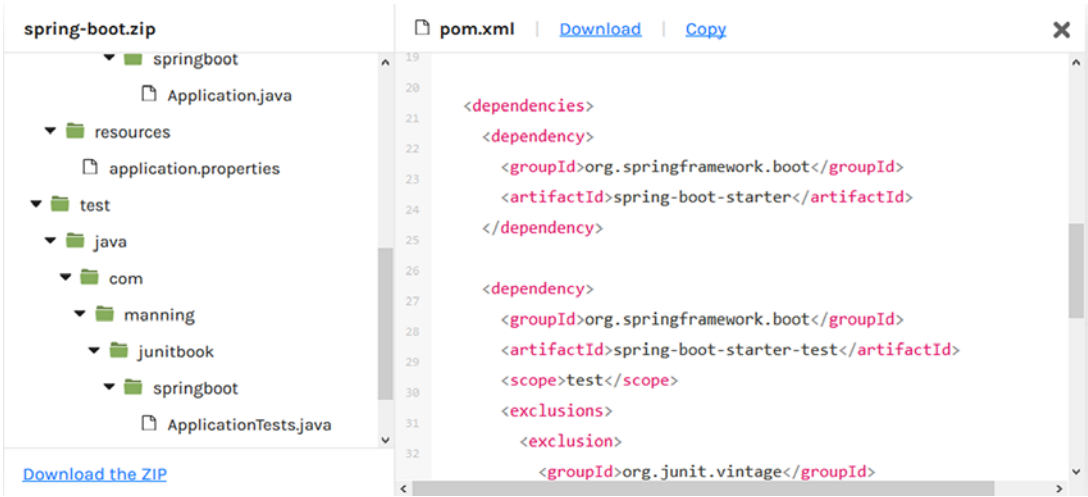


Figure 17.3 The structure of the project (expanded folders) and the content of the Maven pom.xml file (including the `spring-boot-starter` dependency)

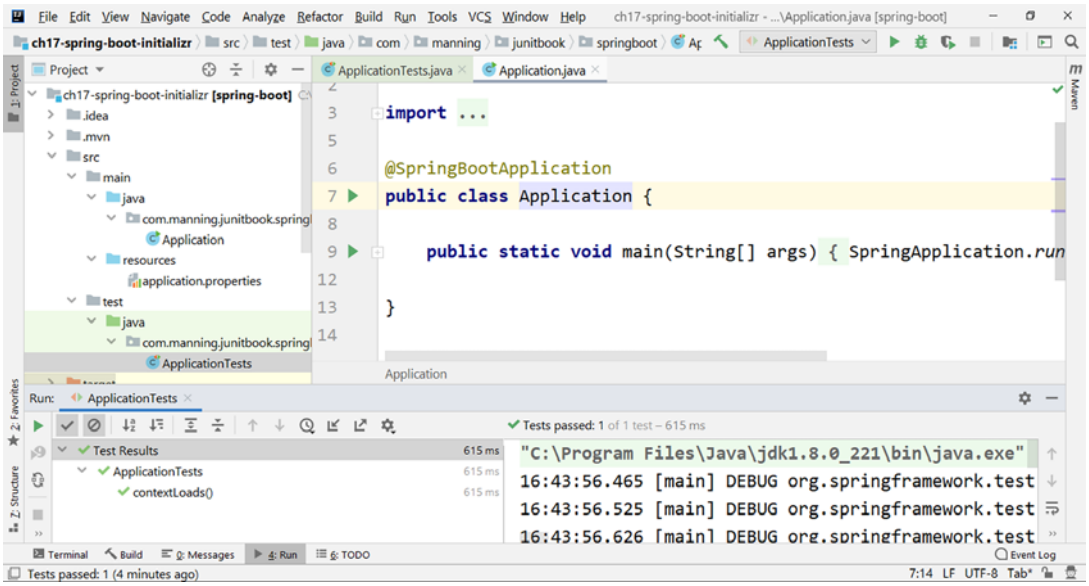


Figure 17.4 Loading the generated Spring Boot application into IntelliJ IDEA, and successfully running the test that is automatically provided

Opening the generated project, we see its structure containing the Maven pom.xml file, a `main` method in the `Application` class, a test in the `ApplicationTests` class, and an application.properties resource file (empty for now, but we'll use it later). So, with a few clicks and choices, in seconds we have a new Spring application. We can simply run the tests (figure 17.4), and then we'll focus on moving our previously created Spring application from chapter 16 to this new Spring Boot structure.

In this example, we use different ways to configure the Spring container. As we mentioned in chapter 16, XML is still the traditional approach for configurations, but recently there is a tendency for developers to work more with annotations. For our testing purposes, we'll use XML: it 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.

17.3 Moving the Spring application to Spring Boot

Now we'll begin moving our previously developed Spring application to the skeleton provided by Spring Boot. You will find this application in the chapter's ch17-spring-boot-initializr-old-feature folder. For a successfully systematic and organized migration, we'll do the following:

1. Introduce two new packages in the application generated by Spring Initializr: `com.manning.junitbook.springboot.model`, where we'll keep the domain classes `Passenger` and `Country`; and `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 figure 17.5.

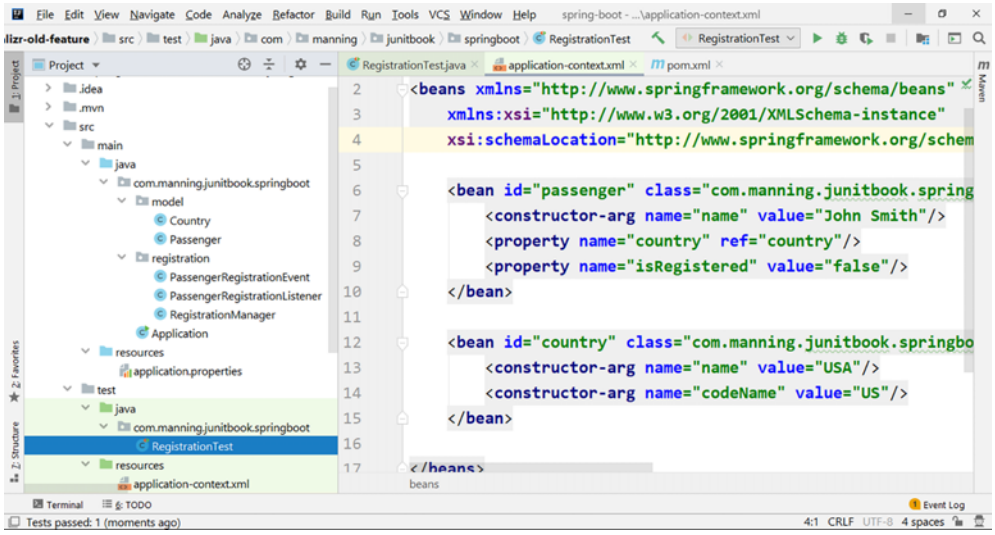


Figure 17.5 The new structure of the Spring Boot registration application. The classes have been distributed to two different packages.

2. The application-context.xml file will no longer contain the directive `<context:component-scan base-package="..." />`. We'll provide an equivalent using Spring annotations at the level of the test. The updated application-context.xml file will look as shown in listing 17.1.
3. Introduce new annotations in the `RegistrationTest` file to replace the directive that we eliminated from the application-context.xml file (listing 17.2).
4. Keep only the `RegistrationTest` class for testing our application, as the `SpringAppTest` class was just 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.

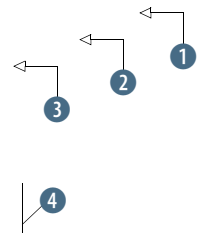
Listing 17.1 application-context.xml configuration file

```
<bean id="passenger"
        class="com.manning.junitbook.springboot.model.Passenger">
    <constructor-arg name="name" value="John Smith"/>
    <property name="country" ref="country"/>
    <property name="isRegistered" value="false"/>
</bean>

<bean id="country"
        class="com.manning.junitbook.springboot.model.Country">
    <constructor-arg name="name" value="USA"/>
    <constructor-arg name="codeName" value="US"/>
</bean>
```

The updated application-context.xml file keeps the definition of the `passenger` and `country` beans, but we remove the original directive `<context:component-scan base-package="..." />`. We'll provide an equivalent using Spring annotations at the level of the test.

Listing 17.2 Rewritten Spring Boot `RegistrationTest`

```
@SpringBootTest
@EnableAutoConfiguration
@ImportResource("classpath:application-context.xml")
class RegistrationTest {

   [...]

   @Autowired
   private RegistrationManager registrationManager;

   [...]
}
```



In this listing:

- We annotate our class with the `@SpringBootTest` annotation (1) initially provided by the test generated by Spring Boot. This annotation provides a few

features; we are using one that, together with `@EnableAutoConfiguration` (2), searches in the current test class package and its subpackages for bean definitions. This way, it can discover and autowire the `RegistrationManager` bean (4).

- There are still beans defined at the level of the XML configuration, which we import with the help of the `@ImportResource` annotation (3).
- The rest of the original `RegistrationTest` class remains the same as the Spring Core application built in chapter 16.

Running the current test is successful, as shown in figure 17.6.

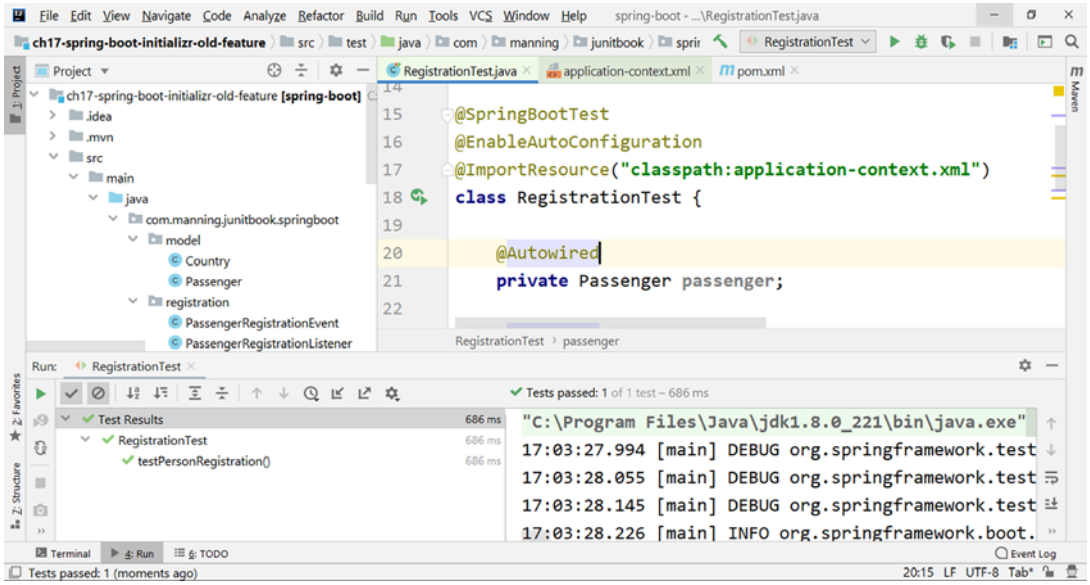


Figure 17.6 Successfully running `RegistrationTest`. The registration functionality works fine after we move the application to Spring Boot.

17.4 Implementing a test-specific configuration for Spring Boot

So far, we are using a Spring Boot application with the business logic of the original Spring Core application. The bean configuration is still at the level of the XML file. As we have mentioned, XML is the traditional approach for Spring configuration, as it is nonintrusive (the classes do not need external dependencies) and can be easily changed (the source files do not need to be recompiled). The `country` and `passenger` data beans are for testing purposes, so keeping them in an XML file will enable different testers with less programming skill to quickly create their own configurations. These examples are in the chapter's ch17-spring-boot-beans folder.

> **NOTE** We are focusing on the different configuration alternatives for testing Spring applications. For more comprehensive discussions of the topic, we recommend these two books by Craig Walls: *Spring in Action* (Manning, www.manning.com/books/spring-in-action-sixth-edition) and *Spring Boot in Action* (Manning, www.manning.com/books/spring-boot-in-action).

Now that the developers at Tested Data Systems have moved to Spring Boot, they would like to try a different configuration approach that is closer to the Spring Boot spirit. They also want 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 bean definitions 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 existing test data beans from application-context.xml will be moved to this `@TestConfiguration`-annotated class, while the functional beans (`RegistrationManager` and `PassengerRegistrationListener` from the code base) will be declared using the `@Service` annotation.

Mike introduces the following test configuration class instead of the application-context.xml file.

Listing 17.3 `TestBeans` class

```
[...]
@TestConfiguration                                          (1)
public class TestBeans {

    @Bean                                                   (2)
    Passenger createPassenger() {
        Passenger passenger = new Passenger("John Smith");  (3)
        passenger.setCountry(createCountry());
        passenger.setIsRegistered(false);
        return passenger;
    }

    @Bean                                                   (4)
    Country createCountry() {
        Country country = new Country("USA", "US");         (5)
        return country;
    }
}
```

In this listing:

- Mike introduces the new class `TestBeans` that replaces the existing application-context.xml file and annotates it with `@TestConfiguration` (1). Its purpose is to provide the beans while executing the tests.

- He writes the `createPassenger` method ❸ and annotates it with `@Bean` ❷. Its purpose is to create and configure a `Passenger` bean that will be injected into the tests.
- He writes the `createCountry` method ❺ and annotates it with `@Bean` ❹. Its purpose is to create and configure a `Country` bean that will be injected into the tests.

Mike modifies the existing test to use the `@TestConfiguration`-annotated class instead of the application-context.xml file for the creation and configuration of the bean.

Listing 17.4 Modified `RegistrationTest` class

```
@SpringBootTest
@Import(TestBeans.class)                    ←1
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());
   }

}
```

In this listing, the change that is propagated in the test class is the replacement of the existing `@EnableAutoConfiguration` and `@ImportResource` annotations with `@Import(TestBeans.class)` ❶. This way, Mike explicitly registers the beans defined in `TestBeans` in the class that contains the tests.

The advantage of making the move to using the `@TestConfiguration` annotation is not only that this approach is more specific to Spring Boot or that we are replacing two annotations with one. Working with Java-based defined beans is *type-safe*: Java prevents errors, and the compiler will report issues if we configure incorrectly. Search and navigation are much simpler, because we take advantage of the IDE. 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 is successful, as shown in figure 17.7. Notice that the application-context.xml configuration file has been removed.

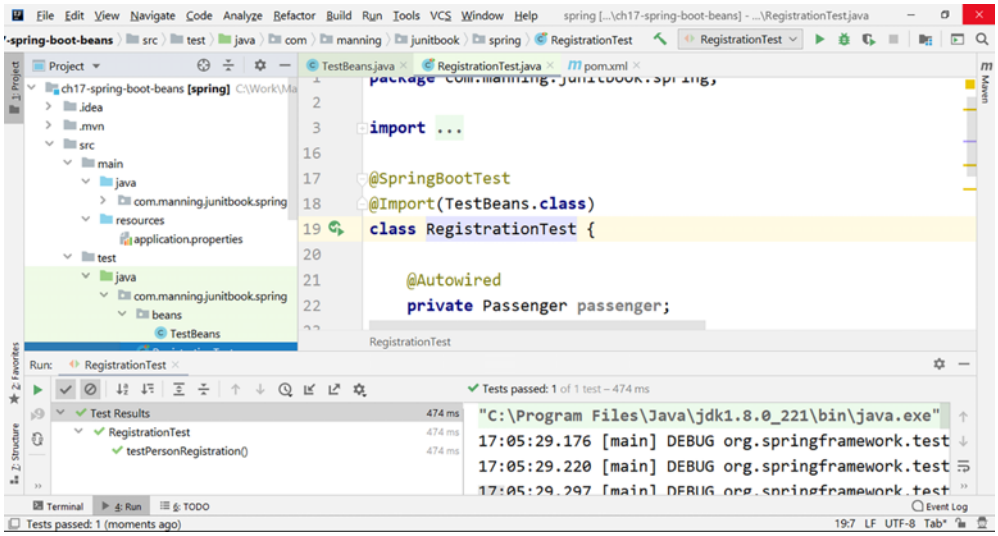


Figure 17.7 Successfully running `RegistrationTest`. The registration functionality works fine after we implement a test-specific configuration for Spring Boot.

17.5 Adding and testing a new feature in the Spring Boot application

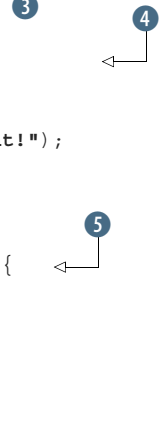
Mike receives a requirement to introduce a new feature: the application must be able to create and set up flights, as well as add passengers to and remove them from flights. He will introduce a new class to model this case: `Flight`. He must check that a list of passengers can be correctly registered on a flight and that all the people on the list receive a registration confirmation. First, he has to introduce the `Flight` class. These examples are in the chapter's ch17-spring-boot-new-feature folder.

Listing 17.5 `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;
    }
```

1
2
3

```
    public Set<Passenger> getPassengers() {
        return Collections.unmodifiableSet(passengers);
    }

    public boolean addPassenger(Passenger passenger) {
         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) {
         return passengers.remove(passenger);
    }

    @Override
    public String toString() {
         return "Flight " + getFlightNumber();
    }

}
```



In this listing:

- Mike provides three fields—`flightNumber`, `seats`, and `passengers`—to describe a `Flight` (1).
- He provides a constructor that has the first two parameters (2).
- He also provides three getters (3) that address the fields he 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.
- 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 (4).
- Mike provides the possibility to remove a passenger from a flight (5).
- Finally, he overrides the `toString` method (6).

Mike describes the list of passengers on the flight using the CSV file in the following listing. A passenger is described with a name and a country code. There are 20 passengers total.

Listing 17.6 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 parses the CSV file and populates the flight with the corresponding passengers. The information is brought in from an external file to the application memory.

Listing 17.7 `FlightUtilBuilder` class

```
[...]
@TestConfiguration                                                    (1)
public class FlightBuilder {

    private static Map<String, Country> countriesMap =                (2)
            new HashMap<>();

    static {                                                          (3)
        countriesMap.put("AU", new Country("Australia", "AU"));
        countriesMap.put("US", new Country("USA", "US"));
        countriesMap.put("UK", new Country("United Kingdom",
                                           "UK"));
    }

    @Bean                                                             (4)
    Flight buildFlightFromCsv() throws IOException {                  (5)
        Flight flight = new Flight("AA1234", 20);
        try(BufferedReader reader = new BufferedReader(               (6)
         new FileReader(
          "src/test/resources/flights_information.csv"))) {
            String line = null;                                       (7)
            do {
                line = reader.readLine();
                if (line != null) {
                    String[] passengerString =                        (9)
                           line.toString().split(";");
                    Passenger passenger = new                         (10)
                           Passenger(passengerString[0].trim());
(8)                 passenger.setCountry(                             (11)
                         countriesMap.get(
                               passengerString[1].trim()));
                    passenger.setIsRegistered(false);                 (12)
                    flight.addPassenger(passenger);                   (13)
                }
            } while (line != null);

        }                                                             (14)

        return flight;
    }
}
```

In this listing:

- Mike creates 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` (1). Thus, it signals that it defines the beans needed for a test.
- He defines the countries map (2) and populates it with three countries (3). The key of the map is the country code, and the value is its name.
- He creates the `buildFlightFromCsv` method and annotates it with `@Bean` (4). Its purpose is to create and configure a `Flight` bean that will be injected into the tests.
- He creates a flight with the help of the constructor (5) and then constructs it by parsing the CSV file (6).
- He initializes the line variable with `null` (7) and then parses the CSV file and reads it line by line (8).
- The line is split by the `;` separator (9). Mike creates a passenger with the help of the constructor, which has as an argument the part of the line before the separator (10). He sets the passenger's country by taking from the countries map the value corresponding to the country code included in the part of the line after the separator (11).
- He sets the passenger as not registered (12), adds them to the flight (13), and, after he finishes parsing all the lines from the CSV file, returns the fully configured flight (14).

Mike will finally implement the `FlightTest` class, which registers all the passengers from a flight and checks that all of them are confirmed.

Listing 17.8 `FlightTest` class

```
[...]
@SpringBootTest
@Import(FlightBuilder.class)                                    (1)
public class FlightTest {

    @Autowired
    private Flight flight;                                      (2)

    @Autowired
    private RegistrationManager registrationManager;            (3)

    @Test
    void testFlightPassengersRegistration() {                   (4)
        for (Passenger passenger : flight.getPassengers()) {    (5)
            assertFalse(passenger.isRegistered());              (6)
            registrationManager.getApplicationContext().
                publishEvent(
                    new PassengerRegistrationEvent(passenger)); (7)
        }
```

```
            for (Passenger passenger : flight.getPassengers()) {
                assertTrue(passenger.isRegistered());
            }
        }
    }
```

In this listing:

- Mike creates the `FlightTest` class, annotates it with `@SpringBootTest`, and imports the beans from the `FlightBuilder` class (1). Remember that `@SpringBootTest` searches in the current package of the test class and in its subpackages for bean definitions. This way, it will be able to discover and autowire the `RegistrationManager` bean (3).
- He autowires the `Flight` bean injected by the `FlightBuilder` class (2).
- He creates the `testFlightPassengersRegistration` method and annotates it with `@Test` (4). In it, he browses all passengers from the injected `Flight` bean (5) and first checks that they are not registered (6). Then he uses the `RegistrationManager` field to publish an event of type `PassengerRegistrationEvent` with the help of the reference to the application context that it is holding (7).
- Finally, he browses all passengers from the injected `Flight` bean and checks that they are now registered (8).

Running `FlightTest` is successful, as shown in figure 17.8.

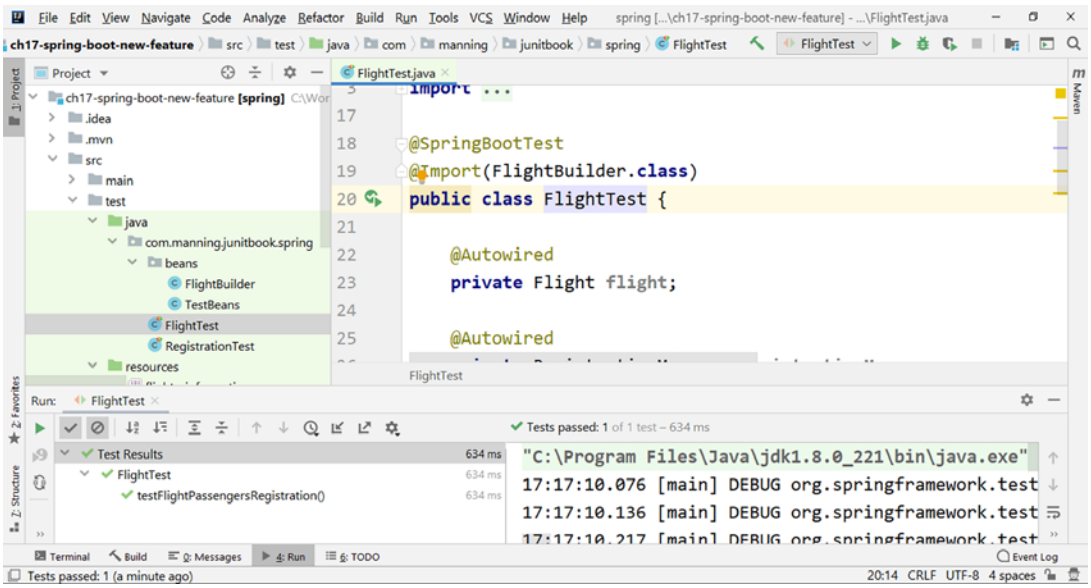


Figure 17.8 Successfully running `FlightTest`, which checks the registration of all passengers on a flight

In the next chapter, we will start building and testing a representational state transfer (REST) application with the help of Spring Boot.

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, a web application that helps to generate a Spring Boot project's structure.
- Moving our previously developed Spring application, which we tested with JUnit 5, to the skeleton provided by Spring Boot.
- Implementing a test-specific configuration for Spring Boot with the help of the `@TestConfiguration` annotation and Java-based beans, and taking advantage of the Spring Boot convention-over-configuration approach.
- Developing a new feature to create and set up flights and add and remove passengers, integrating it into a Spring Boot application, and testing it with JUnit 5.

18 Testing a REST API

This chapter covers

- Creating a RESTful API to manage one or more entities
- Testing a RESTful API

> *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*

Representational state transfer (REST) is a software architectural style for creating web services; it also provides a set of constraints. The American computer scientist Roy Fielding first defined REST, presenting the REST principles in his PhD dissertation in 2000. (Fielding is also one of the authors of the HTTP specification.) Web services following this REST architectural style are called *RESTful web services*. RESTful web services allow interoperability between the internet and computer systems. Requesting systems can access and manipulate web resources represented as text using a well-known set of stateless operations. We'll study them in more detail in this chapter.

18.1 Introducing REST applications

We will first define the terms *client* and *resource* in order to shape what makes an API RESTful. A *client* may be a person or software using the RESTful API. For example, a programmer using a RESTful API to execute actions against the LinkedIn website is such a client. The client can also be a web browser. When you go to the LinkedIn website, your browser is the client that calls the website API and displays the obtained information on the screen.

A *resource* can be any object about which the API can obtain information. In the 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*—Clients are separated from servers, and each has its own concern. Most frequently, a client is concerned with the representation of the user, and 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 is interacting directly with the server or with an intermediary. Layers can be dynamically added and removed. They can provide security, load balancing, and shared caching.
- *Cacheable*—Clients can cache responses. Responses define themselves as cacheable or not.
- *Code on demand (optional)*—Servers can 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 and Java applets.

A RESTful web application provides information about its resources. Resources are identified with the help of URLs. The client can execute actions against such a resource: create, read, update, or delete the 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 an API RESTful, you have to follow a set of rules while developing it. A RESTful API will transfer information to the client, which uses it as a representation of the state of the accessed resource. For example, when you call the LinkedIn API to access a specific user, the API will return the state of that user (name, biography, professional experience, posts). The REST rules make the API easier to understand and simpler for new programmers to use when they join a team.

The representation of the state can be in JSON, XML, or HTML format. The client uses the API to send the following to the server:

- The identifier (URL) of the resource you want to access.
- The operation you 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, using the LinkedIn RESTful API to fetch a specific LinkedIn user requires that you have a URL that identifies the user and that you use the HTTP method `GET`.

18.2 Creating a RESTful API to manage one entity

The flight-management application developed at Tested Data Systems, as we left it at the end of chapter 17, allows us to register a list of passengers for a flight and test 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 a flight's passengers.

The first thing Mike will do is to create the REST API that will deal with countries. For that purpose, Mike will add new dependencies to the Maven pom.xml configuration file.

Listing 18.1 Newly added dependencies in the pom.xml configuration file

```
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
   <groupId>com.h2database</groupId>
   <artifactId>h2</artifactId>
</dependency>
```

(1) (2) (3)

In this listing:

- Mike adds the `spring-boot-starter-web` dependency for building web (including RESTful) applications using Spring Boot (1).
- Because he needs to use Spring and the Java Persistence API (JPA) for persisting information, he includes the `spring-boot-starter-data-jpa` dependency (2).
- H2 is an open source, lightweight Java database that Mike will embed in the Java application and persist information to it. Consequently, he adds the `h2` dependency (3).

As the application will be a functional RESTful one that can be used to access, create, modify, and delete information about the list of passengers, Mike moves the `FlightBuilder` class from the test folder into the main folder. He also extends its functionality to provide access to managing countries.

Listing 18.2 `FlightBuilder` class

```
[...]
public class FlightBuilder {

    private Map<String, Country> countriesMap = new HashMap<>();

    public FlightBuilder() throws IOException {
       try(BufferedReader reader = new BufferedReader(new
           FileReader("src/main/resources/countries_information.csv"))) {
        String line = null;
        do {
            line = reader.readLine();
            if (line != null) {
                String[] countriesString = line.toString().split(";");
                Country country = new Country(countriesString[0].trim(),
                                              countriesString[1].trim());
                countriesMap.put(countriesString[1].trim(), country);
            }
        } while (line != null);
      }
    }
    @Bean
    Map<String, Country> getCountriesMap() {
        return Collections.unmodifiableMap(countriesMap);
    }

    @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);

        }
```

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19

```
        return flight;                    ⟵ 20
    }
}
```

In this listing:

- Mike defines the countries map 1.
- He populates the countries map in the constructor with three countries 2 by parsing the CSV file 3.
- He initializes the `line` variable with `null` 4, parses the CSV file, and reads it line by line 5.
- The line is split with the `;` separator 6. Mike creates a country with the help of the constructor, which has as an argument the part of the line before the separator 7.
- He adds the parsed information to `countriesMap` 8.
- He creates the `getCountriesMap` method and annotates it with `@Bean` 9. Its purpose is to create and configure a `Map` bean that will be injected into the application.
- He creates the `buildFlightFromCsv` method and annotates it with `@Bean` 10. Its purpose is to create and configure a `Flight` bean that will be injected into the application.
- He creates a flight with the help of the constructor 11 by parsing the CSV file 12.
- He initializes the `line` variable with `null` 13, parses the CSV file, and reads it line by line 14.
- The line is split with the `;` separator 15. Mike creates a passenger with the help of the constructor, which has as an argument the part of the line before the separator 16. He sets the passenger's country by taking from the countries map the value corresponding to the country code included in the part of the line after the separator 17.
- He sets the passenger as not registered 18 and adds them to the flight 19. After all the lines from the CSV file are parsed, the fully configured flight is returned 20.

The content of the countries_information.csv file used for building the countries map is shown next.

Listing 18.3 countries_information.csv file

```
Australia; AU
USA; US
United Kingdom; UK
```

Mike would like to start the RESTful application on the custom 8081 port to avoid possible port conflicts. Spring Boot allows the externalization of the configuration so

that different people can work with the same application code in different environments. Various properties can be specified in the application.properties file; in this case, Mike only needs to set `server.port` as `8081`.

Listing 18.4 application.properties file

```
server.port=8081
```

Mike modifies the `Country` class to make it a model component of the RESTful application (see listing 18.5). The model is one of the components of the model-view-controller (MVC) pattern: it is the application's dynamic data structure, independent of the user interface. It directly manages the application's data.

> **DEFINITION** *Model-view-controller (MVC)*—A software design pattern used for creating programs that access data through user interfaces. The program is split into three related parts: *model*, *view*, and *controller*. Thus the inner representation of information is separated from its presentation on the user side, and the system's parts are loosely coupled.

Listing 18.5 Modified `Country` class

```
[...]
@Entity                                    ❶
public class Country {

    @Id                                    ❷
    private String codeName;
    private String name;


    // avoid "No default constructor for entity"   ❸
    public Country() {
    }
    [...]
}
```

In this listing:

- Mike annotates the `Country` class with `@Entity` so it can represent objects that can be persisted ❶.
- He annotates the `codeName` field with `@Id` ❷. This means the `codeName` field is a primary key: it uniquely identifies an entity that is persisted.
- He adds a default constructor to the `Country` class ❸. 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 compiler no longer provides the default constructor because Mike created one.

Next, Mike creates the `CountryRepository` interface, which extends `JpaRepository`.

Listing 18.6 `CountryRepository` interface

```
public interface CountryRepository extends JpaRepository<Country, Long> {
}
```

Defining this interface serves two purposes. First, by extending `JpaRepository`, Mike gets a bunch of generic CRUD (create, read, update, and delete) methods in the `CountryRepository` type that make it possible to work with `Country` objects. Second, this allows Spring to scan the classpath for this interface and create a Spring bean for it.

> **DEFINITION** *CRUD (create, read, update, delete)*—The four basic functions of persistence. In addition to REST, they are often used in database applications and user interfaces.

Mike now writes a controller for `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 `CountryController` class

```
@RestController                                          (1)
public class CountryController {

    @Autowired                                           (2)
    private CountryRepository repository;

    @GetMapping("/countries")                            (3)
    List<Country> findAll() {
        return repository.findAll();                     (4)
    }

}
```

In this listing:

- Mike creates the `CountryController` class and annotates it with `@RestController` (1). The `@RestController` annotation was introduced in Spring 4.0 to simplify creating RESTful web services. It marks the class as a controller and also eliminates the need to annotate every one of its request-handling methods with the `@ResponseBody` annotation (as required before Spring 4.0).
- He declares a `CountryRepository` field and autowires it (2). As `CountryRepository` extends `JpaRepository`, Spring will scan the classpath for this interface, create a Spring bean for it, and autowire it here.
- He creates the `findAll` method and annotates it with `@GetMapping("/countries")` (3). The `@GetMapping` annotation maps HTTP `GET` requests to the `/countries` URL onto the specific handler method. Because Mike uses the `@RestController` annotation on the class itself, he does not need to annotate

the response object of the method with `@ResponseBody`. As mentioned earlier, that was necessary before Spring 4.0, when we also had to use `@Controller` instead of `@RestController` on the class.

- The `findAll` method returns the result of executing `repository.findAll()` (4). `repository.findAll()` is a CRUD method that is automatically generated because `CountryRepository` extends `JpaRepository`. As its name says, it will return all objects from the repository.

Mike now revises the `Application` class, previously created by Spring Boot (see chapter 17).

Listing 18.8 Modified `Application` class

```
[...]
@SpringBootApplication
@Import(FlightBuilder.class)                                    (1)
public class Application {

   @Autowired
   private Map<String, Country> countriesMap;                   (2)

   public static void main(String[] args) {
      SpringApplication.run(Application.class, args);
   }

   @Bean
   CommandLineRunner configureRepository                        (3)
                     (CountryRepository countryRepository) {
      return args -> {
         for (Country country: countriesMap.values()) {         (4)
            countryRepository.save(country);
         }
      };
   }

}
```

In this listing:

- Mike imports `FlightBuilder`, which created the `countriesMap` bean (1), and autowires it (2).
- He creates a bean of type `CommandLineRunner` (3). `CommandLineRunner` is a Spring Boot functional interface (an interface with a single method) that gives access to application arguments as a string array. The created bean browses all the countries in `countriesMap` and saves them into `countryRepository` (4). This `CommandLineRunner` interface is created, and its single method is executed, just before the `run()` method from `SpringApplication` completes.

Mike now executes `Application`. So far, the RESTful application only provides 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 this REST API endpoint using the `curl` program. `curl` (which stands for *client URL*) is a command-line tool that transfers data using various protocols, including HTTP. We simply execute the following command

```
curl -v localhost:8081/countries
```

because the application is running on port 8081 and /countries is the only available endpoint. The result is shown in figure 18.1—the command lists the countries in JSON format.

Figure 18.1 The result of running the `curl -v localhost:8081/countries` command is the list of created countries.

We can also test accessing this endpoint through the browser by going to localhost:8081/countries. Again, the result is provided in JSON format, as shown in figure 18.2.

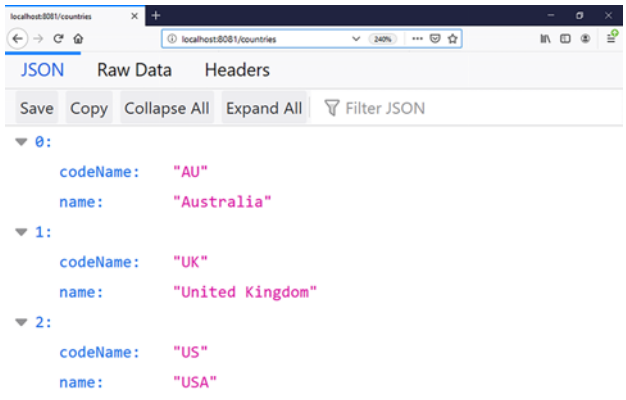


Figure 18.2 Accessing the localhost:8081/countries URL from the browser displays the list of created countries.

18.3 Creating a RESTful API to manage two related entities

Mike continues the implementation by modifying the `Passenger` class to make it a model component of the RESTful application.

Listing 18.9 Modified `Passenger` class

```
@Entity                                          (1)
public class Passenger {

    @Id                                          (2)
    @GeneratedValue
    private Long id;
    private String name;
    @ManyToOne                                   (3)
    private Country country;
    private boolean isRegistered;

    // avoid "No default constructor for entity"
    public Passenger() {                         (4)
    }
    [...]
}
```

In this listing:

- Mike annotates the `Passenger` class with `@Entity` so it has the ability to represent objects that can be persisted (1).
- He introduces a new `Long` field, `id`, and annotates it with `@Id` and `@GeneratedValue` (2). This means the `id` field is a primary key, and its value will be automatically generated for that field by the persistence layer.
- He annotates the `country` field with `@ManyToOne` (3). This means many passengers can be mapped to a single country.
- He adds a default constructor to the `Passenger` class (4). 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 next creates the `PassengerRepository` interface, which extends `JpaRepository`.

Listing 18.10 `PassengerRepository` interface

```
public interface PassengerRepository extends JpaRepository<Country, Long> {
}
```

Defining this interface serves two purposes. First, extending `JpaRepository` gives the `PassengerRepository` generic CRUD methods that let Mike work with `Passenger` objects. Second, this allows Spring to scan the classpath for this interface and create a Spring bean for it.

Mike now writes a custom exception to be thrown when a passenger is not found.

Listing 18.11 `PassengerNotFoundException` class

```
public class PassengerNotFoundException extends RuntimeException {

    public PassengerNotFoundException(Long id) {
        super("Passenger id not found : " + id);
    }

}
```

In this listing, Mike declares `PassengerNotFoundException` by extending `RuntimeException` (1) and creates a constructor that receives an `id` as a parameter (2).

Next, Mike writes a controller for `PassengerRepository`.

Listing 18.12 `PassengerController` class

```
[...]
@RestController
public class PassengerController {

    @Autowired
    private PassengerRepository repository;

    @Autowired
    private Map<String, Country> countriesMap;

    @GetMapping("/passengers")
    List<Passenger> findAll() {
        return repository.findAll();
    }

    @PostMapping("/passengers")
    @ResponseStatus(HttpStatus.CREATED)
    Passenger createPassenger(@RequestBody Passenger passenger) {
        return repository.save(passenger);
    }

    @GetMapping("/passengers/{id}")
    Passenger findPassenger(@PathVariable Long id) {
        return repository.findById(id)
                .orElseThrow(() -> new PassengerNotFoundException(id));
    }

    @PatchMapping("/passengers/{id}")
    Passenger patchPassenger(@RequestBody Map<String, String> updates,
                  @PathVariable Long id) {

        return repository.findById(id)
          .map(passenger -> {
```

(1) (2) (3) (4) (5) (6) (7) (8) (9) (10) (11) (12)

```
                String name = updates.get("name");
                if (null!= name) {
                    passenger.setName(name);
                }

                Country country =
                    countriesMap.get(updates.get("country"));
                if (null != country) {
                    passenger.setCountry(country);
                }

                String isRegistered = updates.get("isRegistered");
                if(null != isRegistered) {
                  passenger.setIsRegistered(
                     isRegistered.equalsIgnoreCase("true")? true: false);
                }
                return repository.save(passenger);
            })
               .orElseGet(() -> {
                   throw new PassengerNotFoundException(id);
            });

    }

    @DeleteMapping("/passengers/{id}")
    void deletePassenger(@PathVariable Long id) {
        repository.deleteById(id);
    }
}
```

13 14 15 16 17 18 19

In this listing:

- Mike creates the `PassengerController` class and annotates it with `@RestController` 1.
- He declares a `PassengerRepository` field and autowires it 2. Because `PassengerRepository` extends `JpaRepository`, Spring will scan the classpath for this interface, create a Spring bean for it, and autowire it here. Mike also declares a `countriesMap` field and autowires it 3.
- He creates the `findAll` method and annotates it with `@GetMapping("/passengers")` 4. This `@GetMapping` annotation maps HTTP `GET` requests to the /passengers URL onto the specific handler method.
- He declares the `createPassenger` method and annotates it with `@PostMapping("/passengers")` 5. `@PostMapping`-annotated methods handle the HTTP `POST` requests matched with given URL expressions. Mike marks that method with `@ResponseStatus`, specifying the response status as `HttpStatus.CREATED` 6. The `@RequestBody` annotation maps the `HttpRequest` body to the annotated domain object. The `HttpRequest` body is deserialized to a Java object 7.
- Mike declares the `findPassenger` method that will look for the passenger by ID and annotates it with `@GetMapping("/passengers/{id}")` 8. The

`@GetMapping` annotation maps HTTP `GET` requests to the /passengers/{id} URL onto the specific handler method. It searches for the passenger in the repository and returns them; or, if the passenger does not exist, it throws the custom declared `PassengerNotFoundException`. The `id` argument of the method is annotated with `@PathVariable`, meaning it will be extracted from the URL's {`id`} value ⑨.

- Mike declares the `patchPassenger` method and annotates it with `@PatchMapping("/passengers/{id}")` ⑩. The `@PatchMapping` annotation maps HTTP `PATCH` requests to the /passengers/{id} URL onto the specific handler method. He annotates the `updates` parameter with `@RequestBody` and the `id` parameter with `@PathVariable` ⑪. 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 URL's {`id`} value.
- Mike searches the repository by the input `id` ⑫. He changes the name of an existing passenger ⑬, the country ⑭, and the registration status ⑮. The modified passenger is saved in the repository ⑯.
- If the passenger does not exist, the custom declared `PassengerNotFoundException` is thrown ⑰.
- Mike declares the `delete` method and annotates it with `@DeleteMapping("/passengers/{id}")` ⑱. The `@DeleteMapping` annotation maps HTTP `DELETE` requests to the /passengers/{id} URL onto the specific handler method. It deletes the passenger from the repository. The `id` argument of the method is annotated with `@PathVariable`, meaning it will be extracted from the URL's {`id`} value ⑲.

Mike now modifies the `Application` class.

Listing 18.13 Modified `Application` class

```
@SpringBootApplication
@Import(FlightBuilder.class)
public class Application {

   @Autowired
   private Flight flight;                                    ①

   @Autowired
   private Map<String, Country> countriesMap;

   public static void main(String[] args) {
      SpringApplication.run(Application.class, args);
   }

   @Bean
   CommandLineRunner configureRepository                                    ②
                    (CountryRepository countryRepository,
                     PassengerRepository passengerRepository) {
      return args -> {
```

```
            for (Country country: countriesMap.values()) {
                countryRepository.save(country);
            }

            for (Passenger passenger : flight.getPassengers()) {
                passengerRepository.save(passenger);
            }
        };
    }

}
```

In this listing:

- Mike autowires the `flight` bean (1) imported from the `FlightBuilder`.
- He modifies the bean of type `CommandLineRunner` by adding a new parameter of type `PassengerRepository` to the `configureRepository` method that creates it (2). `CommandLineRunner` is a Spring Boot interface that provides access to application arguments as a string array. The created bean will additionally browse all passengers in the `flight` and save them in `passengerRepository` (3). This `CommandLineRunner` interface is created, and its method is executed, just before the `run()` method from `SpringApplication` completes.

The list of passengers of the flight is shown in the following CSV file. A passenger is described using a name and a country code; there are 20 passengers total.

Listing 18.14 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 starts, the `FlightBuilder` class parses the file, creates the flight with the list of passengers, and injects the flight into the application. The application browses the list and saves each passenger in the repository.

Mike launches `Application` into execution. The RESTful application now also provides access to the /passengers endpoint. We can test the new functionalities of the REST API endpoint using the `curl` program:

```
curl -v localhost:8081/passengers
```

The application is running on port 8081, and /passengers is available as an endpoint. The result is shown in figure 18.3: a list of passengers in JSON format.

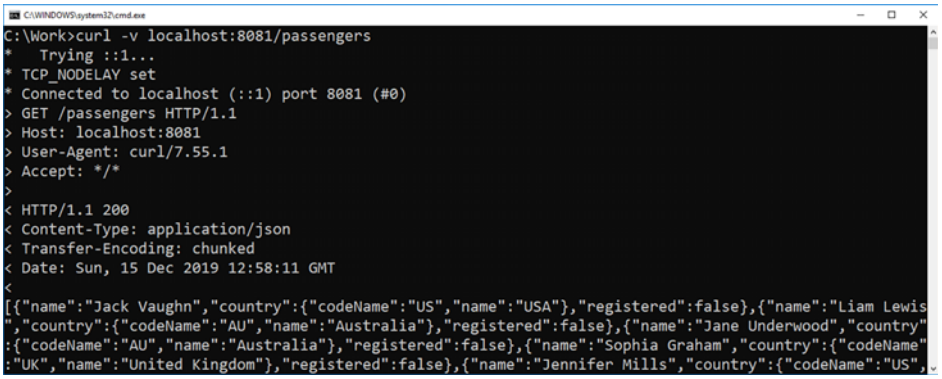


Figure 18.3 The result of running the `curl -v localhost:8081/passengers` command is the list of passengers.

We can also test the other functionalities implemented for the /passengers endpoint. For example, to get the passenger having ID 4, we execute this command:

```
curl -v localhost:8081/passengers/4
```

The result is shown in figure 18.4: the passenger information is provided in JSON format.

Figure 18.4 Running the `curl -v localhost:8081/passengers/4` command shows the passenger with ID 4.

We can update the name, country, and registered status of the passenger with ID 4 by executing this command:

```
curl -v -X PATCH localhost:8081/passengers/4
-H "Content-type:application/json"
-d "{\"name\":\"Sophia Jones\", \"country\":\"AU\",
     \"isRegistered\":\"true\"}"
```

The result is shown in figure 18.5.

Figure 18.5 The result of successfully updating the information for the passenger with ID 4

To delete this passenger, we use this command:

```
curl -v -X DELETE localhost:8081/passengers/4
```

The result is shown in figure 18.6.

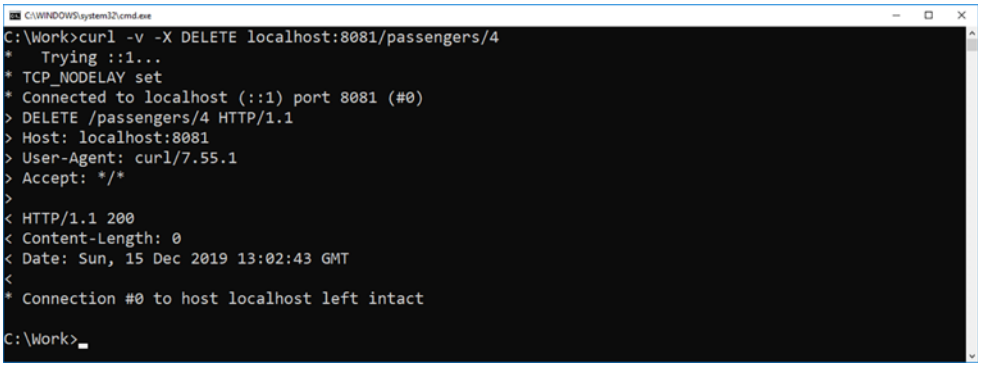


Figure 18.6 The passenger with ID 4 has been deleted.

Finally, we can post a new passenger:

```
curl -v -X POST localhost:8081/passengers
-H "Content-type:application/json"
-d "{\"name\":\"John Smith\"}"
```

The result is shown in figure 18.7.

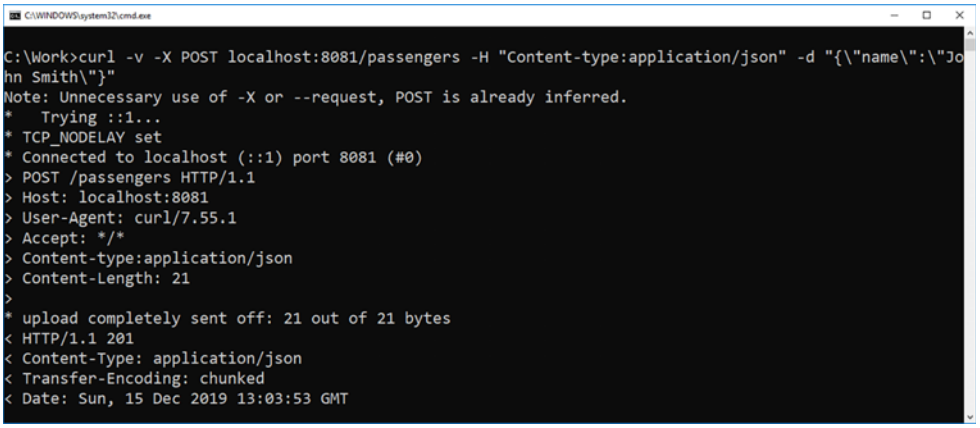


Figure 18.7 We successfully posted a new passenger, John Smith.

18.4 Testing the RESTful API

Next, Mike writes the tests to automatically verify the behavior of the RESTful API.

Listing 18.15 `RestApplicationTest` class

```
[...]
@SpringBootTest
@AutoConfigureMockMvc
@Import(FlightBuilder.class)
public class RestApplicationTest {

    @Autowired
    private MockMvc mvc;

    @Autowired
    private Flight flight;

    @Autowired
    private Map<String, Country> countriesMap;

    @MockBean
    private PassengerRepository passengerRepository;

    @MockBean
    private CountryRepository countryRepository;
```

1
2
3
4
5
6

```
@Test
void testGetAllCountries() throws Exception {
    when(countryRepository.findAll()).thenReturn(new
           ArrayList<>(countriesMap.values()));
    mvc.perform(get("/countries"))
       .andExpect(status().isOk())
       .andExpect(content().contentType(MediaType.APPLICATION_JSON))
       .andExpect(jsonPath("$", hasSize(3)));

    verify(countryRepository, times(1)).findAll();
}

@Test
void testGetAllPassengers() throws Exception {
    when(passengerRepository.findAll()).thenReturn(new
         ArrayList<>(flight.getPassengers()));

    mvc.perform(get("/passengers"))
       .andExpect(status().isOk())
       .andExpect(content().contentType(MediaType.APPLICATION_JSON))
       .andExpect(jsonPath("$", hasSize(20)));

    verify(passengerRepository, times(1)).findAll();
}

@Test
void testPassengerNotFound() {
    Throwable throwable = assertThrows(NestedServletException.class,
              () -> mvc.perform(get("/passengers/30"))
                           .andExpect(status().isNotFound()));
    assertEquals(PassengerNotFoundException.class,
                 throwable.getCause().getClass());
}

@Test
void testPostPassenger() throws Exception {

    Passenger passenger = new Passenger("Peter Michelsen");
    passenger.setCountry(countriesMap.get("US"));
    passenger.setIsRegistered(false);
    when(passengerRepository.save(passenger))
        .thenReturn(passenger);

    mvc.perform(post("/passengers")
        .content(new ObjectMapper().writeValueAsString(passenger))
        .header(HttpHeaders.CONTENT_TYPE, MediaType.APPLICATION_JSON))
       .andExpect(status().isCreated())
       .andExpect(jsonPath("$.name", is("Peter Michelsen")))
       .andExpect(jsonPath("$.country.codeName", is("US")))
       .andExpect(jsonPath("$.country.name", is("USA")))
       .andExpect(jsonPath("$.registered", is(Boolean.FALSE)));

 verify(passengerRepository, times(1)).save(passenger);
}
```

7 8 9 10 11 12 13 14 15 16 17 18

```
    @Test
    void testPatchPassenger() throws Exception {
        Passenger passenger = new Passenger("Sophia Graham");
        passenger.setCountry(countriesMap.get("UK"));
        passenger.setIsRegistered(false);
        when(passengerRepository.findById(1L))
             .thenReturn(Optional.of(passenger));
        when(passengerRepository.save(passenger))
             .thenReturn(passenger);
        String updates =
          "{\"name\":\"Sophia Jones\", \"country\":\"AU\",
           \"isRegistered\":\"true\"}";

        mvc.perform(patch("/passengers/1")
            .content(updates)
            .header(HttpHeaders.CONTENT_TYPE,
                                MediaType.APPLICATION_JSON))
            .andExpect(content().contentType(MediaType.APPLICATION_JSON))
            .andExpect(status().isOk());

        verify(passengerRepository, times(1)).findById(1L);
        verify(passengerRepository, times(1)).save(passenger);
    }

    @Test
    public void testDeletePassenger() throws Exception {

        mvc.perform(delete("/passengers/4"))
               .andExpect(status().isOk());

        verify(passengerRepository, times(1)).deleteById(4L);
    }
}
```

In this listing:

- Mike creates the `RestApplicationTest` class and annotates it with `@SpringBootTest` (1). `@SpringBootTest` searches the current package of the test class and its subpackages for bean definitions.
- He also annotates the class with `@AutoConfigureMockMvc` in order to enable all autoconfiguration related to the `MockMvc` objects used in the test (2).
- He imports `FlightBuilder`, which creates a flight bean and a countries map bean (3).
- He autowires a `MockMvc` object (4). `MockMvc` is the main entry point for server-side Spring REST application testing: Mike will perform a series of REST operations against this `MockMvc` object during the tests.
- He declares a flight and a `countriesMap` field and autowires them (5). These fields are injected from the `FlightBuilder` class.
- He declares `countryRepository` and `passengerRepository` fields and annotates them with `@MockBean` (6). `@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. Mike will provide instructions for the behavior of the mock objects during the tests.

- In the `testGetAllCountries` test, he instructs the mock `countryRepository` bean to return the array of values from `countriesMap` when the `findAll` method is executed on it (7).
- Mike simulates the execution of the `GET` method on the `/countries` URL (8) and verifies the returned status, expected content type, and returned JSON size (9). He also verifies that the method `findAll` has been executed exactly once on the `countryRepository` bean (10).
- In the `testGetAllPassengers` test, he instructs the mock `passengerRepository` bean to return the passengers from the `flight` bean when the `findAll` method is executed on it (11).
- He simulates the execution of the `GET` method on the `/passengers` URL and verifies the returned status, expected content type, and returned JSON size (12). He also verifies that the method `findAll` has been executed exactly once on the `passengerRepository` bean (13).
- In `testPassengerNotFound`, Mike tries to get the passenger having ID 30 and checks that a `NestedServletException` is thrown and that the returned status is "Not Found" (14). He also checks that the cause of the `NestedServletException` is `PassengerNotFoundException` (15).
- In `testPostPassenger`, Mike creates a `passenger` object, configures it, and instructs `passengerRepository` to return that object when a `save` is executed on that passenger (16).
- He simulates the execution of the `POST` method on the `/passengers` URL and verifies that the content consists of the JSON string value of the `passenger` object, the header type, the returned status, and the content of the JSON (17). He uses 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.
- Mike also verifies that the `save` method has been executed exactly once on the previously defined passenger (18).
- In `testPatchPassenger`, he creates a `passenger` object, configures it, and instructs `passengerRepository` to return that object when a passenger `findById` is executed with the argument `1` (19). When the `save` method is executed on `passengerRepository`, that `passenger` is returned as well (20).
- Mike sets a JSON object named `updates`, performs a `PATCH` on the `/passengers/1` URL using that update, and checks the content and the returned status (21).
- He verifies that the `findById` and `save` methods have been executed exactly once on `passengerRepository` (22).

- He performs a `DELETE` operation on the /passengers/4 URL, verifies that the returned status is OK (23), and verifies that the `deleteById` method has been executed exactly once (24).

Running `RestApplicationTest` is successful, as shown in figure 18.8.

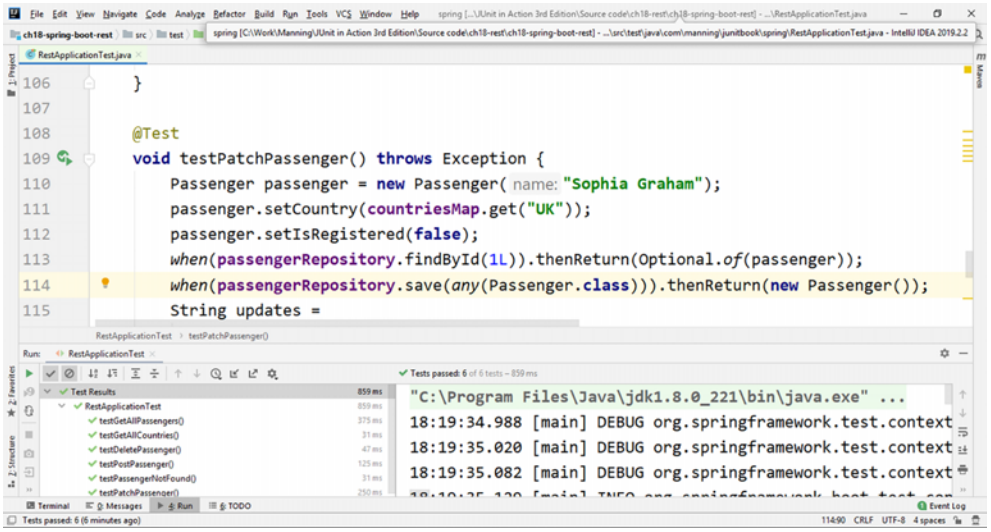


Figure 18.8 Successfully running `RestApplicationTest`, which checks the RESTful application's functionality

The next chapter will be dedicated to testing database applications and the various alternatives for doing so.

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, and code on demand
- Creating a RESTful API to manage a single entity—the country from the flight-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 the passenger—and executing `GET`, `PATCH`, `DELETE`, and `POST` operations against it to get the list of passengers; to get a particular passenger by ID; and to create, update, or delete a passenger
- Testing the RESTful API that manages two related entities 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 JDBC, Spring JDBC, Hibernate, and Spring Hibernate applications
- Comparing the different approaches to 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, 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 due to 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 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). We'll discuss the issues in more detail in this chapter and show possible implementation and testing alternatives for Java database applications.

19.1 The database unit testing impedance mismatch

Let's take a deeper look at the three issues that make up the database unit testing impedance mismatch.

19.1.1 Unit tests must exercise code in isolation

From a purist's point of view, tests that exercise database access code cannot be considered unit tests because 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 that 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 data access object (DAO) 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 that 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 to access the database. In these tests, the persistence layer can be emulated by mocks or stubs. Like 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 in the database get in the way of the tests? Yes, it is possible, so before we run the tests, we must ensure that 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—we have to write SQL statements, mix many levels of `try/catch/finally` code, convert SQL types to and from Java, and so on.

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. Then, we'll introduce Spring as the framework used for our application. Finally, we'll move to ORM and Hibernate.

DEFINITIONS *Java Database Connectivity (JDBC)*—A Java API that defines how a client can access a database. JDBC provides methods to query and update data in a relational database.

Object-relational mapping (ORM)—A programming technique for converting data between relational databases and object-oriented programming languages and vice versa.

Hibernate—An ORM 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 tests. Everything seems fine, 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 10 minutes for the build to complete, 9 of them spent in the database tests. What should you do then?

This is the hardest issue because it cannot always be 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 is to move the database closer to the developer, by either using an embedded database (if the application uses standard SQL that enables a database switch or uses an ORM framework) or locally installing lighter versions of the database.

DEFINITION *Embedded database*—A database that is bundled within an application instead of being managed by external servers (which is the typical scenario). A broad range of embedded databases are available for Java applications, most of them based on open source projects like H2 (https://h2database.com), HSQLDB (http://hsqldb.org), and Apache Derby (https://db.apache.org/derby). The fundamental characteristic of an embedded database is that it is managed by the application, not the language it is written in. For instance, both HSQLDB and Derby support client/server mode (in addition to the embedded option), while SQLite (which is a C-based product) could also be embedded in a Java application.

In the following sections, we will see how to start with a pure JDBC application and an embedded database. Then we will introduce Spring and Hibernate as ORM frameworks and also take steps to solve the database unit testing impedance mismatch.

19.2 Testing a JDBC application

JDBC is a Java API that defines how a client can access a database: it 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). Because it was one of the early APIs used in Java and was designed to be regularly used in database applications, it may still be encountered in projects and may not even be combined with any other technology.

In our example, we'll start from a pure JDBC application, then introduce Spring and Hibernate, and finally test all of these applications. This process will demonstrate how such database applications can be tested and also show how to reduce the database unit testing impedance mismatch.

At Tested Data Systems, the flight-management application we have seen in previous chapters persists information into a database. It is George's job to analyze the application and move it to present-day technologies. The JDBC application that George receives contains a `Country` class describing passengers' countries.

Listing 19.1 `Country` class

```
public class Country {
    private String name;
    private String codeName;

    public Country(String name, String codeName) {
        this.name = name;
        this.codeName = codeName;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getCodeName() {
        return codeName;
    }

    public void setCodeName(String codeName) {
        this.codeName = codeName;
    }

    @Override
    public String toString() {
        return "Country{" +
                "name='" + name + '\'' +
                ", codeName='" + codeName + '\'' +
                '}';
    }
```

1
2
1
1
3

```
    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Country country = (Country) o;
        return Objects.equals(name, country.name) &&
                Objects.equals(codeName, country.codeName);
    }

    @Override
    public int hashCode() {
        return Objects.hash(name, codeName);
    }
}
```

In this listing:

- George declares the `name` and `codeName` fields of the `Country` class, together with the corresponding getters and setters (1).
- He creates a constructor of the `Country` class to initialize the `name` and `codeName` fields (2).
- He overrides the `toString` method to display a country (3).
- He overrides the `equals` and `hashCode` methods to take into account the `name` and `codeName` fields (4).

The application currently uses the embedded H2 database for testing purposes. The Maven pom.xml file includes the JUnit 5 dependencies and H2 dependencies.

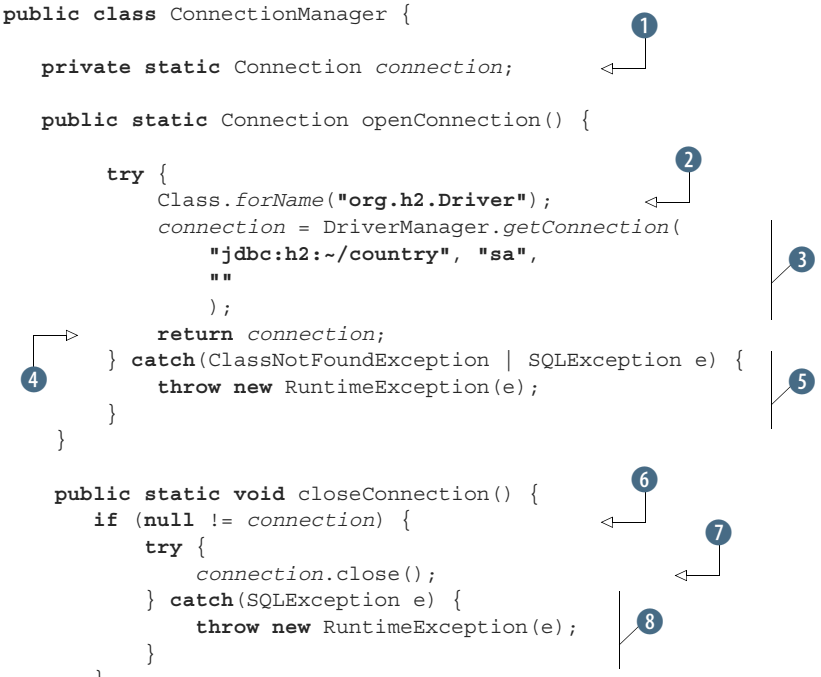
Listing 19.2 Maven pom.xml dependencies

```
<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>1.4.199</version>
   </dependency>
</dependencies>
```

The application manages connections to the database and operations against the database tables through the `ConnectionManager` and `TablesManager` classes (listings 19.3 and 19.4).

Listing 19.3 `ConnectionManager` class

```
[...]
public class ConnectionManager {

   private static Connection connection;

   public static Connection openConnection() {

        try {
            Class.forName("org.h2.Driver");
            connection = DriverManager.getConnection(
                "jdbc:h2:~/country", "sa",
                ""
                );
            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:

- George declares a `Connection` type `connection` field 1.
- In the `openConnection` method, he loads the H2 driver 2 and initializes the previously declared `connection` field to access the H2 `country` database using JDBC, with `sa` as a user and no password 3. If everything goes fine, he returns the initialized `connection` 4.
- If the H2 driver class hasn't been found or the code encounters an `SQLException`, George catches it and rethrows a `RuntimeException` 5.
- In the `closeConnection` method, he first checks to be sure the `connection` is not null 6 and then tries to close it 7. If an `SQLException` occurs, he catches it and rethrows a `RuntimeException` 8.

Listing 19.4 `TablesManager` class

```
[...]
public class TablesManager {

    public static void createTable() {
```

```
        String sql = "CREATE TABLE COUNTRY( ID IDENTITY,
                     NAME VARCHAR(255), CODE_NAME VARCHAR(255) );";
        executeStatement(sql);
    }

    public static void dropTable() {
        String sql = "DROP TABLE IF EXISTS COUNTRY;";
        executeStatement(sql);
    }

    private static void executeStatement(String sql) {
      PreparedStatement statement;

        try {
             Connection connection = openConnection();
             statement = connection.prepareStatement(sql);
             statement.executeUpdate();
             statement.close();
        } catch (SQLException e) {
             throw new RuntimeException(e);
        } finally {
             closeConnection();
        }
    }

}
```

In this listing:

- In the `createTable` method, George declares 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` (1). Then he executes this statement (2).
- In the `dropTable` method, he declares an SQL `DROP TABLE` statement that drops the `COUNTRY` table if it exists (3). Then he executes this statement (4).
- The `executeStatement` method declares a `PreparedStatement` variable (5). Then it opens a connection (6), prepares the statement (7), executes it (8), and closes it (9). If an `SQLException` is caught, George rethrows a `RuntimeException` (10). Regardless of whether the statement is successfully executed, he closes the connection (11).

The application declares a `CountryDao` class, an implementation of the DAO pattern, which provides an abstract interface to the database and executes queries against it.

Listing 19.5 `CountryDao` class

```
[...]
public class CountryDao {
   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 ?";
```

```
    public List<Country> getCountryList() {
        List<Country> countryList = new ArrayList<>();

        try {
            Connection connection = openConnection();
            PreparedStatement statement =
                    connection.prepareStatement(GET_ALL_COUNTRIES_SQL);
            ResultSet resultSet = statement.executeQuery();

            while (resultSet.next()) {
                countryList.add(new Country(resultSet.getString(2),
                                resultSet.getString(3)));
            }
            statement.close();
        } catch (SQLException e) {
            throw new RuntimeException(e);
        } finally {
            closeConnection();
        }
        return countryList;
    }

    public List<Country> getCountryListStartWith(String name) {
        List<Country> countryList = new ArrayList<>();

        try {
            Connection connection = openConnection();
            PreparedStatement statement =
                  connection.prepareStatement(GET_COUNTRIES_BY_NAME_SQL);
            statement.setString(1, name + "%");
            ResultSet resultSet = statement.executeQuery();

            while (resultSet.next()) {
                countryList.add(new Country(resultSet.getString(2),
                                resultSet.getString(3)));
            }
            statement.close();
        } catch (SQLException e) {
            throw new RuntimeException(e);
        } finally {
            closeConnection();
        }
        return countryList;
    }
}
```

In this listing:

- George declares two SQL `SELECT` statements to get all the countries from the `COUNTRY` table (1) and to get the countries whose names match a pattern (2).
- In the `getCountryList` method, he initializes an empty country list (3), opens a connection (4), prepares the statement, and executes it (5).
- He passes through all the results returned from the database and adds them to the list of countries (6). Then he closes the statement (7). If an `SQLException` is

caught, he rethrows a `RuntimeException` ⑧. Regardless of whether the statement is successfully executed, he closes the connection ⑨. He returns the list of countries at the end of the method ⑩.

- In the `getCountryListStartWith` method, George initializes an empty country list ⑪, opens a connection ⑫, prepares the statement, and executes it ⑬.
- He passes through all the results returned from the database and adds them to the list of countries ⑭. Then he closes the statement ⑮. If an `SQLException` is caught, he rethrows a `RuntimeException` ⑯. Regardless of whether the statement is successfully executed, he closes the connection ⑰. He returns the list of countries at the end of the method ⑱.

Moving to the testing side, George has two classes: `CountriesLoader` (listing 19.6), which populates the database and makes sure it is in a known state; and `CountriesDatabaseTest` (listing 19.7), which effectively tests the interaction of the application with the database.

Listing 19.6 `CountriesLoader` class

```
[...]
public class CountriesLoader {

    private static final String LOAD_COUNTRIES_SQL =                    ①
            "insert into country (name, code_name) values ";

    public static final String[][] COUNTRY_INIT_DATA = {                 ②
          { "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) {                 ③
            String sql = LOAD_COUNTRIES_SQL + "('" + countryData[0] +    ④
                         "', '" + countryData[1] + "');";

            try {
                Connection connection = openConnection();                ⑤
                PreparedStatement statement =
                      connection.prepareStatement(sql);                  ⑥
                statement.executeUpdate();
                statement.close();
            } catch (SQLException e) {                                   ⑦
                throw new RuntimeException(e);
            } finally {
                closeConnection();                                       ⑧
            }
        }
    }
}
```

In this listing:

- George declares one SQL `INSERT` statement to insert a country into the `COUNTRY` table (1). He then declares the initialization data for the countries to be inserted (2).
- In the `loadCountries` method, he browses the initialization data for the countries (3) and builds the SQL query that inserts each country (4).
- He opens a connection (5), prepares the statement, executes it, and closes it (6). If an `SQLException` is caught, he rethrows a `RuntimeException` (7). Regardless of whether the statement is successfully executed, he closes the connection (8).

Listing 19.7 `CountriesDatabaseTest` class

```
import static
     com.manning.junitbook.databases.CountriesLoader.COUNTRY_INIT_DATA;
[...]

public class CountriesDatabaseTest {
    private CountryDao countryDao = new CountryDao();
    private CountriesLoader countriesLoader = new CountriesLoader();

    private List<Country> expectedCountryList = new ArrayList<>();
    private List<Country> expectedCountryListStartsWithA =
                          new ArrayList<>();

    @BeforeEach
    public void setUp() {
        TablesManager.createTable();
        initExpectedCountryLists();
        countriesLoader.loadCountries();
    }

    @Test
    public void testCountryList() {
        List<Country> countryList = countryDao.getCountryList();
        assertNotNull(countryList);
        assertEquals(expectedCountryList.size(), countryList.size());
        for (int i = 0; i < expectedCountryList.size(); i++) {
            assertEquals(expectedCountryList.get(i), countryList.get(i));
        }
    }

    @Test
    public void testCountryListStartsWithA() {
        List<Country> countryList =
                      countryDao.getCountryListStartWith("A");
        assertNotNull(countryList);
        assertEquals(expectedCountryListStartsWithA.size(),
                     countryList.size());
```

```
        for (int i = 0; i < expectedCountryListStartsWithA.size();
                        i++) {                                          15
            assertEquals(expectedCountryListStartsWithA.get(i),
                         countryList.get(i));
        }
    }

    @AfterEach                                  16
    public void dropDown() {                    17
        TablesManager.dropTable();
    }

    private void initExpectedCountryLists() {                           18
        for (int i = 0; i < COUNTRY_INIT_DATA.length; i++) {
            String[] countryInitData = COUNTRY_INIT_DATA[i];
            Country country = new Country(countryInitData[0],           19
                                          countryInitData[1]);
            expectedCountryList.add(country);
            if (country.getName().startsWith("A")) {                    21
                expectedCountryListStartsWithA.add(country);
            }
        }
    }
}
```

20

In this listing:

- George statically imports the countries data from `CountriesLoader` and initializes `CountryDao` and `CountriesLoader` (1). He initializes an empty list of expected countries (2) and an empty list of expected countries that start with *A* (3). (He could do this for any letter, but the test is currently looking for the countries with names starting with *A*.)
- He marks the `setUp` method with the `@BeforeEach` annotation so it is executed before each test (4). In it, he creates the empty `COUNTRY` table in the database (5), initializes the expected list of countries (6), and loads the countries in the database (7).
- In the `testCountryList` method, he initializes the list of countries from the database by using `getCountryList` from the `CountryDao` class (8). Then he checks that the list he has obtained is not null (9), that it is the expected size (10), and that its content is as expected (11).
- In the `testCountryListStartsWithA` method, George initializes the list of countries starting with *A* from the database by using `getCountryListStartWith` from the `CountryDao` class (12). Then he checks that the list he has obtained is not null (13), that it is the expected size (14), and that its content is as expected (15).
- He marks the `dropDown` method with the `@AfterEach` annotation so it is executed after each test (16). In it, he drops the `COUNTRY` table from the database (17).

- In the `initExpectedCountryLists` method, he browses the country initialization data ⑱, creates a `Country` object at each step ⑲, and adds it to the list of expected countries ⑳. If the name of the country starts with *A*, he also adds it to the list of expected countries whose names start with *A* ㉑.

The tests run successfully, as shown in figure 19.1.

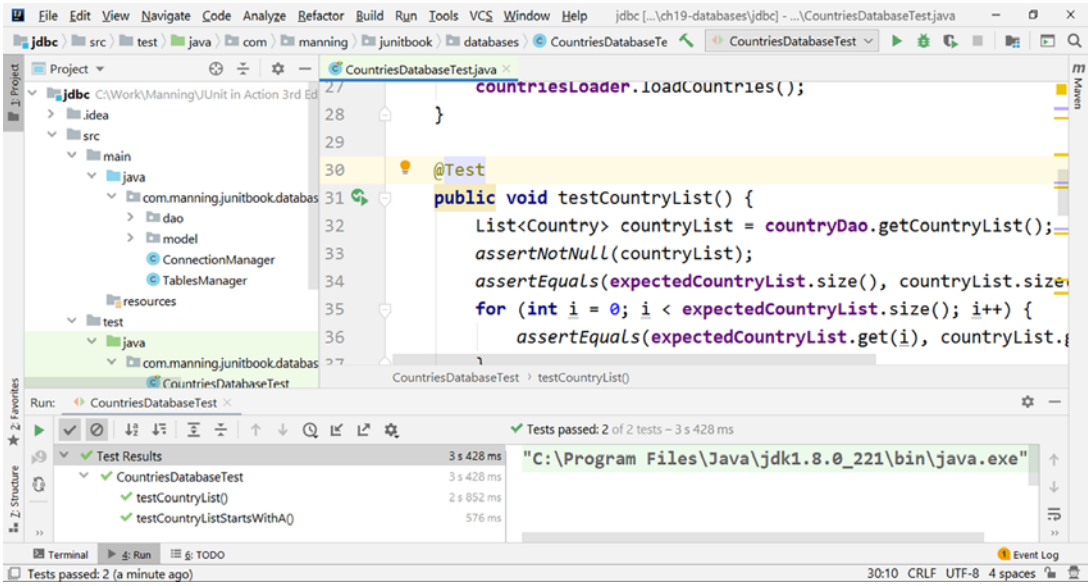


Figure 19.1 Successfully running the tests from the JDBC application to check the interaction with the `COUNTRY` table

This is the state of the application when it is assigned to George, and now he needs to improve the way it is tested. The application currently accesses and tests the database through JDBC, which requires a lot of tedious code to do the following:

- Create and open the connection
- Specify, prepare, and execute statements
- Iterate through the results
- Do the work for each iteration
- Process exceptions
- Close the connection

George will look for means to reduce the "database burden" so the developers can improve the way tests are written and reduce the database unit testing impedance mismatch.

19.3 Testing a Spring JDBC application

We introduced the Spring Framework and testing Spring applications in previous chapters. George has decided to introduce Spring into the flight-management database application, to reduce the database burden and to handle some of the tasks of interacting with the database via Spring inversion of control (IoC).

The application's `Country` class will remain untouched. George will make some other changes for the migration to Spring: first, he introduces the new dependencies in the Maven pom.xml file.

Listing 19.8 New dependencies introduced in the Maven pom.xml file

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>5.2.1.RELEASE</version>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-jdbc</artifactId>
    <version>5.2.1.RELEASE</version>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-test</artifactId>
    <version>5.2.1.RELEASE</version>
</dependency>
```

In this listing, George adds the following dependencies:

- `spring-context`, the dependency for the Spring IoC container (1).
- `spring-jdbc`, because the application is still using JDBC to access the database. Spring controls working with connections, preparing and executing statements, and processing exceptions (2).
- `spring-test`, which provides support for writing tests with the help of Spring and which is necessary to use `SpringExtension` and the `@ContextConfiguration` annotation (3).

In the test/resources project folder, George inserts two files: one to create the database schema and one to configure the Spring context of the application (listings 19.9 and 19.10).

Listing 19.9 db-schema.sql file

```
create table country( id identity , name varchar (255) , code_name varchar
     (255) );
```

In this listing, George creates the `COUNTRY` table with three fields: `ID` (identity field), `NAME`, and `CODE_NAME` (`VARCHAR` type).

Listing 19.10 application-context.xml file

```
<jdbc:embedded-database id="dataSource" type="H2">
   <jdbc:script location="classpath:db-schema.sql"/>
</jdbc:embedded-database>

<bean id="countryDao"
      class="com.manning.junitbook.databases.dao.CountryDao">
       <property name="dataSource" ref="dataSource"/>
</bean>

<bean id="countriesLoader"
     class="com.manning.junitbook.databases.CountriesLoader">
   <property name="dataSource" ref="dataSource"/>
</bean>
```

(1) (2) (3)

In this listing, George instructs the Spring container to create three beans:

- `dataSource`, which points to a JDBC embedded database of type H2. He initializes the database with the help of the db-schema.sql file from listing 19.9, which is on the classpath (1).
- `countryDao`, the DAO bean for executing `SELECT` queries to the database (2). It has a `dataSource` property pointing to the previously declared `dataSource` bean.
- `countriesLoader`, which initializes the content of the database and brings it to a known state (3). It also has a `dataSource` property pointing to the previously declared `dataSource` bean.

George changes the `CountriesLoader` class that loads the countries from the database and sets it in a known state.

Listing 19.11 `CountriesLoader` class

```
public class CountriesLoader extends JdbcDaoSupport
{

    private static final String LOAD_COUNTRIES_SQL =
            "insert into country (name, code_name) values ";

    public static final String[][] COUNTRY_INIT_DATA = {
          { "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) {
      String sql = LOAD_COUNTRIES_SQL + "('" + countryData[0] +
                   "','" + countryData[1] + "');";
       getJdbcTemplate().execute(sql);
    }
}

}
```

(1) (2) (3) (4) (5)

In this listing:

- George declares the `CountriesLoader` class as extending `JdbcDaoSupport` (1).
- George declares one SQL `INSERT` statement, to insert a country in the `COUNTRY` table (2). He then declares the initialization data for the countries to be inserted (3).
- In the `loadCountries` method, he browses the initialization data for the countries (4), builds the SQL query that inserts each particular country, and executes it against the database (5). With the Spring IoC approach, there is no need to perform the earlier tedious tasks: opening the connection, preparing the statement, executing and closing it, treating exceptions, and closing the connection.

Spring JDBC classes

`JdbcDaoSupport` is a Spring JDBC class that facilitates configuring and transferring 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 also changes the tested code to use Spring and reduce the database burden for the developers. He first implements the `CountryRowMapper` class that takes care of the mapping rules between the columns from the `COUNTRY` database table and the fields of the application `Country` class.

Listing 19.12 `CountryRowMapper` class

```
[...]
public class CountryRowMapper implements RowMapper<Country> {
   public static final String NAME = "name";
   public static final String CODE_NAME = "code_name";

   @Override
   public Country mapRow(ResultSet resultSet, int i)
                                 throws SQLException {
      Country country = new Country(resultSet.getString(NAME),
           resultSet.getString(CODE_NAME));
      return country;
   }
}
```

In this listing:

- George declares the `CountryRowMapper` class as implementing `RowMapper` (1). `RowMapper` is a Spring JDBC interface that maps the `ResultSet` obtained by accessing a database to certain objects.

- He declares the string constants to be used in the class, representing the names of the table columns (2). The class will define once how to map the columns to the object fields and can be reused. There is no more need to set the statement parameters each time, as was necessary in the JDBC version.
- He overrides the `mapRow` method inherited from the `RowMapper` interface. He gets the two string parameters from the `ResultSet` coming from the database and builds a `Country` object (3) that is returned at the end of the method (4).

George modifies the existing `CountryDao` class to use Spring to interact with the database.

Listing 19.13 `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 this listing:

- George declares the `CountryDao` class as extending `JdbcDaoSupport` (1).
- He declares two SQL `SELECT` statements to get all the countries from the `COUNTRY` table and to get the countries whose names match a pattern (2). In the second statement, he replaces the parameter with a named parameter (`:name`); it will be used this way in the class.
- He initializes a `CountryRowMapper` instance: the class he previously created (3).

- In the `getCountryList` method, George queries the `COUNTRY` table using the SQL that returns all the countries and the `CountryRowMapper` that matches the columns from the table to the fields from the `Country` object. He directly returns a list of `Country` objects (4).
- In the `getCountryListStartWith` method, George initializes a `NamedParameterJdbcTemplate` variable (5). `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`; it returns the JDBC `DataSource` used by a DAO.
- He initializes an `SqlParameterSource` variable (6). `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.
- He queries the `COUNTRY` table using the SQL that returns all the countries having names starting with *A* and the `CountryRowMapper` that matches the columns from the table to the fields from the `Country` object (7).

Finally, George changes the existing `CountriesDatabaseTest` to take advantage of the Spring JDBC approach.

Listing 19.14 `CountriesDatabaseTest` class

```
[...]
@ExtendWith(SpringExtension.class)
@ContextConfiguration("classpath:application-context.xml")
public class CountriesDatabaseTest {

    @Autowired
    private CountryDao countryDao;

    @Autowired
    private CountriesLoader countriesLoader;

    private List<Country> expectedCountryList =
                                        new ArrayList<Country>();
    private List<Country> expectedCountryListStartsWithA =
          new ArrayList<Country>();

    @BeforeEach
    public void setUp() {
        initExpectedCountryLists();
        countriesLoader.loadCountries();
    }

    @Test
    @DirtiesContext
    public void testCountryList() {
        List<Country> countryList = countryDao.getCountryList();
```

```
        assertNotNull(countryList);
        assertEquals(expectedCountryList.size(), countryList.size());
        for (int i = 0; i < expectedCountryList.size(); i++) {
            assertEquals(expectedCountryList.get(i),
                                                  countryList.get(i));
        }
    }

    @Test
    @DirtiesContext
    public void testCountryListStartsWithA() {
        List<Country> countryList =
                     countryDao.getCountryListStartWith("A");
        assertNotNull(countryList);
        assertEquals(expectedCountryListStartsWithA.size(),
                     countryList.size());
        for (int i = 0; i < expectedCountryListStartsWithA.size();
                                                          i++) {
            assertEquals(expectedCountryListStartsWithA.get(i),
                         countryList.get(i));
        }
    }

    private void initExpectedCountryLists() {
      for (int i = 0; i < CountriesLoader.COUNTRY_INIT_DATA.length; i++) {
          String[] countryInitData =
                                 CountriesLoader.COUNTRY_INIT_DATA[i];
          Country country = new Country(countryInitData[0],
                  countryInitData[1]);
          expectedCountryList.add(country);
          if (country.getName().startsWith("A")) {
              expectedCountryListStartsWithA.add(country);
          }
      }
    }
}
```

In this listing:

- George annotates the test class to be extended with `SpringExtension` (1). `SpringExtension` is used to integrate the Spring TestContext with the JUnit 5 Jupiter test.
- He also annotates the test class to look for the context configuration in the application-context.xml file from the classpath (2).
- He autowires a `CountryDao` bean (3) and a `CountriesLoader` bean (4), which are declared in the application-context.xml file.
- He initializes an empty list of expected countries (5) and an empty list of expected countries that start with *A* (6).
- He marks the `setUp` method with the `@BeforeEach` annotation so it is executed before each test (7). In it, he initializes the expected list of countries (8) and loads the countries in the database (9). The database is initialized by Spring; George has eliminated its manual initialization.

- He annotates the test methods with the `@DirtiesContext` annotation ⑩. This annotation is used when a test has modified the context (in this case, the state of the embedded database). This reduces the database burden; subsequent tests will be supplied with a new, unmodified context.
- In the `testCountryList` method, George initializes the list of countries from the database by using `getCountryList` from the `CountryDao` class ⑪. Then he checks that the list he has obtained is not null ⑫, that it is the expected size ⑬, and that its content is as expected ⑭.
- In the `testCountryListStartsWithA` method, he initializes the list of countries starting with *A* from the database by using `getCountryListStartWith` from the `CountryDao` class ⑮. Then he checks that the list he has obtained is not null ⑯, that it is the expected size ⑰, and that its content is as expected ⑱.
- In the `initExpectedCountryLists` method, he browses the country initialization data ⑲, creates a `Country` object at each step ⑳, and adds it to the expected list of countries ㉑. If the name of the country starts with *A*, he also adds it to the expected list of countries whose names start with *A* ㉒.

The tests run successfully, as shown in figure 19.2.

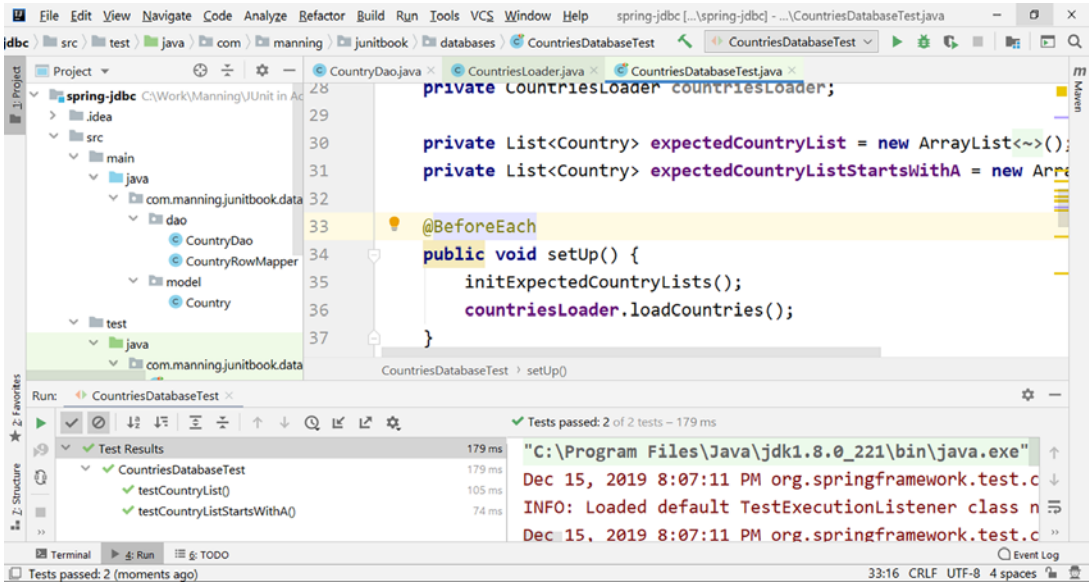


Figure 19.2 Successfully running the tests from the Spring JDBC application that check the interaction with the `COUNTRY` table

The application now accesses and tests the database through Spring JDBC. This approach has several advantages:

- It no longer requires the large amount of tedious code.

- We no longer create and open connections ourselves.
- We no longer prepare and execute statements, process exceptions, or close connections.
- Mainly, we must take care of the application context configuration to be handled by Spring, as well as the row mapper. Other than that, we only have to specify statements and iterate through the results.

Spring allows configuration alternatives, as we have demonstrated in earlier chapters. We use the XML-based configuration for tests here because it is easier to modify. As mentioned previously, for a comprehensive look at the Spring Framework and the configuration possibilities, we recommend *Spring in Action* by Craig Walls (Manning, www.manning.com/books/spring-in-action-sixth-edition).

Next, George will consider further alternatives to test the interaction with the database and to maintain the reduced database burden for the developers.

19.4 Testing a Hibernate application

The Java Persistence API (JPA) is a specification describing the management of relational data, the API the client will operate with, and metadata for ORM. Hibernate is an ORM framework for Java that implements the JPA specifications; it is currently the most popular JPA implementation. It existed before the JPA specifications were published; therefore, Hibernate has retained its old native API and offers some nonstandard features. For our example, we'll use the standard JPA.

Hibernate provides facilities for mapping an object-oriented domain model to relational database tables. It manages the incompatibilities between the object-oriented model and the relational database model by replacing direct database access with object-handling functions. Working with Hibernate provides a series of advantages for accessing and testing the database:

- *Faster development*—Hibernate 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*—Hibernate generates code for basic CRUD operations.

George introduces the Hibernate dependency in the Maven pom.xml configuration.

Listing 19.15 Hibernate dependency introduced in the Maven pom.xml file

```
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-core</artifactId>
    <version>5.4.9.Final</version>
</dependency>
```

Then, George changes the `Country` class by annotating it as an entity and annotating its fields as columns in a table.

Listing 19.16 Annotated `Country` class

```
@Entity                                                    (1)
@Table(name = "COUNTRY")                                   (2)
public class Country {
    @Id                                                    (3)
    @GeneratedValue(strategy = GenerationType.IDENTITY)    (4)
    @Column(name = "ID")                                   (5)
    private int id;

    @Column(name = "NAME")                                 (6)
    private String name;

    @Column(name = "CODE_NAME")                            (6)
    private String codeName;

    [...]
}
```

In this listing:

- George annotates the `Country` class with `@Entity` so it can represent objects in a database (1). The corresponding table in the database is provided by the `@Table` annotation and is named `COUNTRY` (2).
- The `id` field is marked as the primary key (3); its value is automatically generated using a database identity column (4). The corresponding table column is `ID` (5).
- He also marks the corresponding columns of the `name` and `codeName` fields in the class by annotating them with `@Column` (6).

The persistence.xml file is the standard configuration for Hibernate. It is located in the test/resources/META-INF folder.

Listing 19.17 persistence.xml file

```
<persistence-unit name="manning.hibernate">                                    (1)
    <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>        (2)
    <class>com.manning.junitbook.databases.model.Country</class>               (3)
    <properties>
        <property name="javax.persistence.jdbc.driver"                         (4)
                  value="org.h2.Driver"/>
        <property name="javax.persistence.jdbc.url"                            (5)
                  value="jdbc:h2:mem:test;DB_CLOSE_DELAY=-1"/>
           <property name="javax.persistence.jdbc.user" value="sa"/>           (6)
           <property name="javax.persistence.jdbc.password" value=""/>
           <property name="hibernate.dialect"                                  (7)
                     value="org.hibernate.dialect.H2Dialect"/>
           <property name="hibernate.show_sql" value="true"/>                  (8)
```

```
            <property name="hibernate.hbm2ddl.auto" value="create"/>
        </properties>
</persistence-unit>
```

In this listing:

- George specifies the persistence unit as `manning.hibernate` (1). The persistence.xml file must define a persistence unit with a unique name in the currently scoped class loader.
- He specifies the provider, meaning the underlying implementation of the JPA `EntityManager` (2). An `EntityManager` manages a set of persistent objects and has an API to insert new objects and read/update/delete the existing ones. In this case, the `EntityManager` is Hibernate.
- He defines the entity class that is managed by Hibernate as the `Country` class from our application (3).
- George specifies the JDBC driver as H2 because this is the database type in use (4).
- He specifies the URL of the H2 database. In addition, `DB_CLOSE_DELAY=-1` keeps the database open and its content in-memory as long as the virtual machine is alive (5).
- He specifies the credentials to access the database: a user and password (6).
- He sets the SQL dialect for the generated query to `H2Dialect` (7) and shows the generated SQL query on the console (8).
- He creates the database schema from scratch every time he executes the tests (9).

Finally, George rewrites the test that verifies the functionality of the database application, this time using Hibernate.

Listing 19.18 `CountriesHibernateTest` file

```
[...]
public class CountriesHibernateTest {

    private EntityManagerFactory emf;
    private EntityManager em;

    private List<Country> expectedCountryList =
                          new ArrayList<>();
    private List<Country> expectedCountryListStartsWithA =
                          new ArrayList<>();

    public static final String[][] COUNTRY_INIT_DATA = {
        { "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" } };
```

```
@BeforeEach
public void setUp() {
    initExpectedCountryLists();

    emf = Persistence.
                createEntityManagerFactory("manning.hibernate");
    em = emf.createEntityManager();

    em.getTransaction().begin();

    for (int i = 0; i < COUNTRY_INIT_DATA.length; i++) {
        String[] countryInitData = COUNTRY_INIT_DATA[i];
        Country country = new Country(countryInitData[0],
                                      countryInitData[1]);
        em.persist(country);
    }

    em.getTransaction().commit();
}

@Test
public void testCountryList() {
    List<Country> countryList = em.createQuery(
           "select c from Country c").getResultList();
    assertNotNull(countryList);
    assertEquals(COUNTRY_INIT_DATA.length, countryList.size());
    for (int i = 0; i < expectedCountryList.size(); i++) {
      assertEquals(expectedCountryList.get(i), countryList.get(i));
    }

}

@Test
public void testCountryListStartsWithA() {
    List<Country> countryList = em.createQuery(
        "select c from Country c where c.name like 'A%'").
                                            getResultList();
    assertNotNull(countryList);
    assertEquals(expectedCountryListStartsWithA.size(),
                 countryList.size());
    for (int i = 0; i < expectedCountryListStartsWithA.size();
                    i++) {
        assertEquals(expectedCountryListStartsWithA.get(i),
                     countryList.get(i));
    }
}

@AfterEach
public void dropDown() {
    em.close();
    emf.close();
}

private void initExpectedCountryLists() {
    for (int i = 0; i < COUNTRY_INIT_DATA.length; i++) {
```

5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21

```
            String[] countryInitData = COUNTRY_INIT_DATA[i];
            Country country = new Country(countryInitData[0],
                                          countryInitData[1]);
            expectedCountryList.add(country);
            if (country.getName().startsWith("A")) {
                expectedCountryListStartsWithA.add(country);
            }
        }
    }
}
```

In this listing:

- George initializes `EntityManagerFactory` (1) and `EntityManager` objects (2). `EntityManagerFactory` provides instances of `EntityManager` for connecting to the same database, while `EntityManager` accesses a database in a particular application.
- He initializes an empty list of expected countries and an empty list of expected countries that start with *A* (3). He then declares the initialization data for the countries to be inserted (4).
- He marks the `setUp` method with the `@BeforeEach` annotation so it is executed before each test (5). In it, he initializes the expected list of countries (6), `EntityManagerFactory`, and `EntityManager` (7).
- Within a transaction (8), George initializes each country, one after the other (9), and persists the newly created country to the database (10).
- In the `testCountryList` method, he initializes the list of countries from the database by using the `EntityManager` and querying the `Country` entity using a JPQL `SELECT` (11). (Java Persistence Query Language [JPQL] is a platform-independent, object-oriented query language that is a part of the JPA specification.) Note that `Country` must be written exactly like the name of the class: an uppercase first letter followed by lowercase letters. Then George checks that the list he has obtained is not null (12), that it is the expected size (13), and that its content is as expected (14).
- In the `testCountryListStartsWithA` method, he initializes the list of countries starting with *A* from the database by using the `EntityManager` and querying the `Country` entity using a JPQL `SELECT` (15). Then he checks that the list he has obtained is not null (16), that it is the expected size (17), and that its content is as expected (18).
- George marks the `dropDown` method with the `@AfterEach` annotation so it is executed after each test (19). In it, he closes `EntityManagerFactory` and `EntityManager` (20).
- In the `initExpectedCountryLists` method, he browses the country initialization data (21), creates a `Country` object at each step (22), and adds it to the expected list of countries (23). If the name of the country starts with *A*, he also adds it to the expected list of countries whose names start with *A* (24).

The tests run successfully, as shown in figure 19.3.

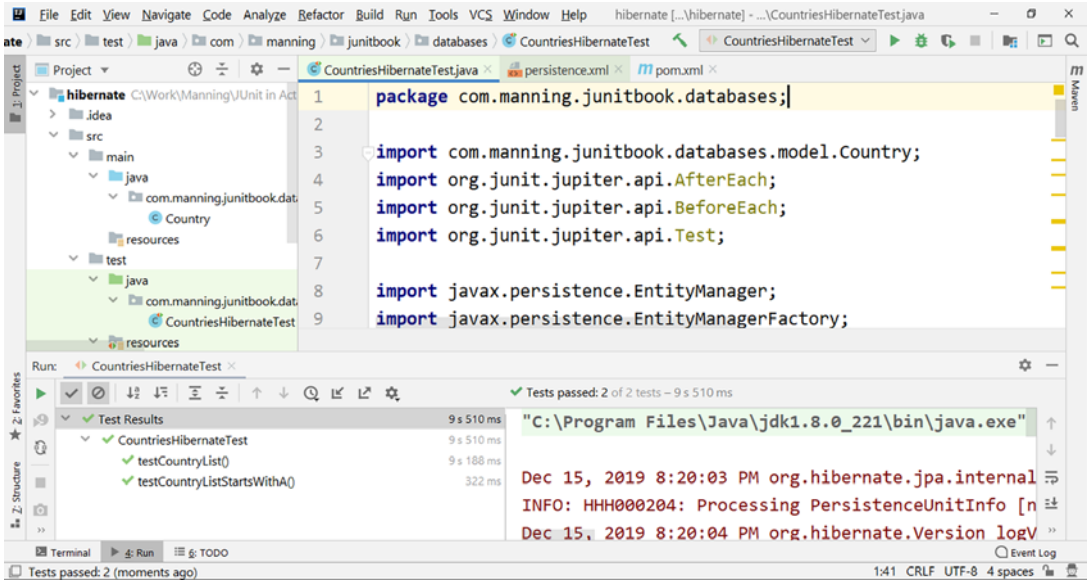


Figure 19.3 Successfully running the tests from the Hibernate application that check the interaction with the `COUNTRY` table

The application now accesses and tests the database through Hibernate. This approach comes with a few advantages:

- We no longer have to write SQL code in the application. We work only with Java code and JPQL, which are portable.
- We no longer need to map the query result columns to object fields and vice versa.
- Hibernate knows how to transform operations with implemented classes into vendor-specific SQL. So, if we change the underlying database, we will not have to touch the existing code; we'll only change the Hibernate configuration and the database dialect.

George will take one more step in considering alternatives to test interactions with the database: combining Spring and Hibernate. We'll see this approach in the next section.

19.5 Testing a Spring Hibernate application

Hibernate provides facilities for mapping an object-oriented domain model to relational database tables. Spring can take advantage of the IoC pattern to simplify database interaction tasks. To integrate Hibernate and Spring, George first adds the needed dependencies in the Maven pom.xml configuration file.

Listing 19.19 Spring and Hibernate dependencies in the Maven pom.xml file

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>5.2.1.RELEASE</version>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-orm</artifactId>
    <version>5.2.1.RELEASE</version>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-test</artifactId>
    <version>5.2.1.RELEASE</version>
</dependency>
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-core</artifactId>
     <version>5.4.9.Final</version>
</dependency>
```

In this listing, George adds the following dependencies:

- `spring-context`, the dependency for the Spring IoC container (1).
- `spring-orm`, because the application is still using Hibernate as an ORM framework to access the database. Spring handles working with connections, preparing and executing statements, and processing exceptions (2).
- `spring-test`, which provides support for writing tests with the help of Spring and which is necessary to use `SpringExtension` and the `@ContextConfiguration` annotation (3).
- `hibernate-code`, for interacting with the database through Hibernate (4).

George makes some changes to the persistence.xml file, the standard configuration for Hibernate. Only some minimal information will remain here, as database access control will be handled by Spring, and a good part of the information was moved to application-context.xml.

Listing 19.20 persistence.xml file

```
<persistence-unit name="manning.hibernate">
    <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
    <class>com.manning.junitbook.databases.model.Country</class>
</persistence-unit>
```

In this listing:

- George specifies the persistence unit as `manning.hibernate` (1). The persistence.xml file must define a persistence unit with a unique name in the currently scoped class loader.

- He specifies the provider, meaning the underlying implementation of the JPA `EntityManager` (2). In this case, the `EntityManager` is Hibernate.
- He defines the entity class that is managed by Hibernate as the `Country` class from the application (3).

Next, George moves the database access configuration to the application-context.xml file that configures the Spring container.

Listing 19.21 application-context.xml file

```
<tx:annotation-driven transaction-manager="txManager"/>
<bean id="dataSource" class=
     "org.springframework.jdbc.datasource.DriverManagerDataSource">
     <property name="driverClassName" value="org.h2.Driver"/>
     <property name="url" value="jdbc:h2:mem:test;DB_CLOSE_DELAY=-1"/>
     <property name="username" value="sa"/>
     <property name="password" value=""/>
</bean>

<bean id="entityManagerFactory" class=
  "org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
     <property name="persistenceUnitName" value="manning.hibernate" />
     <property name="dataSource" ref="dataSource"/>
     <property name="jpaProperties">
         <props>
           <prop key=
           "hibernate.dialect">org.hibernate.dialect.H2Dialect</prop>
           <prop key="hibernate.show_sql">true</prop>
           <prop key="hibernate.hbm2ddl.auto">create</prop>
         </props>
     </property>
</bean>

<bean id="txManager" class=
     "org.springframework.orm.jpa.JpaTransactionManager">
     <property name="entityManagerFactory" ref="entityManagerFactory" />
     <property name="dataSource" ref="dataSource" />
</bean>

<bean class="com.manning.junitbook.databases.CountryService"/>
```

(1) (2) (3) (4) (5) (6) (7) (8) (9) (10) (11) (12) (13) (14) (15)

In this listing:

- `<tx:annotation-driven>` tells the Spring context to use an annotation-based transaction management configuration (1).
- George configures access to the data source (2) by specifying the driver as H2, because this is the database type in use (3). He specifies the URL of the H2 database. In addition, `DB_CLOSE_DELAY=-1` keeps the database open and its content in memory as long as the virtual machine is alive (4).

- He specifies the credentials to access the database: a user and password (5).
- He creates an `EntityManagerFactory` bean (6) and sets its properties: the persistence unit name (as defined in the persistence.xml file) (7), the data source (defined previously) (8), and the SQL dialect for the generated query (`H2Dialect`) (9). He shows the generated SQL query on the console (10) and creates the database schema from scratch every time he executes the tests (11).
- To process the annotation-based transaction configuration, a transaction manager bean needs to be created. George declares it (12) and sets its entity manager factory (13) and data source (14) properties.
- He declares a `CountryService` bean (15) because he'll create this class in the code and group the logic of the interaction with the database.

George next creates the `CountryService` class that contains the logic of the interaction with the database.

Listing 19.22 `CountryService` class

```
[...]
public class CountryService {

    @PersistenceContext
    private EntityManager em;                                          (1)

    public static final String[][] COUNTRY_INIT_DATA =
      { { "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" } };       (2)

    @Transactional
    public void init() {
        for (int i = 0; i < COUNTRY_INIT_DATA.length; i++) {
            String[] countryInitData = COUNTRY_INIT_DATA[i];
            Country country = new Country(countryInitData[0],
                                          countryInitData[1]);         (4)
            em.persist(country);
        }
    }                                                                  (3)

    @Transactional
    public void clear() {
        em.createQuery("delete from Country c").executeUpdate();      (5)
    }

    public List<Country> getAllCountries() {
        return em.createQuery("select c from Country c")
                 .getResultList();                                     (6)
    }
```

```
    public List<Country> getCountriesStartingWithA() {
        return em.createQuery(
            "select c from Country c where c.name like 'A%'")
            .getResultList();
    }
}
```

In this listing:

- George declares an `EntityManager` bean and annotates it with `@PersistenceContext` (1). `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, George simply needs to request that it be injected into one of the components via `@PersistenceContext`.
- He declares the initialization data for the countries to be inserted (2).
- He annotates the `init` and `clear` methods as `@Transactional` (3). He does so because these methods modify the content of the database, and all such methods must be executed within a transaction.
- He browses the initialization data for the countries, creates each `Country` object, and persists it within the database (4).
- The `clear` method deletes all countries from the `Country` entity using a JPQL `DELETE` (5). As in listing 19.18, `Country` must be written exactly this way to match the class name.
- The `getAllCountries` method selects all the countries from the `Country` entity using a JPQL `SELECT` (6).
- The `getCountriesStartingWithA` method selects all countries from the `Country` entity having names starting with *A* using a JPQL `SELECT` (7).

Finally, George modifies the `CountriesHibernateTest` class that tests the logic of the interaction with the database.

Listing 19.23 `CountriesHibernateTest` class

```
[...]
@ExtendWith(SpringExtension.class)
@ContextConfiguration("classpath:application-context.xml")
public class CountriesHibernateTest {

    @Autowired
    private CountryService countryService;

    private List<Country> expectedCountryList = new ArrayList<>();
    private List<Country> expectedCountryListStartsWithA =
                                new ArrayList<>();

    @BeforeEach
    public void setUp() {
        countryService.init();
        initExpectedCountryLists();
    }
```

```
    @Test
    public void testCountryList() {
        List<Country> countryList = countryService.getAllCountries();
        assertNotNull(countryList);
        assertEquals(COUNTRY_INIT_DATA.length, countryList.size());
        for (int i = 0; i < expectedCountryList.size(); i++) {
          assertEquals(expectedCountryList.get(i), countryList.get(i));
        }
    }

    @Test
    public void testCountryListStartsWithA() {
        List<Country> countryList =
                countryService.getCountriesStartingWithA();
        assertNotNull(countryList);
        assertEquals(expectedCountryListStartsWithA.size(),
                     countryList.size());
        for (int i = 0; i < expectedCountryListStartsWithA.size(); i++) {
            assertEquals(expectedCountryListStartsWithA.get(i),
                         countryList.get(i));
        }
    }

    @AfterEach
    public void dropDown() {
        countryService.clear();
    }

    private void initExpectedCountryLists() {
        for (int i = 0; i < COUNTRY_INIT_DATA.length; i++) {
            String[] countryInitData = COUNTRY_INIT_DATA[i];
            Country country = new Country(countryInitData[0],
                                          countryInitData[1]);
            expectedCountryList.add(country);
            if (country.getName().startsWith("A")) {
                expectedCountryListStartsWithA.add(country);
            }
        }
    }
}
```

In this listing:

- George annotates the test class to be extended with `SpringExtension` (1). `SpringExtension` is used to integrate the Spring TestContext with the JUnit 5 Jupiter test. This allows us to use other Spring annotations as well (such as `@ContextConfiguration` and `@Transactional`), but it also requires us to use JUnit 5 and its annotations.
- He annotates the test class to look for the context configuration in the application-context.xml file from the classpath (2).
- He declares and autowires a `CountryService` bean. This bean is created and injected by the Spring container (3).
- George initializes an empty list of expected countries and an empty list of expected countries that start with *A* (4).

- He marks the `setUp` method with the `@BeforeEach` annotation so it is executed before each test ❺. In it, he initializes the database content through the `init` method from the `CountryService` class ❻ and initializes the expected list of countries ❼.
- In the `testCountryList` method, he initializes the list of countries from the database by using the `getAllCountries` method from the `CountryService` class ❽. Then he checks that the list he has obtained is not null ❾, that it is the expected size ❿, and that its content is as expected ⓫.
- In the `testCountryListStartsWithA` method, George initializes the list of countries starting with *A* from the database by using the `getCountriesStartingWithA` method from the `CountryService` class ⓬. Then he checks that the list he has obtained is not null ⓭, that it is the expected size ⓮, and that its content is as expected ⓯.
- He marks the `dropDown` method with the `@AfterEach` annotation so it is executed after each test ⓰. In it, he clears the `COUNTRY` table's contents using the `clear` method from the `CountryService` class ⓱.
- In the `initExpectedCountryLists` method, George browses the country initialization data ⓲, creates a `Country` object at each step ⓳, and adds it to the expected list of countries ⓴. If the name of the country starts with *A*, he also adds it to the expected list of countries whose names start with *A* ㉑.

The tests run successfully, as shown in figure 19.4.

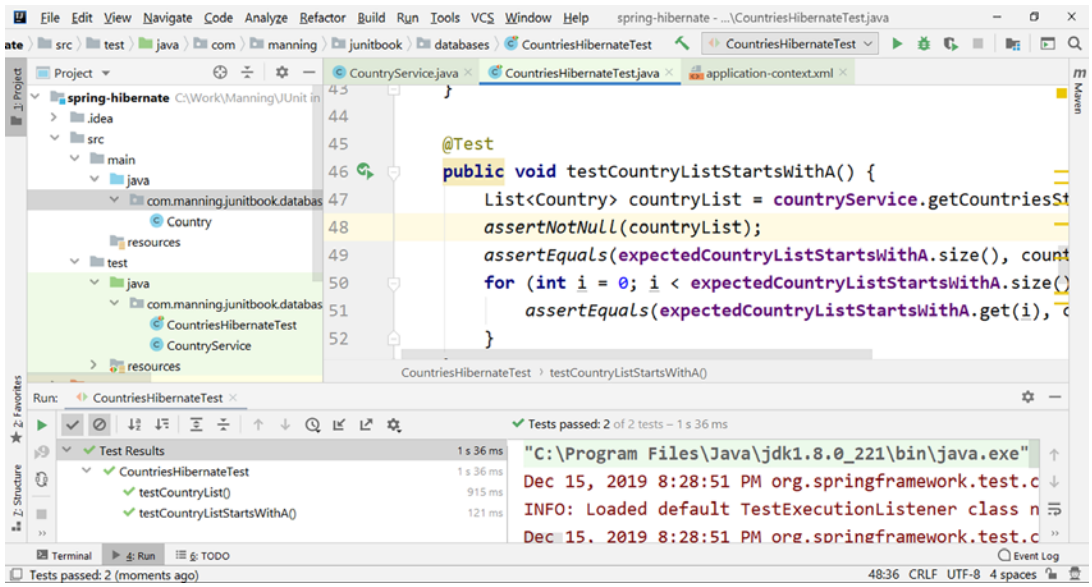


Figure 19.4 Successfully running the tests from the Spring Hibernate application that check the interaction with the `COUNTRY` table

The application now accesses and tests the database through Spring Hibernate. This approach offers several advantages:

- We no longer need to write SQL code in the application. We work only with Java code and JPQL, which are portable.
- We no longer have to create, open, or close connections ourselves.
- We do not have to process exceptions.
- We primarily take care of the application context to be handled by Spring, which includes the data source, transaction manager, and entity manager factory configuration.
- Hibernate knows how to transform operations with the implemented classes into vendor-specific SQL. So, if we change the underlying database, we will not have to touch the existing code; we'll only change the Hibernate configuration and the database dialect.

19.6 Comparing the approaches for testing database applications

We have followed George as he revised a simple JDBC application for use with Spring and Hibernate. We demonstrated how parts of the application were revised and how each approach simplified testing and interacting with the database. Our purpose was to analyze each approach and how the database burden can be reduced for developers. Table 19.1 summarizes the characteristics of these approaches.

Table 19.1 Comparison of working with a database application and testing it with JDBC, Spring JDBC, Hibernate, and Spring Hibernate

Application type	Characteristics
JDBC	■ SQL code needs to be written in the tests. ■ No portability between databases. ■ Full control over what the application is doing. ■ Manual work for the developer to interact with the database; for example: – Creating and opening connections – Specifying, preparing, and executing statements – Iterating through the results – Doing the work for each iteration – Processing exceptions – Closing the connection
Spring JDBC	■ SQL code needs to be written in the tests. ■ No portability between databases. ■ Need to take care of the row mapper and the application context configuration that is handled by Spring. ■ Control on the queries that the application is executing against the database. ■ Reduces the manual work to interact with the database: – No creating/opening/closing connections ourselves – No preparing and executing statements – No processing exceptions

Table 19.1 Comparison of working with a database application and testing it with JDBC, Spring JDBC, Hibernate, and Spring Hibernate *(continued)*

Application type	Characteristics
Hibernate	▪ No SQL code in the application; only JPQL, which is portable. ▪ Developers work only with Java code. ▪ No mapping 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 through Java code.
Spring Hibernate	▪ No SQL code in the application; only JPQL, which is portable. ▪ Developers work only with Java code. ▪ No mapping 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 information from the application context.

Notice that introducing at least one framework like Spring or Hibernate into the application greatly simplifies testing the database and developing the application itself. These are the most popular Java frameworks, and they provide many benefits (including testing interaction with a database, as we have demonstrated); we recommend that you consider adopting them into your project (if you haven't already).

This chapter has focused on alternatives for testing a database application with JUnit 5. The example tests covered only insertion and selection operations, but you can easily extend the tests offered here to include update and delete operations.

Chapter 20 will start the last part of the book, dedicated to the systematic development of applications with the help of JUnit 5. It will discuss one of the most widely used development techniques today: test-driven development (TDD).

Summary

This chapter has covered the following:

- Examining the database unit testing impedance mismatch, including these challenges: unit tests must exercise code in isolation; unit tests must be easy to write and run; and unit tests must be fast to run.
- Implementing tests for a JDBC application, which requires us to write SQL code in the tests and do a lot of tedious work: creating/opening/closing connections to the database; specifying, preparing, and executing statements; and handling exceptions.
- Implementing tests for a Spring JDBC application. This still requires us to write SQL code in the tests, but the Spring container handles creating/opening/closing connections; specifying, preparing, and executing statements; and handling exceptions.

- Implementing tests for a Hibernate application. No SQL code is required; we work only with 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; we work only with Java code. The application is portable to another database with minimum configuration changes. Additionally, the database configuration is handled by Spring, based on information from the application context.

Part 5

Developing applications with JUnit 5

This part of the book examines working with JUnit 5 as part of the everyday activity of contemporary projects. Chapter 20 discusses project development using one of today's popular development techniques: test-driven development. We will demonstrate how to create safe applications whose functionality is driven by the tests.

Chapter 21 discusses developing projects using behavior-driven development. We will show how to create applications that address business needs: applications that not only do things right but also do the right thing.

In chapter 22, we build a test pyramid strategy with the help of JUnit 5. We will demonstrate testing from the ground level (unit testing) to the upper levels (integration testing, system testing, and acceptance testing).

20 Test-driven development with JUnit 5

This chapter covers

- Moving a non-TDD application to TDD
- Refactoring a TDD application
- Using TDD to implement new functionality

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

In this chapter, we will show how to develop safe, flexible applications using test-driven development (TDD): a technique that can greatly increase development speed and eliminate much of the debugging nightmare—all with the help of JUnit 5 and its features. We will point out the main concepts involved in TDD and apply them in developing a Java application that Tested Data Systems (our example company) will use to implement the business logic for managing flights and passengers and following a set of policies. Our focus will be on clearly explaining TDD and proving its benefits by demonstrating how to put it in practice, step by step.

20.1 TDD main concepts

Test-driven development is a programming practice that uses a short, repeating development cycle in which requirements are converted into test cases, and then the program is modified to make the tests pass:

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

The development of this technique is attributed to the American software engineer Kent Beck. TDD supports simple designs and inspires safety: it looks for "clean code that works."

This is different from traditional software development, where code may be added without having to verify that it meets requirements. In a classical approach, developing a program means we write code and then do some testing by observing its behavior. So, the conventional development cycle goes something like this:

```
[code, test, (repeat)]
```

TDD uses a surprising variation:

```
[test, code, (repeat)]
```

The test drives the design and becomes the first client of the method.

TDD's benefits include the following:

- We write code that is driven by clear goals, and we make sure we address exactly what our application needs to do.
- Introducing new functionality is much faster. On the one hand, tests drive us to implement code that does what it is supposed to do. On the other hand, tests will prevent us from introducing bugs into the existing working code.
- Tests act as documentation for the application. We can follow them and understand what problems our code is supposed to solve.

We said that TDD uses this development cycle:

```
[test, code, (repeat)]
```

In fact, it looks like this:

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

Refactoring is the process of modifying a software system in a way that does not impact its external behavior but does improve its internal structure. To make sure external behavior is not affected, we need to rely on the tests.

When we receive specifications to add new functionality to an application, we have to first understand them before we can add them to the code. What if we first implement a test that will show us *what* we have to do, and then think about *how* to do it? This is one of the fundamental principles of TDD.

When we begin working on an application, at the very least, we need to understand the fundamental idea of what the software is supposed to do. But if we want to

check what classes or methods do, our choices are limited: read the documentation or look for sample code that invokes the functionality. Most programmers prefer to work with the code. And well-written unit tests do exactly this: they invoke our code and, consequently, provide a working specification for the code's functionality. As a result, TDD effectively helps to build a significant part of the application's technical documentation.

20.2 The flight-management application

As we have discussed throughout this book, Tested Data Systems (our example company) is developing a flight-management application for one of its customers. Currently, the application is able to create and set up flights, and add passengers to and remove them from flights.

In this chapter, we'll walk through scenarios that follow the developers' everyday work. We'll start with the non-TDD application, which is supposed to do several things such as follow company policies for regular and VIP passengers. We need to understand the application and make sure it is really implementing the expected operations. So, we have to cover the existing code with unit tests. Once we've done that, we'll address another challenge: adding new functionality by first understanding what needs to be done; next, writing tests that fail; and then, writing the code that fixes the tests. This work cycle is one of the foundations of TDD.

John is joining the development of the flight-management application, which is a Java application built with the help of Maven. The software must maintain a policy regarding adding passengers to and removing them from flights. Flights may be different types: currently, there are economy and business flights, but other types may be added later, depending on customer requirements. Both VIP passengers and regular customers may be added to economy flights, but only VIP passengers may be added to business flights (figure 20.1).

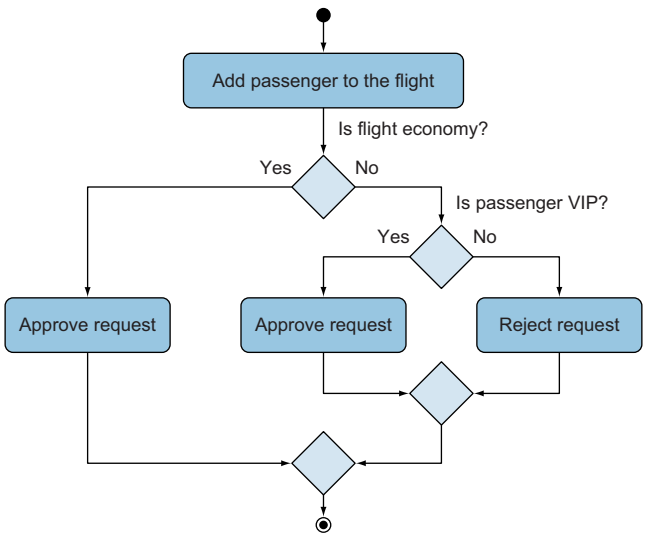


Figure 20.1 The business logic of adding passengers to a flight: if it is a business flight, only VIP passengers may be added to it. Any passenger can be added to an economy flight.

There is also a policy for removing passengers from flights: a regular passenger may be removed from a flight, but a VIP passenger cannot be removed (figure 20.2). As we can see from these two activity diagrams, the initial business logic focuses on decision making.

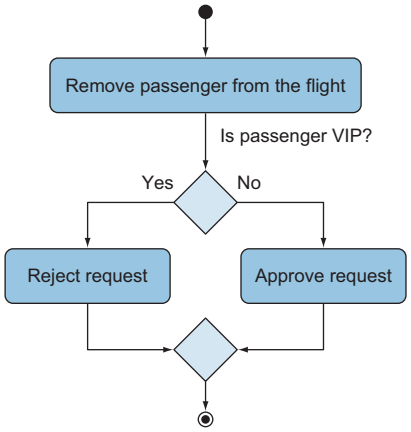


Figure 20.2 The business logic of removing passengers from a flight: only regular passengers may be removed.

Let's look at the initial design for this application (figure 20.3). It has a field called `flightType` in the `Flight` class. Its value determines the behavior of the `addPassenger` and `removePassenger` methods. The developers need to focus on decision making at the level of the code for these two methods.

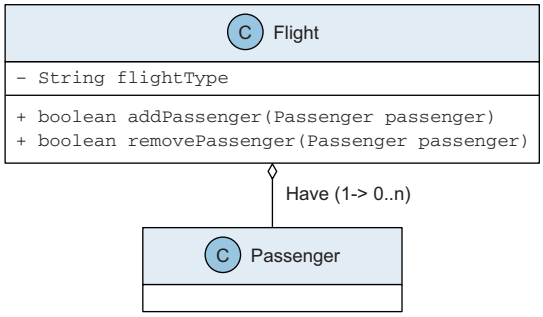


Figure 20.3 The class diagram for the flight-management application: the flight type is kept as a field in the `Flight` class.

The following listing shows the `Passenger` class.

Listing 20.1 `Passenger` class

```
public class Passenger {

    private String name;
    private boolean vip;
```

```
    public Passenger(String name, boolean vip) {
        this.name = name;
        this.vip = vip;
    }

    public String getName() {
        return name;
    }

    public boolean isVip() {
        return vip;
    }
}
```

In this listing:

- The `Passenger` class contains a `name` field ❶ together with a getter for it ❹.
- It also contains a `vip` field ❷ together with a getter for it ❺.
- The constructor of the `Passenger` class initializes the `name` and `vip` fields ❸.

The next listing shows the `Flight` class.

Listing 20.2 `Flight` class

```
public class Flight {

    private String id;
    private List<Passenger> passengers = new ArrayList<Passenger>();
    private String flightType;

    public Flight(String id, String flightType) {
        this.id = id;
        this.flightType = flightType;
    }

    public String getId() {
        return id;
    }

    public List<Passenger> getPassengersList() {
        return Collections.unmodifiableList(passengers);
    }

    public String getFlightType() {
        return flightType;
    }


    public boolean addPassenger(Passenger passenger) {
        switch (flightType) {
            case "Economy":
                return passengers.add(passenger);
```

```
            case "Business":
                if (passenger.isVip()) {
                    return passengers.add(passenger);
                }
                return false;
            default:
                throw new RuntimeException("Unknown type: " + flightType);
        }

    }

    public boolean removePassenger(Passenger passenger) {
        switch (flightType) {
            case "Economy":
                if (!passenger.isVip()) {
                    return passengers.remove(passenger);
                }
                return false;
            case "Business":
                return false;
            default:
                throw new RuntimeException("Unknown type: " + flightType);
            }
    }

}
```

In this listing:

- The `Flight` class contains an identifier ❶ together with a getter for it ❺, a list of passengers initialized as an empty list ❷ together with a getter for it ❻, and a flight type ❸ together with a getter for it ❼.
- The constructor of the `Flight` class initializes the `id` and the `flightType` fields ❹.
- The `addPassenger` method checks the flight type ❽. If it is an economy flight, any passengers can be added ❾. If it is a business flight, only VIP passengers can be added ❿. Otherwise (if the flight is neither an economy nor a business flight), the method will throw an exception, as it cannot handle an unknown flight type ⓫.
- The `removePassenger` method checks the flight type ⓬. If it is an economy flight, only regular passengers can be removed ⓭. If it is a business flight, passengers cannot be removed ⓮. Otherwise (if the flight is neither an economy nor a business flight), the method will throw an exception, as it cannot handle an unknown flight type ⓯.

The application has no tests yet. Instead, the initial developers wrote some code in which they simply followed the execution and compared it with their expectations. For example, there is an `Airport` class, including a `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 `Airport` class, including the `main` method

```
public class Airport {

    public static void main(String[] args) {
        Flight economyFlight = new Flight("1", "Economy");
        Flight businessFlight = new Flight("2", "Business");

        Passenger james = new Passenger("James", true);
        Passenger mike = new Passenger("Mike", false);

        businessFlight.addPassenger(james);
        businessFlight.removePassenger(james);
        businessFlight.addPassenger(mike);
        economyFlight.addPassenger(mike);

        System.out.println("Business flight passengers list:");
        for (Passenger passenger: businessFlight.getPassengersList()) {
            System.out.println(passenger.getName());
        }

        System.out.println("Economy flight passengers list:");
        for (Passenger passenger: economyFlight.getPassengersList()) {
            System.out.println(passenger.getName());
        }
    }
}
```

In this listing:

- We initialize an economy flight and a business flight (1). We also initialize James as a VIP passenger and Mike as a regular passenger (2).
- We try to add James to and remove him from the business flight (3), and then we try to add Mike to and remove him from the business flight (4) and the economy flight (5).
- We print the list of passengers on the business flight (6) and the economy flight (7).

The result of running this program is shown in figure 20.4. James, a VIP passenger, has been added to the business flight, and 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.

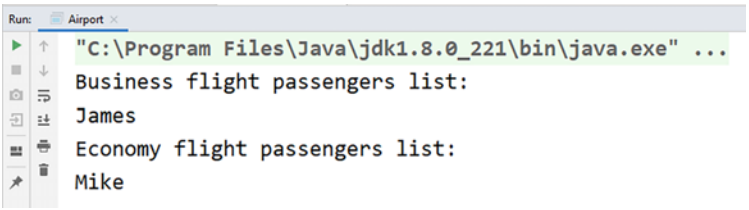


Figure 20.4 The result of running the non-TDD flight-management application: the VIP passenger has been added to the business flight, and the regular passenger has been added to the economy flight.

So far, things are working as expected, following the policies that we previously defined. John is satisfied with the way the application works, but he needs to develop it further. To build a reliable application and to be able to easily and safely understand and implement the business logic, John considers moving the application to the TDD approach.

20.3 Preparing the flight-management application for TDD

To move the flight-management application to TDD, John first needs to cover the existing business logic with JUnit 5 tests. He adds 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 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, multiplying two flight types by two passenger types, this means four 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 uses the JUnit 5 nested test capability, as the tests share similarities between them and can be grouped: tests for economy flights and tests for business flights.

Listing 20.5 Testing the business logic for an economy flight

```
public class AirportTest {

    @DisplayName("Given there is an economy flight")
    @Nested
    class EconomyFlightTest {

        private Flight economyFlight;

        @BeforeEach
```

1

2

```
        void setUp() {
            economyFlight = new Flight("1", "Economy");
        }

        @Test
        public void testEconomyFlightRegularPassenger() {
            Passenger mike = new Passenger("Mike", false);

            assertEquals("1", economyFlight.getId());
            assertEquals(true, economyFlight.addPassenger(mike));
            assertEquals(1, economyFlight.getPassengersList().size());
            assertEquals("Mike",
                    economyFlight.getPassengersList().get(0).getName());

            assertEquals(true, economyFlight.removePassenger(mike));
            assertEquals(0, economyFlight.getPassengersList().size());
        }

        @Test
        public void testEconomyFlightVipPassenger() {
            Passenger james = new Passenger("James", true);

            assertEquals("1", economyFlight.getId());
            assertEquals(true, economyFlight.addPassenger(james));
            assertEquals(1, economyFlight.getPassengersList().size());
            assertEquals("James",
                    economyFlight.getPassengersList().get(0).getName());

            assertEquals(false, economyFlight.removePassenger(james));
            assertEquals(1, economyFlight.getPassengersList().size());
        }
    }
}
```

In this listing:

- John declares a nested test class `EconomyFlightTest` and labels it `"Given there is an economy flight"` with the help of the `@DisplayName` annotation (1).
- He declares an economy flight and initializes it before the execution of each test (2).
- When testing how the economy flight works with a regular passenger, he creates Mike as a regular passenger (3). Then, he checks the ID of the flight (4), whether he can add Mike on the economy flight and that he can find Mike there (5), and whether he can remove Mike from the economy flight and that Mike is no longer there (6).
- When testing how the economy flight works with a VIP passenger, he creates James as a VIP passenger (7). Then, he checks the ID of the flight (8), whether he can add James on the economy flight and that he can find James there (9), and whether he cannot remove James from the economy flight and that James is still there (10).

John follows the business logic for a business flight and translates it into the following tests.

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 testBusinessFlightVipPassenger() {
        Passenger james = new Passenger("James", true);

        assertEquals(true, businessFlight.addPassenger(james));
        assertEquals(1, businessFlight.getPassengersList().size());
        assertEquals(false, businessFlight.removePassenger(james));
        assertEquals(1, businessFlight.getPassengersList().size());

    }
}
}
```

In this listing:

- John declares a nested test class `BusinessFlightTest` and labels it `"Given there is a business flight"` with the help of the `@DisplayName` annotation (1).
- He declares a business flight and initializes it before the execution of each test (2).
- When testing how the business flight works with a regular passenger, he creates Mike as a regular passenger (3). Then, he checks that he cannot add Mike to the business flight (4) and that trying to remove Mike from the business flight also has no effect (5).

- When testing how the business flight works with a VIP passenger, he creates James as a VIP passenger (6). Then, he checks that he can add James to the business flight and that he can find James there (7) and that he cannot remove James from the business flight and that James is still there (8).

If we run the tests with coverage from within IntelliJ IDEA, we get the results shown in figure 20.5. For more details about test coverage and how to run tests with coverage from IntelliJ IDEA, you can revisit chapter 6.

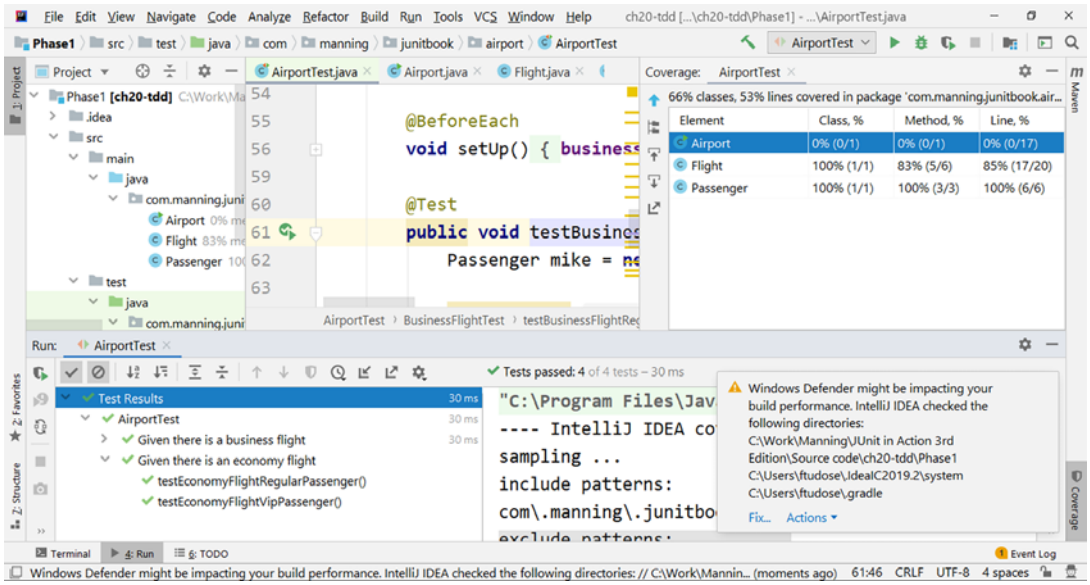


Figure 20.5 The result of running the economy and business flight tests with coverage using IntelliJ IDEA: the `Airport` class is uncovered (it contains the `main` method; we do not test it), and the `Flight` class has coverage of less than 100%.

John has successfully verified the functionality of the application by writing tests for all the scenarios that result from the business logic (figure 20.1 and 20.2). It is possible that in real life you may begin working with an application that has no tests and want to move to TDD. Before you do, you will have to test the application as it is.

John's work also provides additional conclusions. 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. In addition, the code coverage is not 100%. The `getFlightType` method is not used, and the default case—when a flight is neither economy nor business type—is not covered. This suggests to John the need to refactor the application, to remove the elements that are not used. He is confident about doing this because the application is now covered with tests and, as we said earlier, TDD enables us to gain confidence in our code over time.

20.4 Refactoring the flight-management application

John has noticed that the lines of code that are not being executed are those related to using the `flightType` field. And the default case will never be executed, as the flight type is expected to be either economy or business; these default alternatives are needed because the code will not compile otherwise. Can John get rid of them by doing some refactoring and replacing the conditional statements with polymorphism?

The key to refactoring is to move the design to using polymorphism instead of procedural-style conditional code. With *polymorphism* (the ability of one object to pass more than one IS-A test), the method you are calling is determined not at compile time, but at runtime, depending on the effective object type (see chapter 6).

The principle in action here is called the *open/closed principle* (figure 20.6). Practically, it means the design shown on the left will require changes to the existing class each time we add a new flight type. These changes may reflect in each conditional decision made based on the flight type. In addition, we are forced to rely on the `flightType` field and introduce unexecuted default cases.

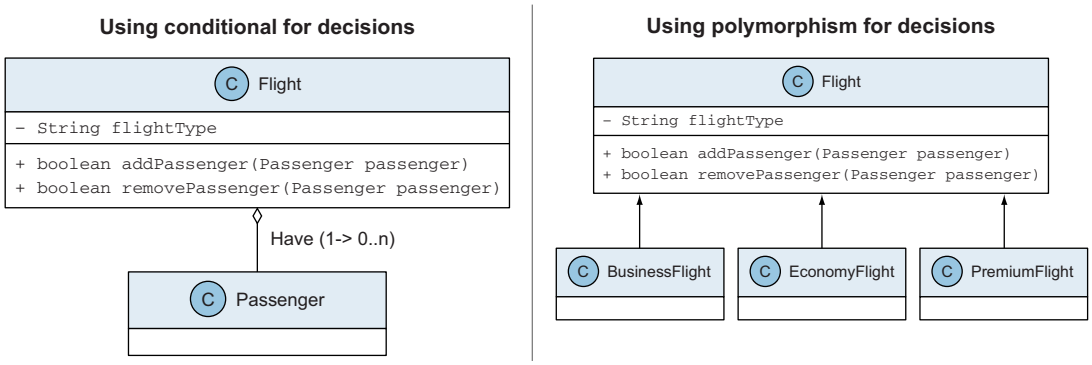


Figure 20.6 Refactoring the flight-management application by replacing the conditional with polymorphism: the `flightType` field is removed, and a hierarchy of classes is introduced.

With the design on the right—which is refactored by replacing conditional with polymorphism—we do not need a `flightType` evaluation or a default value in the `switch` instructions from listing 20.2. We can even add a new type—let's anticipate a little and call it `PremiumFlight`—by simply extending the base class and defining its behavior. According to the open/closed principle, the hierarchy will be open for extensions (we can easily add new classes) but closed for modifications (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 not affecting already working functionality?" The answer is that passing the tests

provides assurance that existing functionality is untouched. The benefits of the TDD approach really show themselves!

The refactoring will be achieved by keeping the base `Flight` class (listing 20.7) and, for each conditional type, adding a separate class to extend `Flight`. John will change `addPassenger` and `removePassenger` to abstract methods and delegate their implementation to subclasses. The `flightType` field is no longer significant and will be removed.

Listing 20.7 Abstract `Flight` class, the basis of the hierarchy

```
public abstract class Flight {

   private String id;
   List<Passenger> passengers = new ArrayList<Passenger>();

   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);

   public abstract boolean removePassenger(Passenger passenger);

}
```

In this listing:

- John declares the class as abstract, making it the basis of the flight hierarchy (1).
- He makes the `passengers` list package-private, allowing it to be directly inherited by the subclasses in the same package (2).
- John declares `addPassenger` and `removePassenger` as abstract methods, delegating their implementation to the subclasses (3).

John introduces an `EconomyFlight` class that extends `Flight` and implements the inherited `addPassenger` and `removePassenger` abstract methods.

Listing 20.8 `EconomyFlight` class, extending the abstract `Flight` class

```
public class EconomyFlight extends Flight {

   public EconomyFlight(String id) {
      super(id);
   }
```

```
    @Override
    public boolean addPassenger(Passenger passenger) {
        return passengers.add(passenger);
    }

    @Override
    public boolean removePassenger(Passenger passenger) {
        if (!passenger.isVip()) {
            return passengers.remove(passenger);
        }
        return false;
    }

}
```

In this listing:

- John declares the `EconomyFlight` class extending the `Flight` abstract class (1) and creates a constructor calling the constructor of the superclass (2).
- He implements the `addPassenger` method according to the business logic: he simply adds a passenger to an economy flight with no restrictions (3).
- He implements the `removePassenger` method according to the business logic: a passenger can be removed from a flight only if the passenger is not a VIP (4).

John also introduces a `BusinessFlight` class that extends `Flight` and implements the inherited `addPassenger` and `removePassenger` abstract methods.

Listing 20.9 `BusinessFlight` class, extending the abstract `Flight` class

```
public class BusinessFlight extends Flight {

    public BusinessFlight(String id) {
        super(id);
    }

    @Override
    public boolean addPassenger(Passenger passenger) {
        if (passenger.isVip()) {
            return passengers.add(passenger);
        }
        return false;
    }

    @Override
    public boolean removePassenger(Passenger passenger) {
        return false;
    }

}
```

In this listing:

- John declares the `BusinessFlight` class extending the `Flight` abstract class (1) and creates a constructor calling the constructor of the superclass (2).
- He implements the `addPassenger` method according to the business logic: only a VIP passenger can be added to a business flight (3).
- He implements the `removePassenger` method according to the business logic: a passenger cannot be removed from a business flight (4).

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 previous default case that was never expected and that threw an exception. Of course, the refactoring and the API changes propagate into the tests, as shown next.

Listing 20.10 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");
        }
        [...]
    }

    @DisplayName("Given there is a business flight")
    @Nested
    class BusinessFlightTest {
        private Flight businessFlight;

        @BeforeEach
        void setUp() {
            businessFlight = new BusinessFlight("2");
        }
        [...]
    }

}
```

(1) (2)

In this listing, John replaces the previous `Flight` instantiations with instantiations of `EconomyFlight` (1) and `BusinessFlight` (2). He also removes the `Airport` class that served as a client for the `Passenger` and `Flight` classes—it is no longer needed, now that John has introduced the tests. It previously served to declare the `main` method that created different types of flights and passengers and made them act together.

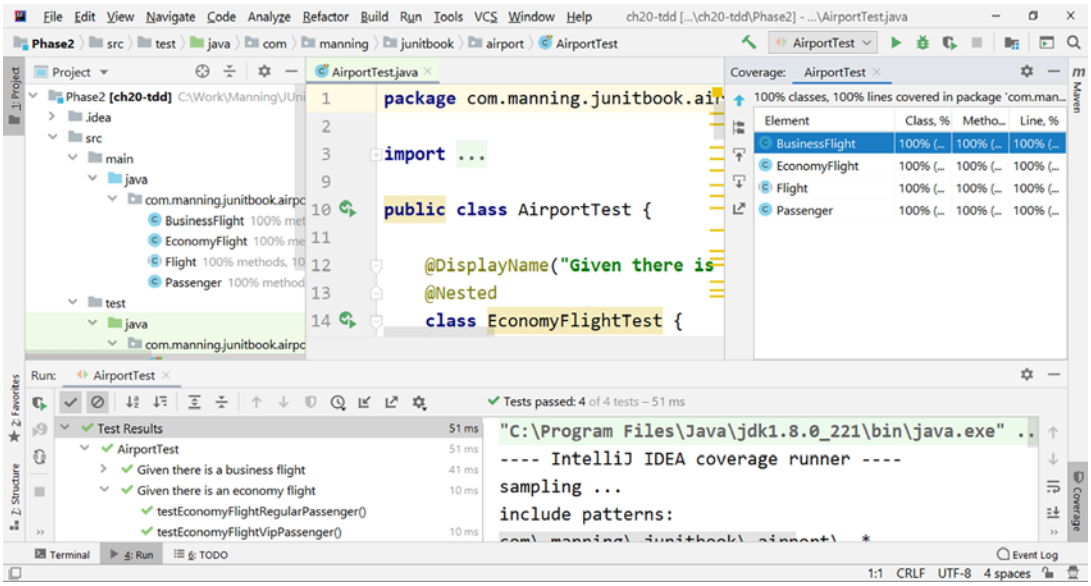


Figure 20.7 Running the economy and business flight tests after refactoring the flight-management application brings us to 100% code coverage.

If we run the tests now, we see that the code coverage is 100% (figure 20.7). So, refactoring the TDD application has helped both to improve the quality of the code and to increase the testing code coverage.

John has covered the flight-management application with tests and refactored it, resulting in better code quality and 100% code coverage. It is time for him to start introducing new features by working with TDD!

20.5 Introducing new features using TDD

After moving the software to TDD and refactoring it, John is responsible for the implementation of new features required by the customer that extend the application policies.

20.5.1 Adding a premium flight

The first new features that John will implement are a new flight type—premium—and policies concerning this flight type. There is a policy for adding a passenger: if the passenger is a VIP, the passenger should be added to the premium flight; otherwise, the request must be rejected (figure 20.8). There is also a policy for removing a passenger: if required, a passenger may be removed from a flight (figure 20.9). (The company is sorry; you may be an important person, but there are rules and restrictions, and the company may be forced to remove you from a flight.)

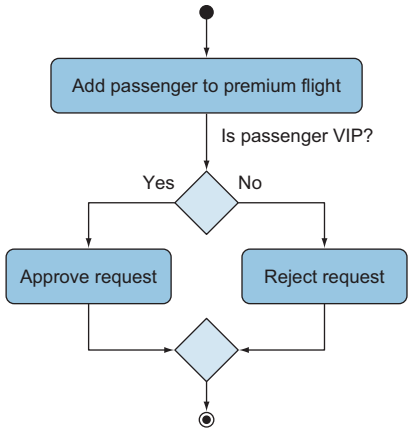


Figure 20.8 The extended business logic of adding a passenger to a premium flight: only VIP passengers are allowed to join.

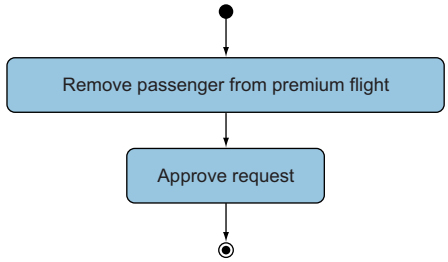


Figure 20.9 The extended business logic of removing a passenger: any type of passenger may be removed from a premium flight.

John realizes that this new feature has similarities to the previous ones. He would like to take increased advantage of working TDD style and do more refactoring—this time, to the tests. This is in the spirit of the *Rule of Three*, as stated by Don Roberts (https://en.wikipedia.org/wiki/Rule_of_three_(computer_programming)):

> *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.*

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 existing tests using the JUnit 5 `@Nested` annotation and then implement the premium flight requirement in a similar way. Following is the refactored `AirportTest` class before moving to the work for the premium flight.

Listing 20.11 Refactored `AirportTest` class

```
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() {
```

1

```
        economyFlight = new EconomyFlight("1");
        mike  = new Passenger("Mike", false);
        james = new Passenger("James", true);
    }

    @Nested
    @DisplayName("When we have a regular passenger")
    class RegularPassenger {

        @Test
        @DisplayName(
           "Then you can add and remove him from an economy flight")
        public void testEconomyFlightRegularPassenger() {
            assertAll(
                      "Verify all conditions for a regular passenger
                       and an economy flight",
                    () -> assertEquals("1", economyFlight.getId()),
                    () -> assertEquals(true,
                      economyFlight.addPassenger(mike)),
                    () -> assertEquals(1,
                      economyFlight.getPassengersList().size()),
                    () -> assertEquals("Mike",
                      economyFlight.getPassengersList()
                                   .get(0).getName()),
                    () -> assertEquals(true,
                      economyFlight.removePassenger(mike)),
                    () -> assertEquals(0,
                      economyFlight.getPassengersList().size())
            );
        }
    }

    @Nested
    @DisplayName("When we have a VIP passenger")
    class VipPassenger {
        @Test
        @DisplayName("Then you can add him but
                      cannot remove him from an economy flight")
        public void testEconomyFlightVipPassenger() {
            assertAll("Verify all conditions for a VIP passenger
                       and an economy flight",
                    () -> assertEquals("1", economyFlight.getId()),
                    () -> assertEquals(true,
              economyFlight.addPassenger(james)),
                    () -> assertEquals(1,
              economyFlight.getPassengersList().size()),
                    () -> assertEquals("James",
              economyFlight.getPassengersList().get(0).getName()),
                    () -> assertEquals(false,
              economyFlight.removePassenger(james)),
                    () -> assertEquals(1,
              economyFlight.getPassengersList().size())
            );

        }
    }
}
```

2

3

4

4

5

5

```
@DisplayName("Given there is a business flight")
@Nested
class BusinessFlightTest {
    private Flight businessFlight;
    private Passenger mike;
    private Passenger james;

    @BeforeEach
    void setUp() {
        businessFlight = new BusinessFlight("2");
        mike = new Passenger("Mike", false);
        james = new Passenger("James", true);
    }

    @Nested
    @DisplayName("When we have a regular passenger")
    class RegularPassenger {

        @Test
        @DisplayName("Then you cannot add or remove him
                      from a business flight")
        public void testBusinessFlightRegularPassenger() {
            assertAll("Verify all conditions for a regular passenger
                       and a business flight",
                      () -> assertEquals(false,
              businessFlight.addPassenger(mike)),
                      () -> assertEquals(0,
              businessFlight.getPassengersList().size()),
                      () -> assertEquals(false,
              businessFlight.removePassenger(mike)),
                      () -> assertEquals(0,
              businessFlight.getPassengersList().size())
            );
        }
    }

    @Nested
    @DisplayName("When we have a VIP passenger")
    class VipPassenger {

        @Test
        @DisplayName("Then you can add him but cannot remove him
                      from a business flight")
        public void testBusinessFlightVipPassenger() {
            assertAll("Verify all conditions for a VIP passenger
                       and a business flight",
                      () -> assertEquals(true,
                         businessFlight.addPassenger(james)),
                      () -> assertEquals(1,
                         businessFlight.getPassengersList().size()),
                      () -> assertEquals(false,
                         businessFlight.removePassenger(james)),
                      () -> assertEquals(1,
                         businessFlight.getPassengersList().size())
            );
        }
    }
}
```

1
2
3
4
5
4
5

In this listing:

- In the existing nested classes `EconomyFlightTest` and `BusinessFlightTest`, John groups the flight and passenger fields, as he would like to add one more testing level and reuse these fields for all tests concerning a particular flight type (1). He initializes these fields before the execution of each test (2).
- He introduces a new nesting level to test different passenger types. He uses the JUnit 5 `@DisplayName` annotation to label the classes in a way that is more expressive and easier to follow (3). All of these labels start with the keyword `When`.
- He labels all existing tests with the help of the JUnit 5 `@DisplayName` annotation (4). All of these labels start with the keyword `Then`.
- He refactors the checking of the conditions by using the `assertAll` JUnit 5 method and grouping all previously existing conditions, which can now be read fluently (5).

This is how John has refactored the existing tests, to facilitate continuing to work in TDD style and to introduce the newly required premium flight business logic. If we run the tests now, we can easily follow the way they work and how they check the business logic (figure 20.10). Any new developer joining the project will find these tests extremely valuable as part of the documentation!

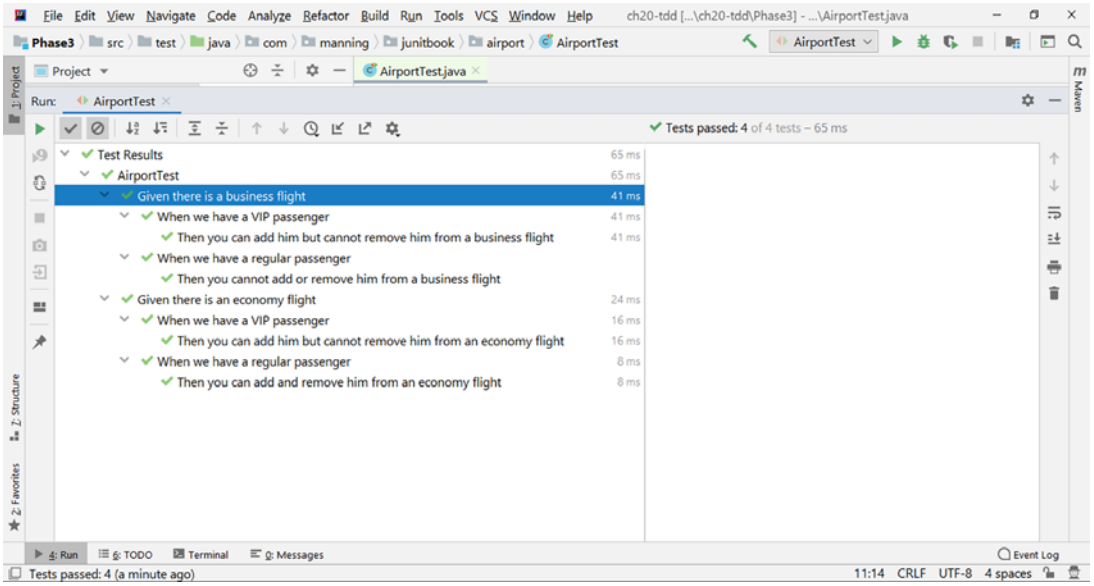


Figure 20.10 Running the refactored `AirportTest` for the economy flight and the business flight allows the developer to follow how the tests are working.

John moves now to the implementation of the `PremiumFlight` class and its logic. He will create `PremiumFlight` as a subclass of `Flight` and override the `addPassenger` and `removePassenger` methods, but they act like stubs—they do not do anything and simply return `false`. Their behavior will be extended later. Working TDD style involves creating the tests first and then the business logic.

Listing 20.12 Initial design of the `PremiumFlight` class

```
public class PremiumFlight extends Flight {

   public PremiumFlight(String id) {
      super(id);
   }

   @Override
   public boolean addPassenger(Passenger passenger) {
      return false;
   }

   @Override
   public boolean removePassenger(Passenger passenger) {
       return false;
   }

}
```

In this listing:

- John declares the `PremiumFlight` class that extends `Flight` (1), and he creates a constructor for it (2).
- He creates the `addPassenger` (3) and `removePassenger` (4) methods as stubs, without any business logic. They simply return `false`.

John now implements the tests according to the premium flight business logic from figures 20.8 and 20.9.

Listing 20.13 Tests for the behavior of `PremiumFlight`

```
public class AirportTest {
    [...]

    @DisplayName("Given there is a premium flight")
    @Nested
    class PremiumFlightTest {
        private Flight premiumFlight;
        private Passenger mike;
        private Passenger james;

        @BeforeEach
        void setUp() {
```

```
            premiumFlight = new PremiumFlight("3");
            mike = new Passenger("Mike", false);
            james = new Passenger("James", true);
        }

        @Nested
        @DisplayName("When we have a regular passenger")
        class RegularPassenger {

            @Test
            @DisplayName("Then you cannot add or remove him
                          from a premium flight")
            public void testPremiumFlightRegularPassenger() {
                assertAll("Verify all conditions for a regular passenger
                           and a premium flight",
                          () -> assertEquals(false,
                                premiumFlight.addPassenger(mike)),
                          () -> assertEquals(0,
                                premiumFlight.getPassengersList().size()),
                          () -> assertEquals(false,
                                premiumFlight.removePassenger(mike)),
                          () -> assertEquals(0,
                                premiumFlight.getPassengersList().size())
                );
            }
        }

        @Nested
        @DisplayName("When we have a VIP passenger")
        class VipPassenger {
            @Test
            @DisplayName("Then you can add and remove him
                          from a premium flight")
            public void testPremiumFlightVipPassenger() {
                assertAll("Verify all conditions for a VIP passenger
                           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 this listing:

- John declares the nested class `PremiumFlightTest` (1) that contains the fields representing the flight and the passengers (2) that are set up before each test (3).

- He creates two classes nested at the second level in `PremiumFlightTest`: `RegularPassenger` ❹ and `VipPassenger` ❾. He uses the JUnit 5 `@DisplayName` annotation to label these classes starting with the keyword `When`.
- He inserts one test in each of the newly added `RegularPassenger` ❺ and `VipPassenger` ❿ classes. He labels these tests with the JUnit 5 `@DisplayName` annotation starting with the keyword `Then`.
- Testing a premium flight and a regular passenger, John uses the `assertAll` method to verify multiple conditions ❻. He checks that he cannot add a passenger to a premium flight and that trying to add a passenger does not change the size of the passenger list ❼. Then, he checks that he cannot remove a passenger from a premium flight and that trying to remove a passenger does not change the size of the passenger list ❽.
- Testing a premium flight and a VIP passenger, John again uses `assertAll` ⓫. He checks that he can add a passenger to a premium flight and that doing so increases the size of the passenger list ⓬. Then, he checks that he can remove a passenger from a premium flight and that doing so decreases the size of the passenger list ⓭.

After writing the tests, John runs them. Remember, he is working TDD style, so tests come first. The result is shown in figure 20.11.

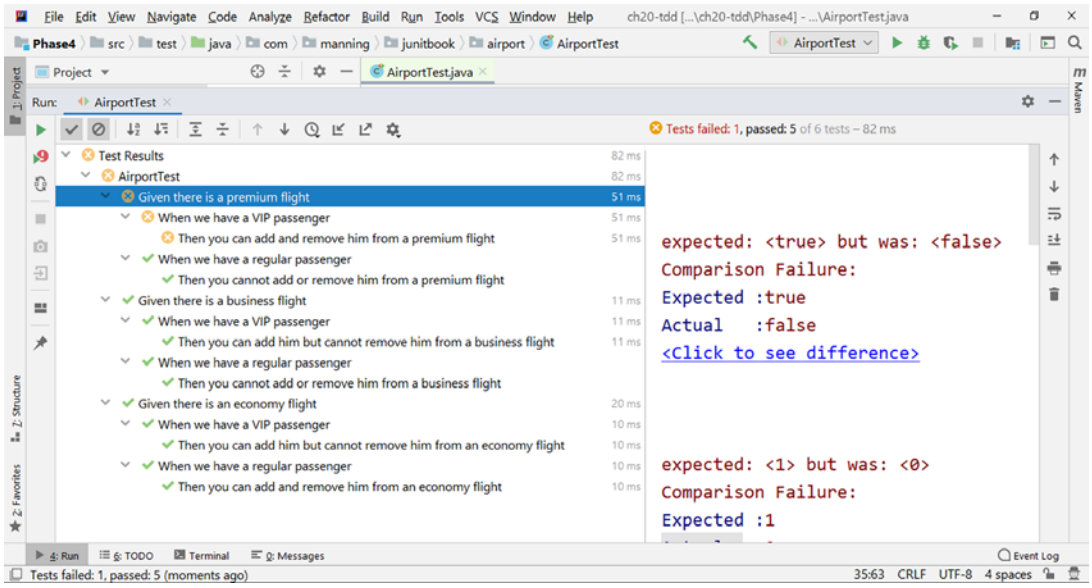


Figure 20.11 Running the newly added tests to check premium flights before the code implementation results in some test failures. We need to understand which behavior to introduce to fix the failing tests.

The fact that one of the tests is failing is not a problem. On the contrary: it is what John expected. Remember, working TDD style means being driven by tests, so we first create the test to fail and then write the piece of code that will make the test pass. But there is another remarkable thing here: the test for a premium flight and a regular passenger is already green. This means the existing business logic (the `addPassenger` and `removePassenger` methods returning `false`) is just enough for this case. John understands that he only has 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 moves back to the `PremiumFlight` class and adds the business logic only for VIP passengers. Driven by tests, he gets straight to the point.

Listing 20.14 `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()) {
         return passengers.add(passenger);          (1)
      }
      return false;
   }

   @Override
   public boolean removePassenger(Passenger passenger) {
        if (passenger.isVip()) {
            return passengers.remove(passenger);    (2)
        }
       return false;
   }

}
```

In this listing:

- John adds a passenger only if the passenger is a VIP (1).
- John removes a passenger only if the passenger is a VIP (2).

The result of running the tests now is shown in figure 20.12. Everything went smoothly and was driven by the tests that guide the developer in writing the code that makes them pass. Additionally, the code coverage is 100%.

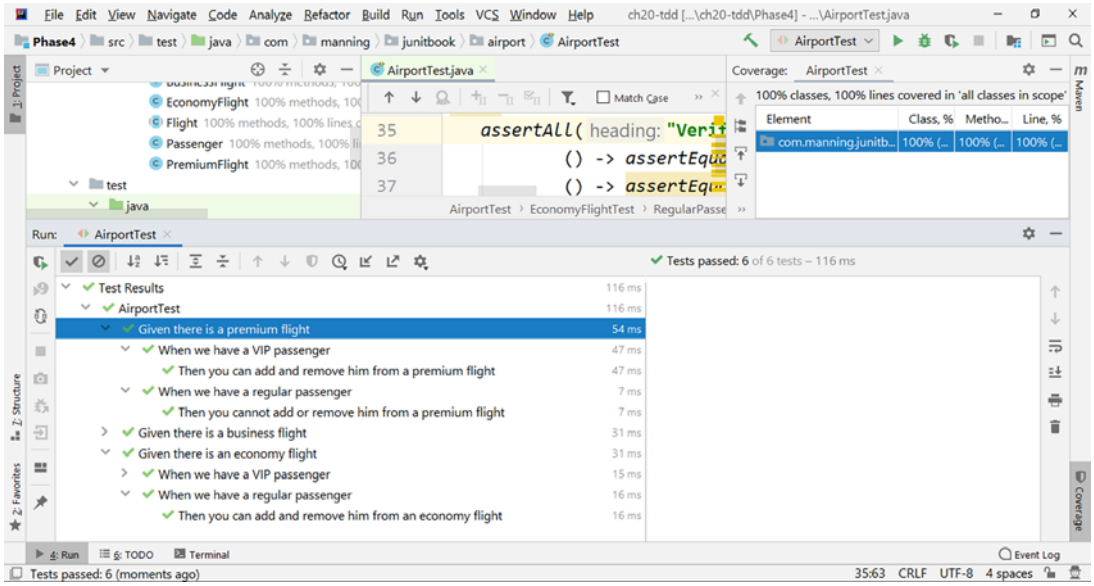


Figure 20.12 Running the full test suite (economy, business, and premium flights) after adding the business logic for `PremiumFlight`: code coverage is 100%.

20.5.2 Adding a passenger only once

Occasionally, on purpose or by mistake, the same passenger has been added to a flight more than once. This has caused problems with managing seats, and these situations must be avoided. John needs to make sure that whenever someone tries to add a passenger, if the passenger 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 will begin 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, as shown in the following listing. 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 this listing:

- John marks the test as `@RepeatedTest` five times (1) and uses the `RepetitionInfo` parameter in it (2).
- Each time a test is executed, he tries to add the passenger the number of times specified by the `RepetitionInfo` parameter (3).
- He performs verifications using the `assertAll` method (4): he checks that the list of passengers is size `1` (5), that the list contains the added passenger (6), and that the passenger is in the first position (7).

If we run the tests, they fail. There is no business logic yet to prevent adding a passenger more than once (figure 20.13).

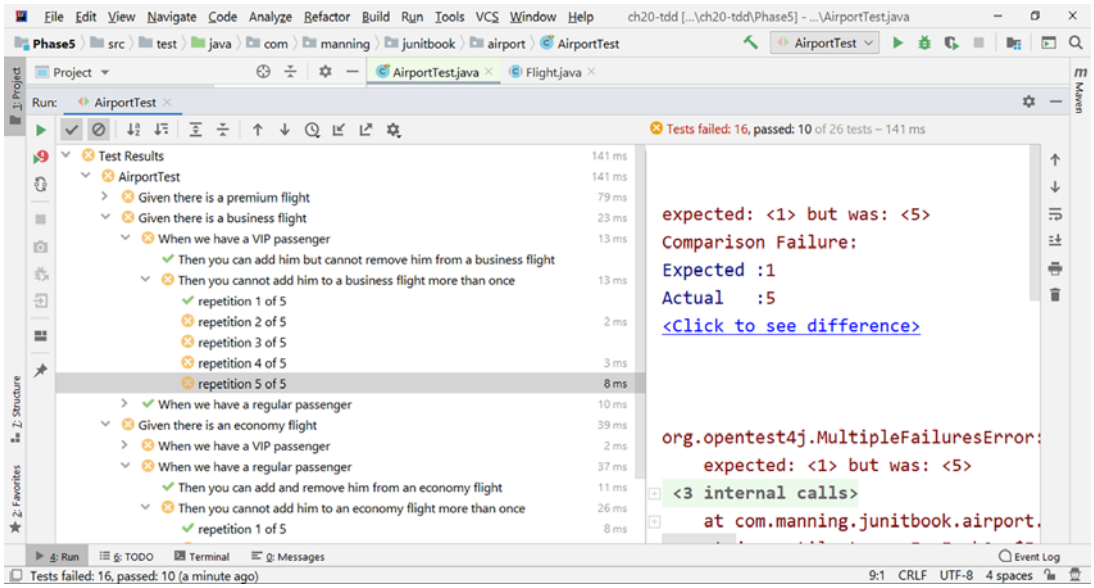


Figure 20.13 Running the tests that check whether a passenger can be added only once to a flight, before implementing the business logic, will result in failure.

To ensure the unicity of the passengers on a flight, John changes the passenger list structure to a set. So, he does some code refactoring that will also propagate across the tests. The `Flight` class changes as shown next.

Listing 20.16 `Flight` class after changing the list of passengers to a set

```
public abstract class Flight {

    [...]
    Set<Passenger> passengers = new HashSet<>();    ❶
    [...]

    public Set<Passenger> getPassengersSet() {      ❷
        return Collections.unmodifiableSet(passengers);   ❸
    }

    [...]
}
```

In this listing, John changes the type and the initialization of the `passengers` attribute to a set ❶, changes the name of the method ❷, and returns an unmodifiable set ❸.

John then creates a new test to check that a passenger can be added only once to a flight.

Listing 20.17 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,
               economyFlight.getPassengersSet().size()),
               () -> assertTrue(
               economyFlight.getPassengersSet().contains(mike)),
               () -> assertTrue(
               new ArrayList<>(economyFlight.getPassengersSet())
                  .get(0).getName().equals("Mike")));
}
```

In this listing, John checks the size of the passengers set ❶, the fact that this set contains the newly added passenger ❷, and that the passenger is in the first position ❸, after constructing a list from the existing set (he needs to do this because a set has no order for the elements).

Running the tests is now successful, with code coverage of 100% (figure 20.14). John has implemented this new feature in TDD style.

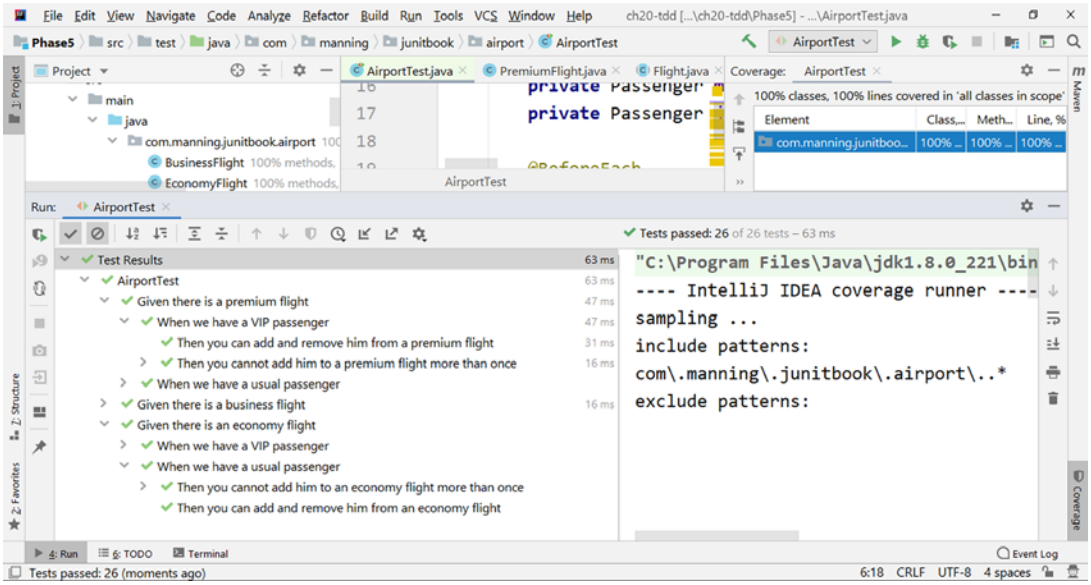


Figure 20.14 Successfully running the entire test suite after implementing the business logic to check that a passenger can be added only once to a flight

The next chapter is dedicated to another software development process that is frequently used today: behavior-driven development (BDD).

Summary

This chapter has covered the following:

- Examining the concept of TDD and demonstrating how it helps us develop safe applications because tests prevent the introduction of bugs into working code and act as part of the documentation
- Preparing a non-TDD 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 we developed
- Implementing new features in by working TDD style, starting by writing tests and then implementing the business logic

21 Behavior-driven development with JUnit 5

This chapter covers

- Analyzing the benefits and challenges of BDD
- Moving a TDD application to BDD
- Developing a BDD application with Cucumber and JUnit 5
- Developing a BDD application with JBehave and JUnit 5

> *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

As we discussed in chapter 20, test-driven development (TDD) is an effective technique that uses unit tests to verify the code. Despite TDD's clear benefits, its usual loop

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

can cause developers to lose the overall picture of the application's business goals. The project may become larger and more complex, the number of unit tests will

increase, and those tests will become harder to understand and maintain. The tests may also be strongly coupled with the implementation. They focus on the unit (the class or the method) that is tested, and the business goals may not be considered.

Starting from TDD, a new technique has been created: *behavior-driven development* (BDD). It focuses on the features themselves and makes sure they work as expected.

21.1 Introducing behavior-driven development

Dan North originated BDD in the mid-2000s. It is a software development technique that starts from the business requirements and goals and transforms them into working features. BDD encourages teams to interact, use concrete examples to communicate how the application must behave, and deliver software that matters, supporting cooperation between stakeholders. TDD helps us build software that works; BDD helps us build software that provides business value. Using BDD, we determine which features the organization really needs and focus on implementing them. We can also discover what users actually need, not just what they ask about.

> **NOTE** 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 discussion of this subject, we recommend *BDD in Action* by John Ferguson Smart; the second edition of the book is in development at the time of writing this chapter (Manning, www.manning.com/books/bdd-in-action-second-edition).

Communication between people involved in the same project may lead to problems and misunderstandings. Usually, the flow works this way:

1. The customer communicates to the business analyst their understanding of the functionality of a feature.
2. The business analyst builds the requirements for the developer, describing the way the software must work.
3. The developer creates the code based on the requirements and writes unit tests to implement the new feature.
4. The tester creates the test cases based on the requirements and uses them to verify the way the new feature works.

But it is possible that the information may be misunderstood, modified, or ignored—and thus the new feature may not do exactly what was initially expected. We'll analyze how things evolve when a new feature is introduced.

21.1.1 Introducing a new feature

The business analyst talks with the customer to decide what software features 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," and "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," or "Find the quickest route between source and destination."

Stories are defined through concrete examples. These examples become the *acceptance criteria* for a story. Acceptance criteria are expressed BDD style through the keywords `Given`, `When`, and `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 flight-management application, one business goal that we can formulate is "Increase sales by providing overall higher quality 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 express what will make the stakeholder agree that the application is working as expected.

In BDD, acceptance criteria are defined using the `Given/When/Then` keywords:

```
Given <a context>
When <an action occurs>
Then <expect a result>
```

Here's 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

Here are some benefits of the BDD approach:

- *Addresses user needs*—Users care less about the implementation and are mainly interested in application functionality. Working BDD style, we get closer to addressing these needs.
- *Provides clarity*—Scenarios clarify what software should do. They are described in simple language that is easy for technical and nontechnical people to understand. Ambiguities can be clarified by analyzing the scenario or by adding another scenario.
- *Supports change*—Scenarios represent part of the software documentation: it's living documentation, as it evolves simultaneously with the application. It also helps locate incoming changes. Automated acceptance tests 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 prevents the introduction of features that are not useful to the project. We can also prioritize functionalities.
- *Reduces costs*—Prioritizing the importance of functionalities and avoiding unnecessary ones will prevent wasting resources and concentrate these resources on doing exactly what is needed.

The challenges of BDD are that it requires engagement, strong collaboration, interaction, direct communication, and constant feedback. This may be a challenge for some people and, in the context of present-day globalization and distributed teams, may require language skills and managing time zones.

21.2 Working BDD style with Cucumber and JUnit 5

In chapter 20, John, a programmer at Tested Data Systems, worked TDD style to develop the flight-management application to a stage where it can work with three types of flights: economy, business, and premium. In addition, he implemented the

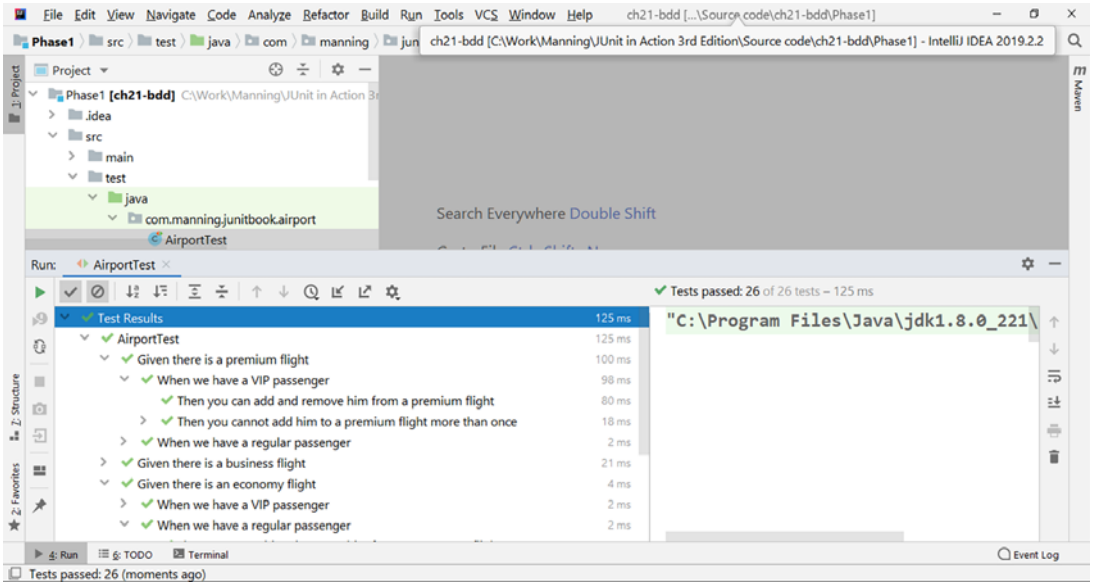


Figure 21.1 Successfully running the tests of the TDD flight-management application. Users can follow the execution of the annotated methods and read the scenarios.

requirement that a passenger can be added only once to a flight. The functionality of the application can be quickly reviewed by running the tests (figure 21.1).

John has already introduced, in a discrete way, a first taste of the BDD way of working. We can easily read how the application works by following the tests using the `Given`, `When`, and `Then` keywords. In this chapter, John will move the application to BDD with Cucumber and also introduce more new features.

21.2.1 Introducing Cucumber

Cucumber (https://cucumber.io/) is a BDD testing framework. It describes application scenarios in plain English using a language called Gherkin. Cucumber is easy for stakeholders to read and understand and allows automation.

The main capabilities of Cucumber are as follows:

- 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 describing the success or failure of each scenario.

The main capabilities of Gherkin are as follows:

- Gherkin defines simple grammar rules that allow Cucumber to understand plain English text.

- Gherkin documents the behavior of the system. The requirements are always up to date, as they are provided through scenarios that represent living specifications.

Cucumber ensures that technical and nontechnical people 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 looks like this:

```
Given there is an economy flight
When we have a regular passenger
Then you can add and remove him from an economy flight
```

Again, notice the `Given/When/Then` keywords for describing a scenario, which we introduced in our previous work with JUnit 5. We'll no longer use them just for labeling: Cucumber interprets sentences starting with these keywords and generates methods that it annotates using the annotations `@Given`, `@When`, and `@Then`.

The acceptance tests are written in Cucumber feature files. A *feature file* is an entry point to the Cucumber tests; in the file, we describe our tests in Gherkin. A feature file can contain one or many scenarios.

John makes plans for starting to work with Cucumber in the project. He will first introduce the Cucumber dependencies into the existing Maven configuration. He will create a Cucumber feature and generate the skeleton of the Cucumber tests. Then, he will move the existing JUnit 5 tests to fill in this Cucumber-generated test skeleton.

Listing 21.1 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 this listing, John introduces the two needed Maven dependencies: `cucumber-java` and `cucumber-junit`.

21.2.2 Moving a TDD feature to Cucumber

Now John will begin creating Cucumber features. He follows the Maven standard folder structure and introduces the features into the test/resources folder. He creates the test/resources/features folder and, in it, creates the passengers_policy.feature file (figure 21.2).

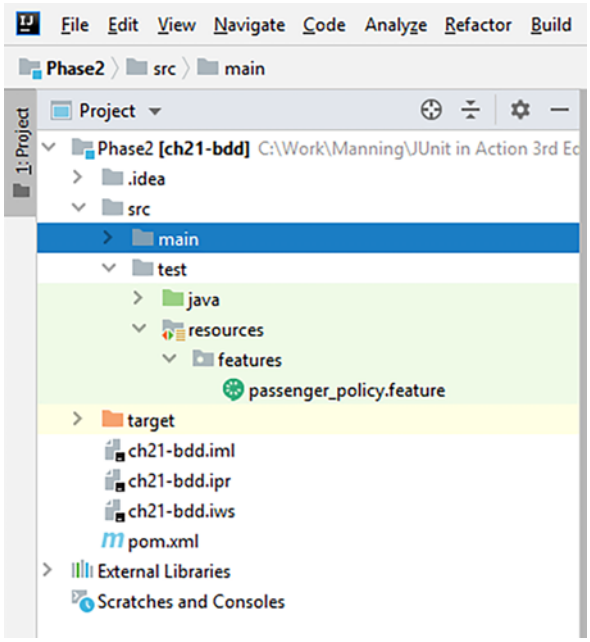


Figure 21.2 The new Cucumber passenger_policy.feature file is created in the test/resources/features folder, following Maven rules.

John follows the Gherkin syntax and introduces a feature named `Passengers Policy`, together with a short description of what it should do. Then he follows the Gherkin syntax to write scenarios.

Listing 21.2 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
```

The keywords `Feature`, `Scenario`, `Given`, `When`, `Then`, and `And` are highlighted. Right-clicking this feature file shows the option to run it directly (figure 21.3).

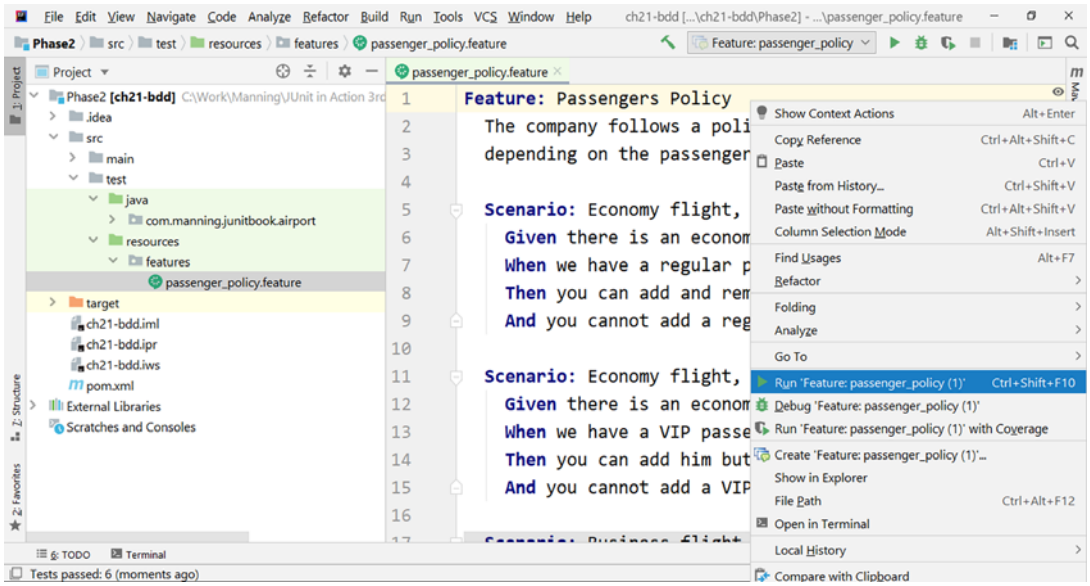


Figure 21.3 Directly running the passengers_policy.feature file by right-clicking the file

This is possible only if two requirements are fulfilled. First, the appropriate plugins must be activated. To do this in IntelliJ, go to File > Settings > Plugins and install the Cucumber for Java and Gherkin plugins (figures 21.4 and 21.5).

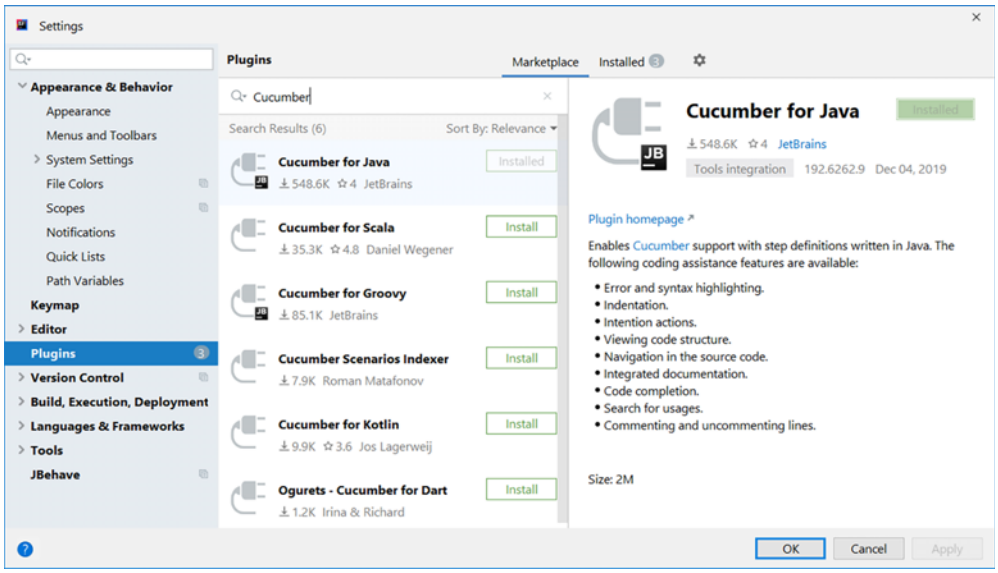


Figure 21.4 Installing the Cucumber for Java plugin from the File > Settings > Plugins menu

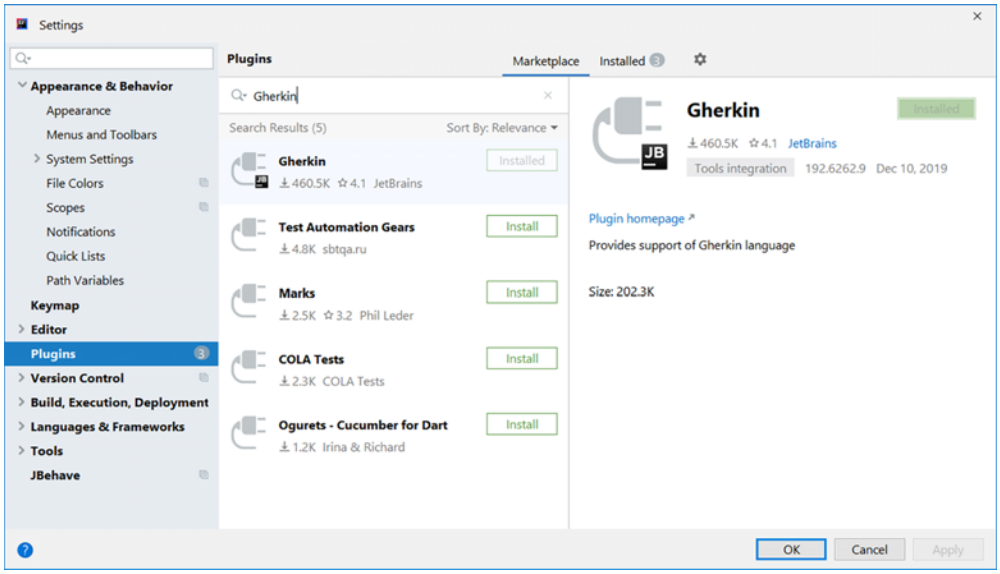


Figure 21.5 Installing the Gherkin plugin from the File > Settings > Plugins menu

Second, we must configure the way the feature is run. Go to Run > Edit Configurations, and set the following options (figure 21.6):

- Main Class: `cucumber.api.cli.Main`

- Glue (the package where step definitions are stored): `com.manning.junit-book.airport`
- Feature or Folder Path: the test/resources/features folder we have created
- Working Directory: the project folder

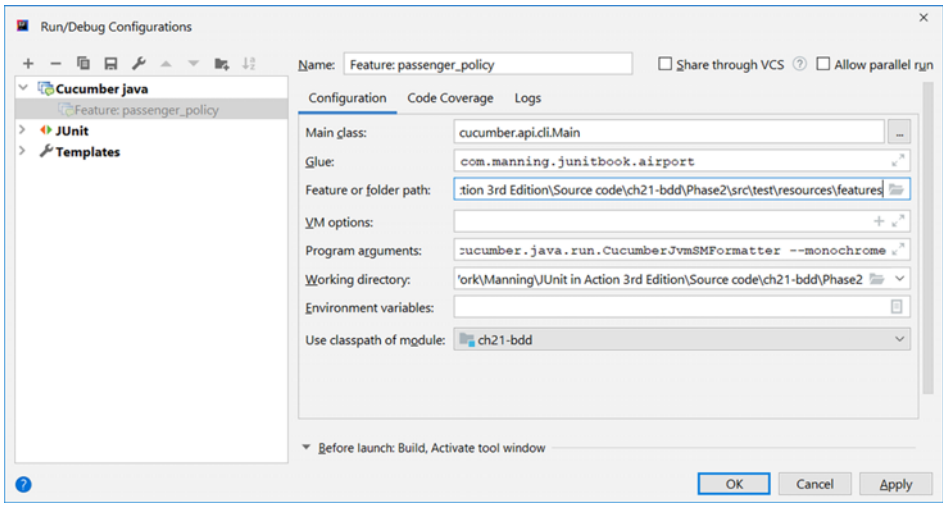


Figure 21.6 Setting the feature configuration by filling in the Main Class, Glue, Feature or Folder Path, and Working Directory fields

Running the feature directly generates the skeleton of the Java Cucumber tests (figure 21.7).

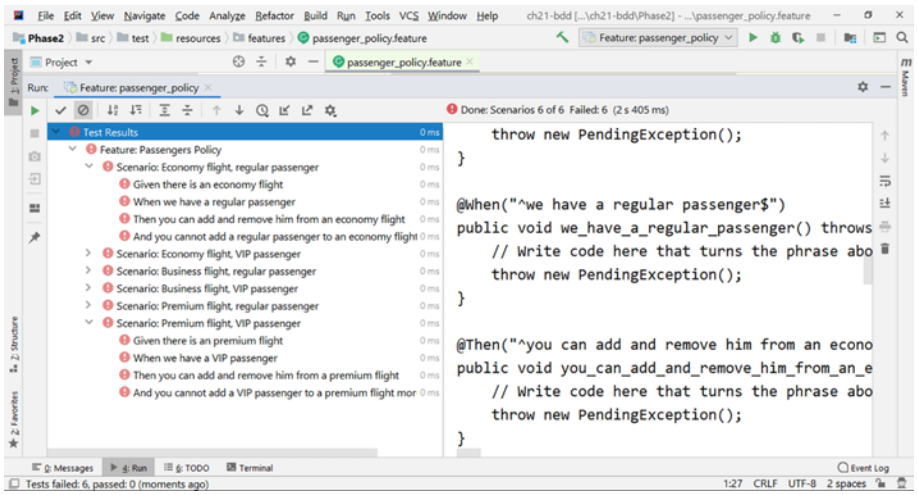


Figure 21.7 Getting the skeleton of the `Passengers Policy` feature by directly running the feature file. The annotated methods are executed to verify the scenarios.

John will now create a new Java class in the test/java folder, in the `com.manning.junitbook.airport` package. This class will be named `PassengersPolicy` and, at first, will contain the test skeleton (listing 21.3). The execution of such a test follows 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 Initial `PassengersPolicy` class

```
public class PassengerPolicy {
   @Given("^there is an economy flight$")                          (1)
   public void there_is_an_economy_flight() throws Throwable {
    // Write code here that turns the phrase above into concrete actions
    throw new PendingException();                                  (2)
   }

   @When("^we have a regular passenger$")                          (3)
   public void we_have_a_regular_passenger() throws Throwable {
   // Write code here that turns the phrase above into concrete actions
      throw new PendingException();                                (4)
   }

(5)@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
      throw new PendingException();                                (6)
   }

   [...]

}
```

In this listing:

- The Cucumber plugin generates a method annotated with `@Given("^there is an economy flight$")`, meaning this method is executed when the step `Given there is an economy flight` from the scenario is executed (1).
- The plugin generates a method stub to be implemented with the code addressing the step `Given there is an economy flight` from the scenario (2).
- The plugin generates a method annotated with `@When("^we have a regular passenger$")`, meaning this method is executed when the step `When we have a regular passenger` from the scenario is executed (3).
- The plugin generates a method stub to be implemented with the code addressing the step `When we have a regular passenger` from the scenario (4).
- The plugin generates a method annotated with `@Then("^you can add and remove him from an economy flight$")`, meaning this method is executed

when the step `Then you can add and remove him from an economy flight` from the scenario is executed (5).

- The plugin generates a method stub to be implemented with the code addressing the step `Then you can add and remove him from an economy flight` from the scenario (6).
- 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 translates 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;
    private Passenger mike;
    [...]

    @Given("^there is an economy flight$")
    public void there_is_an_economy_flight() throws Throwable {
        economyFlight = new EconomyFlight("1");
    }

    @When("^we have a regular passenger$")
    public void we_have_a_regular_passenger() throws Throwable {
        mike  = new Passenger("Mike", false);
    }

    @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()),
          () -> assertEquals(true, economyFlight.addPassenger(mike)),
          () -> assertEquals(1,
                  economyFlight.getPassengersSet().size()),
          () ->
            assertTrue(economyFlight.getPassengersSet().contains(mike)),
          () -> assertEquals(true, economyFlight.removePassenger(mike)),
          () -> assertEquals(0, economyFlight.getPassengersSet().size())
        );
    }
    [...]
}
```

In this listing:

- John declares the instance variables for the test, including `economyFlight`, and `mike` as a `Passenger` (1).
- He writes the method corresponding to the `Given there is an economy flight` business logic step (2) by initializing `economyFlight` (3).

- He writes the method corresponding to the `When we have a regular passenger` business logic step (4) by initializing the regular passenger `mike` (5).
- He writes the method corresponding to the `Then you can add and remove him from an economy flight` business logic step (6) by checking all the conditions using the `assertAll` JUnit 5 method, which can now be read fluently (7).
- The rest of the methods are implemented in a similar way; we have covered the `Given`, `When`, and `Then` steps of one scenario.

To run the Cucumber tests, John will need a special class. The name of the class could be anything; he chooses `CucumberTest`.

Listing 21.5 `CucumberTest` class

```
[...]
@RunWith(Cucumber.class)
@CucumberOptions(
   plugin = {"pretty"},
   features = "classpath:features")
public class CucumberTest {

   /**
    * This class should be empty, step definitions should be in separate
     classes.
    */

}
```

In this listing:

- John annotates this class with `@RunWith(Cucumber.class)` (1). Executing it like any JUnit test class runs all the 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` annotation (2) provides the plugin option (3) that is used to specify different formatting options for the output reports. Using `"pretty"`, the Gherkin source is printed with additional colors (figure 21.8). Other plugin options include `"html"` and `"json"`, but `"pretty"` is appropriate for now. And the `features` option (4) helps Cucumber locate the feature file in the project folder structure. It looks 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 we have kept the test functionality that existed before moving to Cucumber.

There is another advantage of moving to BDD. Comparing the length of the pre-Cucumber `AirportTest` class, which has 207 lines, with the `PassengersPolicy` class, which has 157 lines, the testing code is only 75% of the pre-Cucumber size but has the same 100% coverage. Where does this gain come from? Remember that the

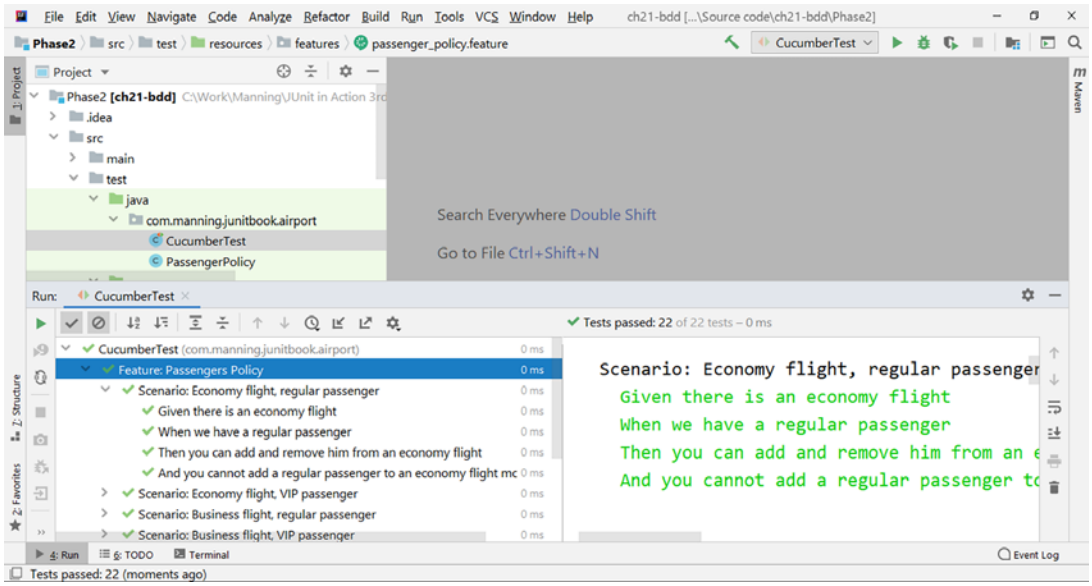


Figure 21.8 Running `CucumberTest`. The Gherkin source is pretty-printed, successful tests are displayed in green, and the code coverage is 100%.

`AirportTest` file contained seven classes on three levels: `AirportTest` at the top level; `EconomyFlightTest` and `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 when we only had JUnit 5. With Cucumber, each step is implemented only once. 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 bonus points that are awarded to passengers. The specifications for calculating bonus points consider the mileage, meaning the distance traveled by each passenger. The bonus is calculated for all of the passenger's flights and depends on a factor: the mileage is divided by 10 for VIP passengers and by 20 for regular passengers (figure 21.9).

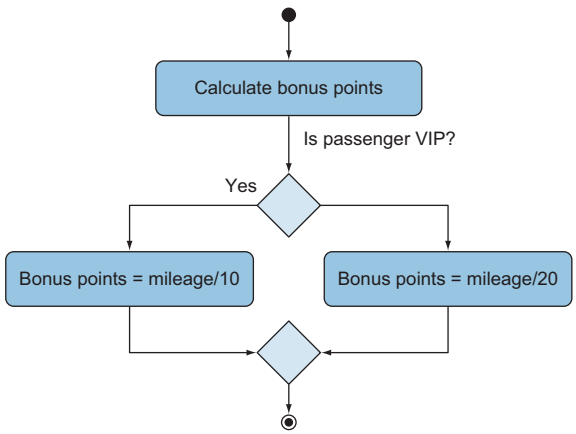


Figure 21.9 The business logic of awarding bonus points: the mileage is divided by 10 for VIP passengers and by 20 for regular passengers.

John moves to the BDD scenarios, tests, and implementation. He defines the scenarios for awarding bonus points (listing 21.6) and generates the Cucumber tests that describe the scenarios. They are expected to fail at first. Then he will 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
    Given we have a regular passenger with a mileage
    When the regular passenger travels <mileage1> and <mileage2>
                                                 and <mileage3>
    Then the bonus points of the regular passenger should be <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 Outline: 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 should be <points>

    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 |
```

In this listing:

- We introduce a new capability of Cucumber: `Scenario Outline` (1). With `Scenario Outline`, values do not need to be hardcoded in the step definitions.
- Values are replaced with parameters in the step definition itself—you can see `<mileage1>`, `<mileage2>`, `<mileage3>`, and `<points>` as parameters (2).
- The effective values are defined in the `Examples` table at the end of the `Scenario Outline` (3). The first row in the first table defines the values of three 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 and has the advantages that the values are kept in the scenarios and can be easily understood by everyone.

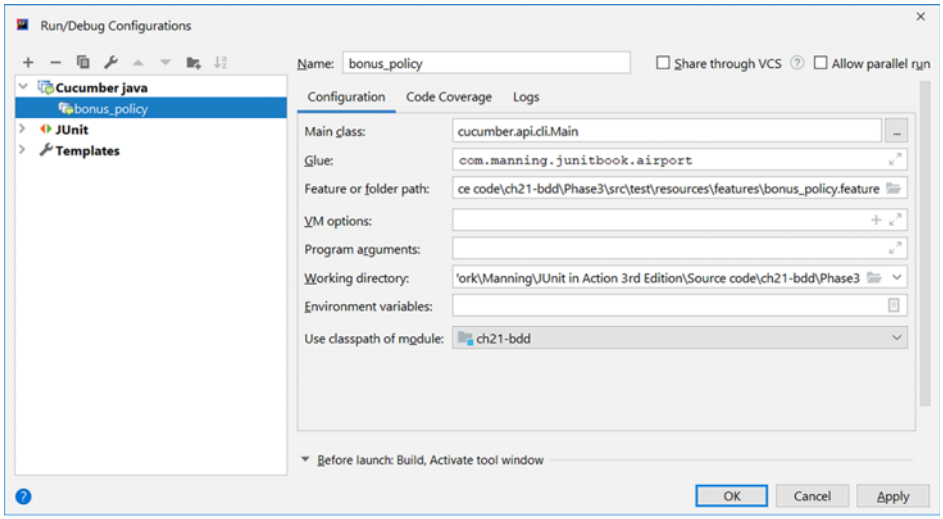


Figure 21.10 Setting the configuration for the new `Bonus_Policy` feature by filling in the Main Class, Glue, Feature or Folder Path, and Working Directory fields

To configure the way the feature is run, choose Run > Edit Configurations, and set the following options (figure 21.10):

- Main Class: `cucumber.api.cli.Main`
- Glue : `com.manning.junitbook.airport`
- Feature or Folder Path: test/resources/features/bonus_policy.feature
- Working Directory: the project folder

Running the feature directly generates the skeleton of the Java Cucumber tests (figure 21.11).

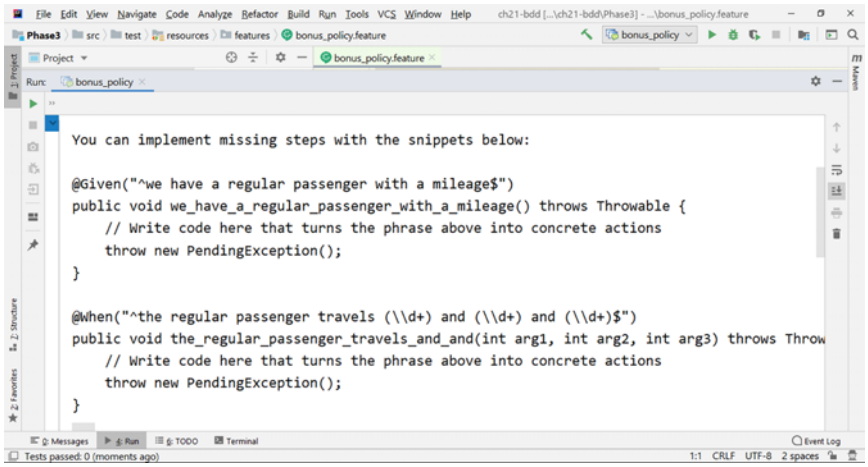


Figure 21.11 Getting the skeleton of the `Bonus Policy` feature by directly running the feature file

John now creates a new Java class in the test/java folder, in the `com.manning.junitbook.airport` package. This class is named `BonusPolicy` and, at first, contains the test skeleton (listing 21.7). Executing this test will follow the scenarios described in the bonus_policy.feature file.

Listing 21.7 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()
           throws Throwable {
    // Write code here that turns the phrase above into concrete actions
    throw new PendingException();
   }

   @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 {
    // Write code here that turns the phrase above into concrete actions
    throw new PendingException();
   }

   @Then("^the bonus points of the regular passenger should be (\\d+)$")
   public void the_bonus_points_of_the_regular_passenger_should_be
                                                          (int arg1)
       throws Throwable {
    // Write code here that turns the phrase above into concrete actions
    throw new PendingException();
   }
   [...]

}
```

(1) (2) (3) (4) (5) (6)

In this listing:

- The Cucumber plugin generates a method annotated with `@Given("^we have a regular passenger with a mileage$")`, meaning this method is executed when the step `Given we have a regular passenger with a mileage` from the scenario is executed (1).
- The plugin generates a method to be implemented with the code addressing the step `Given we have a regular passenger with a mileage` from the scenario (2).
- The plugin generates a method annotated with `@When("^the regular passenger travels (\\d+) and (\\d+) and (\\d+)$")`, meaning this method is executed when the step `When the regular passenger travels <mileage1> and <mileage2> and <mileage3>` from the scenario is executed (3).
- The plugin generates a method to be implemented with the code addressing the step `When the regular passenger travels <mileage1> and <mileage2> and <mileage3>` from the scenario (4). This method has three parameters corresponding to the three different mileages.

- The plugin generates a method annotated with `@Then("^the bonus points of the regular passenger should be (\\d+)$")`, meaning this method is executed when the step `Then the bonus points of the regular passenger should be <points>` from the scenario is executed (5).
- The plugin generates a method to be implemented with the code addressing the step `Then the bonus points of the regular passenger should be <points>` from the scenario (6). 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.

Next, John creates 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, and then implement the methods and make the tests pass.

Listing 21.8 `Mileage` class with no implementation of the methods

```
public class Mileage {

    public static final int VIP_FACTOR = 10;
    public static final int REGULAR_FACTOR = 20;

    private Map<Passenger, Integer> passengersMileageMap =
                                    new HashMap<>();
    private Map<Passenger, Integer> passengersPointsMap =
                                    new HashMap<>();

    public void addMileage(Passenger passenger, int miles) {

    }

    public void calculateGivenPoints() {

    }

}
```

(1) (2) (3) (4)

In this listing:

- John declares the `VIP_FACTOR` and `REGULAR_FACTOR` constants corresponding to the factor by which the mileage is divided for each type of passenger in order to get the bonus points (1).
- He declares `passengersMileageMap` and `passengersPointsMap`, two maps having the passenger as a key and keeping as a value the mileage and points for that passenger, respectively (2).
- He declares the `addMileage` method, which populates `passengersMileageMap` with the mileage for each passenger (3). The method does not do anything for now; it will be written later to fix the tests.

- He declares the `calculateGivenPoints` method, which populates the `passengersPointsMap` with the bonus points for each passenger (4). Again, it will be written later to fix the tests.

John now turns his attention to writing the unimplemented tests from the `BonusPolicy` class to follow the business logic of this feature.

Listing 21.9 Business logic of the steps from `BonusPolicy`

```
public class BonusPolicy {
    private Passenger mike;                                          (1)
    private Mileage mileage;
    [...]

    @Given("^we have a regular passenger with a mileage$")           (2)
    public void we_have_a_regular_passenger_with_a_mileage()
                throws Throwable {
        mike   = new Passenger("Mike", false);                       (3)
        mileage  = new Mileage();
    }

    @When("^the regular passenger travels (\\d+) and (\\d+) and (\\d+)$")   (4)
    public void the_regular_passenger_travels_and_and(int mileage1, int
                mileage2, int mileage3) throws Throwable {
        mileage.addMileage(mike, mileage1);                          (5)
        mileage.addMileage(mike, mileage2);
        mileage.addMileage(mike, mileage3);
    }

    @Then("^the bonus points of the regular passenger should be (\\d+)$")   (6)
    public void the_bonus_points_of_the_regular_passenger_should_be
                 (int points) throws Throwable {
        mileage.calculateGivenPoints();                              (7)
        assertEquals(points,                                         (8)
           mileage.getPassengersPointsMap().get(mike).intValue());
    }
    [...]
}
```

In this listing:

- John declares the instance variables for the test, including `mileage`, and `mike` as a `Passenger` (1).
- He writes the method corresponding to the `Given we have a regular passenger with a mileage` business logic step (2) by initializing the passenger and the mileage (3).
- He writes the method corresponding to the `When the regular passenger travels <mileage1> and <mileage2> and <mileage3>` business logic step (4) by adding mileages to the regular passenger `mike` (5).
- He writes the method corresponding to the `Then the bonus points of the regular passenger should be <points>` business logic step (6) by calculating the given points (7) and checking that the calculated value is as expected (8).

- 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 point tests now, they fail (figure 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). In fact, we get a `NullPointerException` at (8) in listing 21.9: the points map does not exist yet, and the `Mileage` class is not yet implemented. John moves on to the implementation of the two remaining business logic methods from the `Mileage` class (`addMileage` and `calculateGivenPoints`).

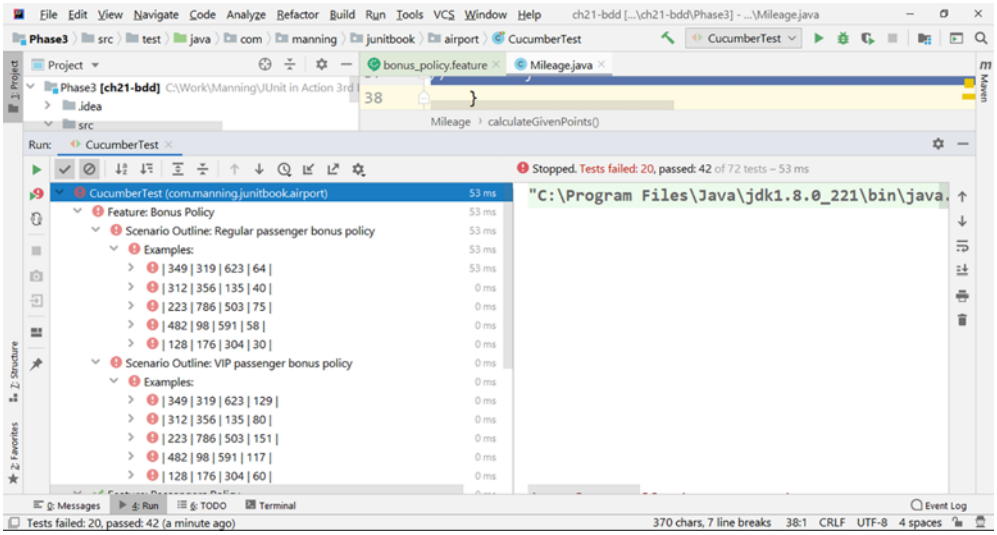


Figure 21.12 The bonus point tests fail when they are run before the business logic is implemented.

Listing 21.10 Implementing the 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);
```

(1) (2) (3) (4) (5)

```
            } else {
                passengersPointsMap.put(passenger,
                    passengersMileageMap.get(passenger)/ REGULAR_FACTOR);   ❻
            }
        }
    }
```

In this listing:

- In the `addMileage` method, John checks whether `passengersMileageMap` already contains a passenger ❶. If that passenger already exists, he adds the mileage to the passenger ❷; otherwise, he creates a new entry in the map having that passenger as a key and the miles as the initial value ❸.
- In the `calculateGivenPoints` method, he browses the passengers set ❹ and, for each passenger, if the passenger is a VIP, calculates the bonus points by dividing the mileage by the VIP factor ❺. Otherwise, he calculates the bonus points by dividing the mileage by the regular factor ❻.

Running the bonus point tests through `CucumberTest` is now successful. The results are nicely displayed, as shown in figure 21.13. John has successfully implemented the bonus policy feature while working BDD style with JUnit 5 and Cucumber.

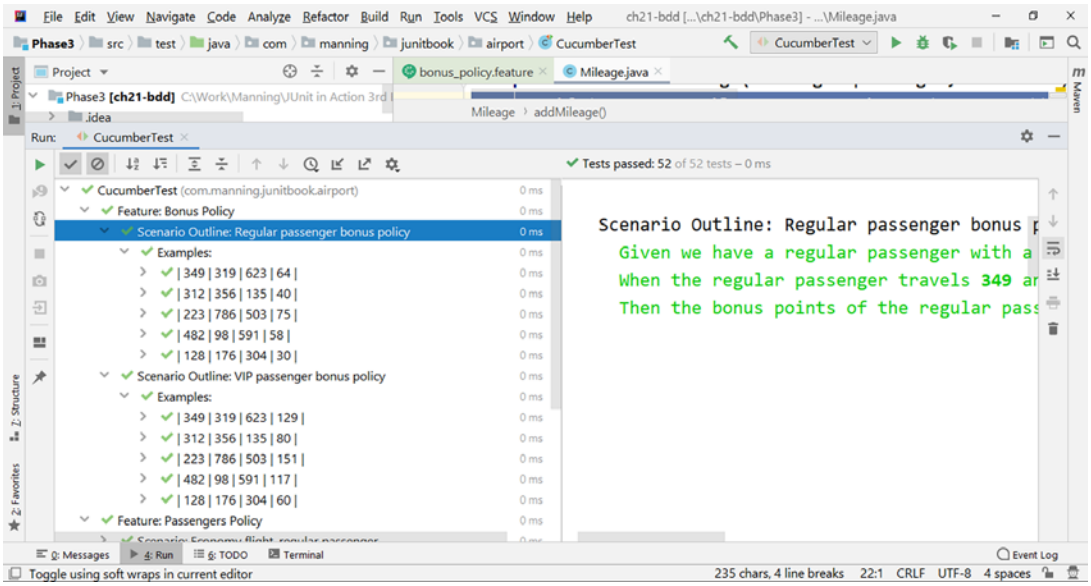


Figure 21.13 The bonus point tests succeed after the business logic is implemented.

21.3 Working BDD style with JBehave and JUnit 5

There are a few alternatives when choosing a BDD framework. In addition to Cucumber, we'll take a look at another very popular framework: JBehave.

21.3.1 Introducing JBehave

JBehave is a BDD testing framework that allows us to write stories in plain text that can be understood by everyone involved in the project. Through the stories, we can define scenarios that express the desired behavior.

Like other BDD frameworks, JBehave has its own terminology:

- *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*—Defined using the classic BDD keywords: `Given`, `When`, and `Then`

21.3.2 Moving a TDD feature to JBehave

With the help of JBehave, John would like to implement the same features and tests that he implemented with Cucumber. This will allow him to make some comparisons between the two BDD frameworks and decide which one to use.

John begins by introducing the JBehave dependency into the Maven configuration. He will first create a JBehave story, generate the test skeleton, and fill it in.

Listing 21.11 JBehave dependency added to the pom.xml file

```
<dependency>
   <groupId>org.jbehave</groupId>
   <artifactId>jbehave-core</artifactId>
   <version>4.1</version>
</dependency>
```

Next, John installs the plugins for IntelliJ. He goes to File > Settings > Plugins > Browse Repositories, types `JBehave`, and chooses JBehave Step Generator and JBehave Support (figure 21.14).

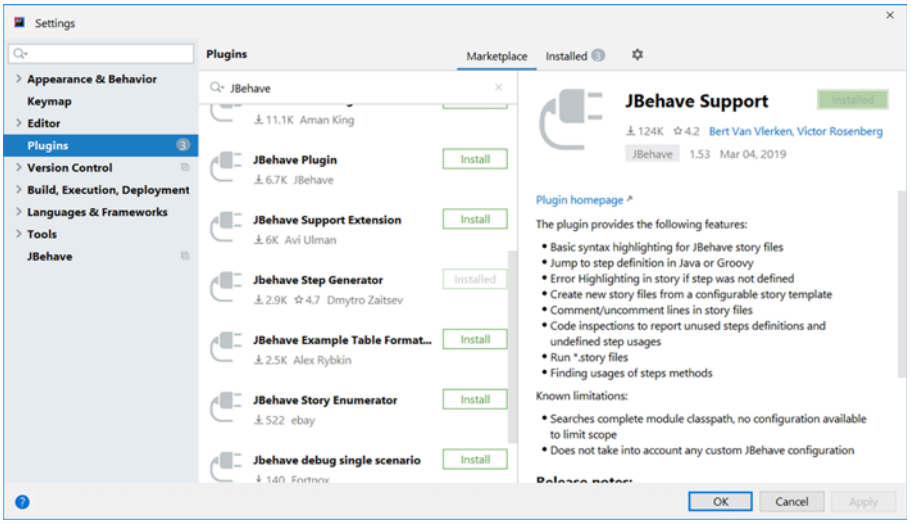


Figure 21.14 Installing the JBehave for Java plugins from the File > Settings > Plugins menu

John will now start creating the story. He follows the Maven standard folder structure and introduces the stories into the test/resources folder. He creates the folder com/manning/junitbook/airport and inserts the passengers_policy_story.story file. He also creates, in the test folder, the `com.manning.junitbook.airport` package containing the `PassengersPolicy` class (figure 21.15).

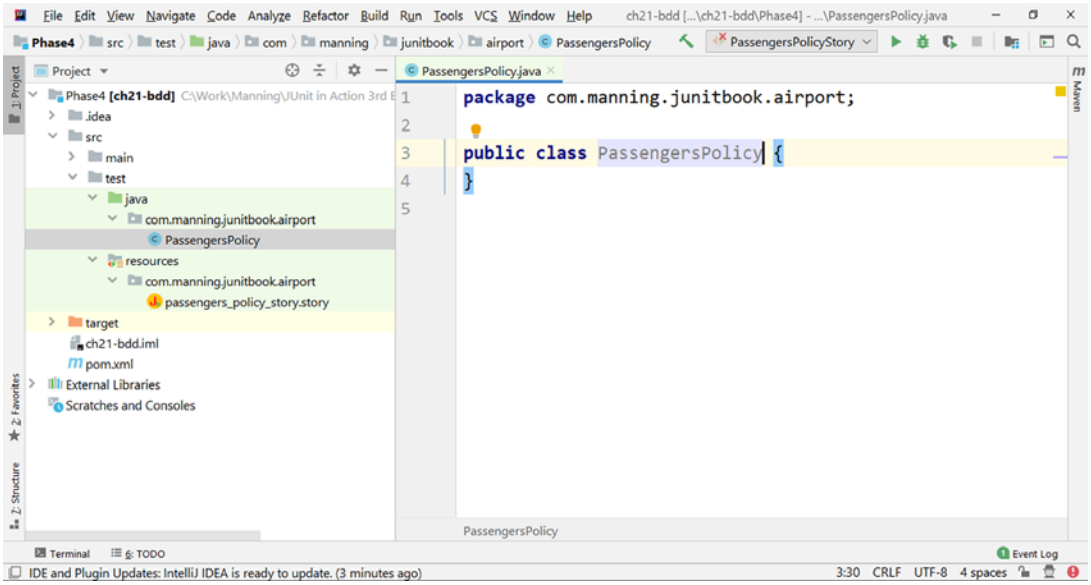


Figure 21.15 The newly added `PassengersPolicy` class corresponds to the story file found in the test/resources/com/manning/junitbook/airport folder.

The story contains meta-information about itself, the narrative (what it intends to do), and the scenarios.

Listing 21.12 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
```

To generate the steps into a Java file, John places the cursor on any not-yet-created test step (underlined in red) and presses Alt-Enter (figure 21.16).

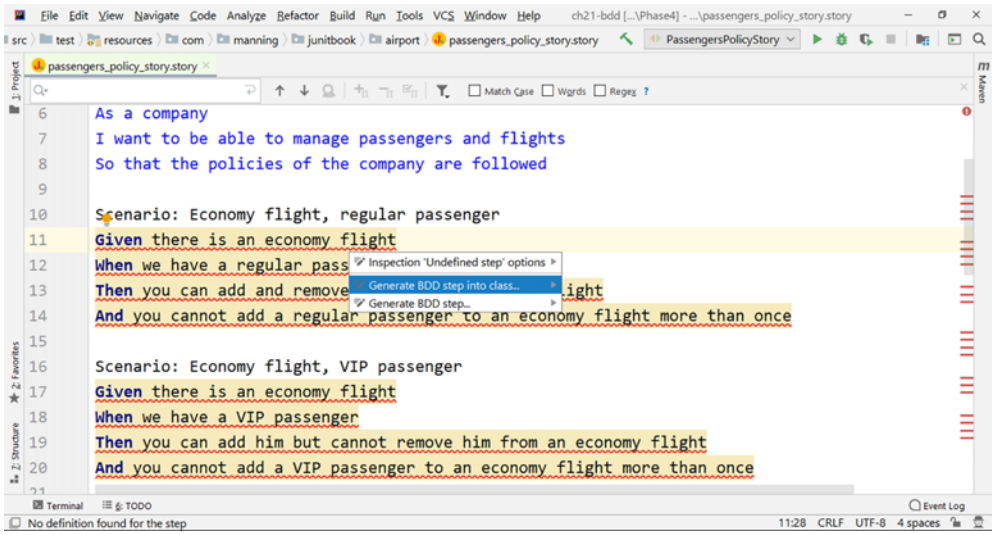


Figure 21.16 Pressing Alt-Enter and generating the BDD steps into a class

He generates all the steps in the newly created `PassengersPolicy` class (figure 21.17). The skeleton needs all the tests filled in.

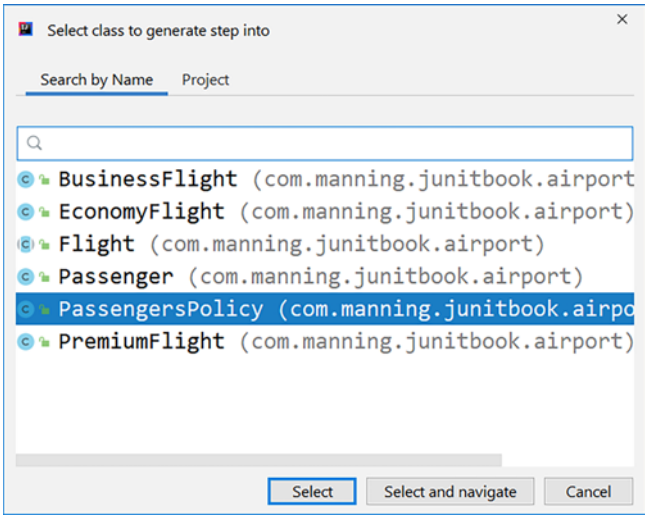


Figure 21.17 Choosing `PassengersPolicy` as the class in which to generate the steps of the story

The JBehave `PassengersPolicy` class is shown in the following listing with the tests skeleton filled in.

Listing 21.13 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 now implements the tests according to the business logic. He writes the code corresponding to each step defined through a method.

Listing 21.14 Implemented tests from `PassengersPolicy`

```
public class PassengersPolicy {
    private Flight economyFlight;                                    (1)
    private Passenger mike;
    [...]

    @Given("there is an economy flight")                             (2)
    public void givenThereIsAnEconomyFlight() {                      (3)
        economyFlight = new EconomyFlight("1");
    }

    @When("we have a regular passenger")                             (4)
    public void whenWeHaveARegularPassenger() {                      (5)
        mike  = new Passenger("Mike", false);
    }

    @Then("you can add and remove him from an economy flight")       (6)
    public void thenYouCanAddAndRemoveHimFromAnEconomyFlight() {
        assertAll("Verify all conditions for a regular passenger
                   and an economy flight",                           (7)
            () -> assertEquals("1", economyFlight.getId()),
            () -> assertEquals(true,
                  economyFlight.addPassenger(mike)),
            () -> assertEquals(1,
                  economyFlight.getPassengersSet().size()),
            () -> assertEquals("Mike", new ArrayList<>(
              economyFlight.
                getPassengersSet()).get(0).getName()),
            () -> assertEquals(true,
                        economyFlight.removePassenger(mike)),
            () -> assertEquals(0,
                        economyFlight.getPassengersSet().size())
        );
    }
[...]

}
```

In this listing:

- John declares the instance variables for the test, including `economyFlight`, and `mike` as a `Passenger` (1).
- He writes the method corresponding to the `Given there is an economy flight` business logic step (2) by initializing `economyFlight` (3).
- He writes the method corresponding to the `When we have a regular passenger` business logic step (4) by initializing the regular passenger `mike` (5).
- He writes the method corresponding to the `Then you can add and remove him from an economy flight` business logic step (6) by checking all the conditions using the `assertAll` JUnit 5 method, which can now be read fluently (7).
- The rest of the methods are implemented in a similar way; we have covered the `Given`, `When`, and `Then` steps of one scenario.

To be able to run these tests, John needs a special new class that represents the test configuration. He names this class `PassengersPolicyStory`.

Listing 21.15 `PassengersPolicyStory` class

```
[...]
public class PassengersPolicyStory extends JUnitStory {

    @Override
    public Configuration configuration() {
        return new MostUsefulConfiguration()
                   .useStoryReporterBuilder(
                      new StoryReporterBuilder().
                             withDefaultFormats().
                          withFormats(Format.CONSOLE));
    }

    @Override
    public InjectableStepsFactory stepsFactory() {
        return new InstanceStepsFactory(configuration(),
                       new PassengersPolicy());
    }
}
```

In this listing:

- John declares the `PassengersPolicyStory` class that extends `JUnitStory` (1). A JBehave story class must extend `JUnitStory`.
- He overrides the `configuration` method (2) and specifies that the configuration of the report is the one that works for the most situations that users are likely to encounter (3) and that the report will be displayed on the console (4).
- He overrides the `stepsFactory` method (5) and specifies that the steps definition is to be found in the `PassengersPolicy` class (6).

The result of running these tests is shown in figure 21.18. The tests are a success, and the code coverage is 100%. However, JBehave's reporting capabilities do not allow the same nice display as in the case of Cucumber.

We can compare the length of the pre-BDD `AirportTest` class, which has 207 lines, with the JBehave `PassengersPolicy` class, which has 157 lines (just like the Cucumber version). The testing code is now only 75% of the pre-BDD size, but it has the same 100% coverage. Where does this gain come from? Remember that the `AirportTest` file contained seven classes on three 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 that only had JUnit 5.

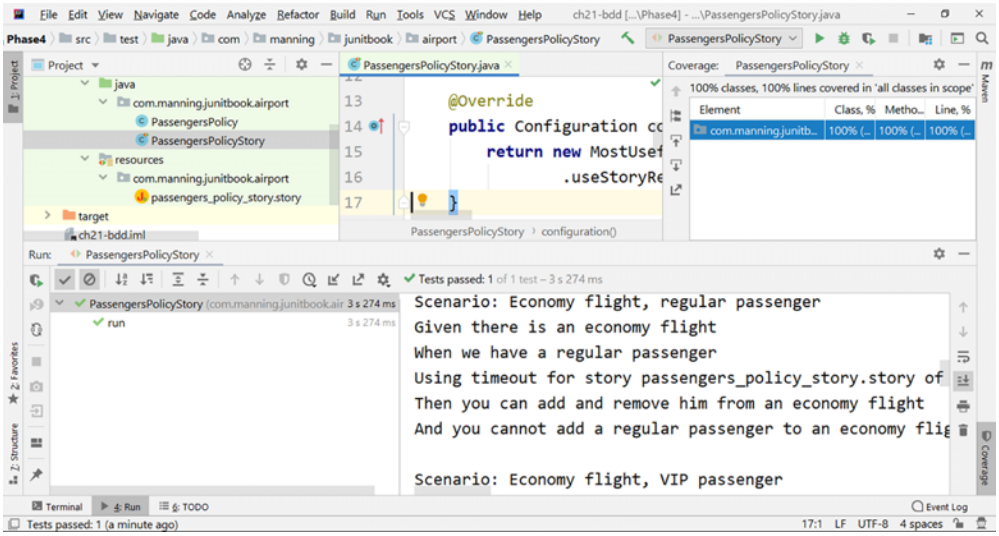


Figure 21.18 The JBehave passenger policy tests run successfully, and the code coverage is 100%.

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 awarded to passengers. John defines the scenarios for awarding bonus points in the bonus_policy_story.story file and generates JBehave tests that describe the scenarios. They are expected to fail at first.

Listing 21.16 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                    (1)
Given we have a regular passenger with a mileage
When the regular passenger travels <mileage1> and <mileage2> and <mileage3>   (2)
Then the bonus points of the regular passenger should be <points>

Examples:
| mileage1 | mileage2 | mileage3 | points |                  (3)
|      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 should be <points>

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 |
```

In this listing:

- John introduces new scenarios for the bonus policy, using the `Given`, `When`, and `Then` keywords (1).
- Values are replaced with parameters in the step definition: `<mileage1>`, `<mileage2>`, `<mileage3>`, and `<points>` (2).
- The effective values are defined in the `Examples` table at the end of each `Scenario` (3). The first row in the first table defines the values of three 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 and has the advantage that the values are kept in the scenarios and are easy for everyone to understand.

In the test folder, John creates the `BonusPolicy` class in the `com.manning.junitbook.airport` package (figure 21.19).

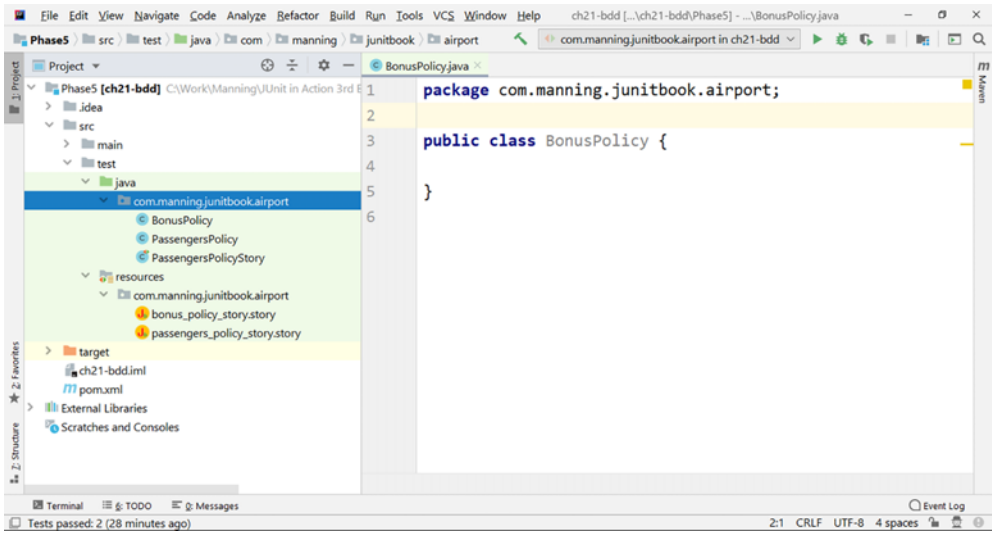


Figure 21.19 The newly introduced `BonusPolicy` class corresponds to the story file in the test/resources/com/manning/junitbook/airport folder.

To generate the steps in a Java file, John places the cursor on any not-yet-created step and presses Alt-Enter (figure 21.20). He generates all the steps in the newly created `BonusPolicy` class (figure 21.21). The `BonusPolicy` class is shown in listing 21.17 with the tests skeleton filled in.

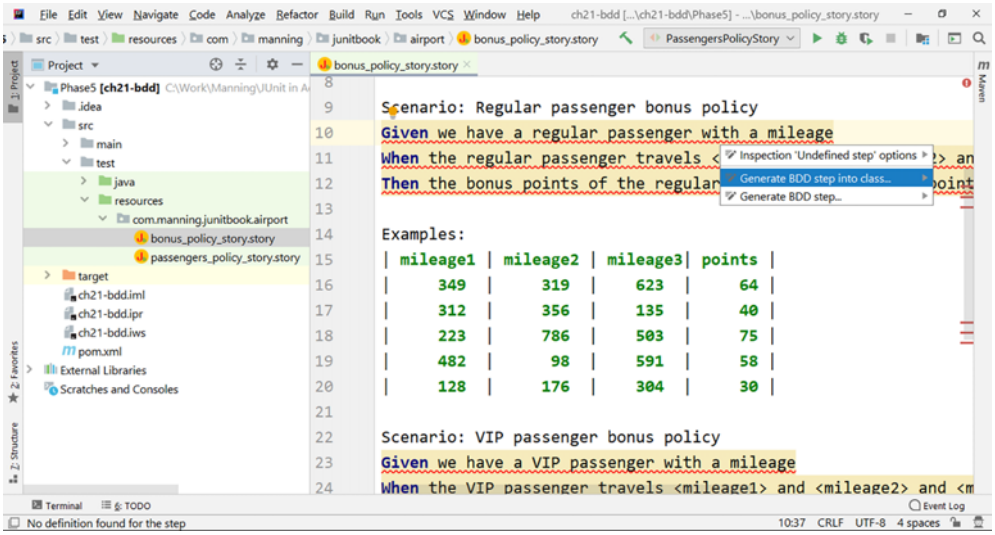


Figure 21.20 Pressing Alt-Enter to generate the BDD steps in a class

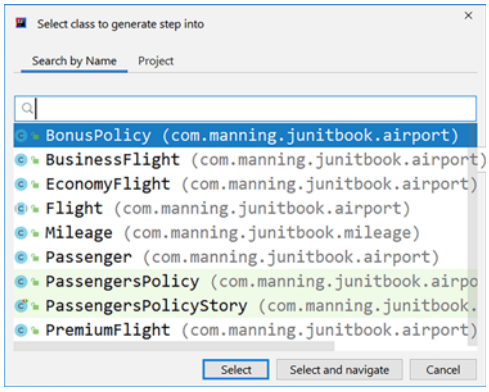


Figure 21.21 Choosing `BonusPolicy` as the class in which to generate the steps of the story

Listing 21.17 Skeleton of the JBehave `BonusPolicy` test

```
public class BonusPolicy {

    @Given("we have a regular passenger with a mileage")
    public void givenWeHaveARegularPassengerWithAMileage() {  ← 1

    }
```

```
    @When("the regular passenger travels <mileage1> and
           <mileage2> and <mileage3>")                          (2)
    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>")
    public void thenTheBonusPointsOfTheRegularPassengerShouldBePoints(
      @Named("points") int points) {                            (3)

   }
   [...]
}
```

In this listing:

- The JBehave plugin generates a method annotated with `@Given("we have a regular passenger with a mileage")`, meaning this method is executed when the step `Given we have a regular passenger with a mileage` from the scenario is executed (1).
- The plugin generates a method annotated with `@When("the regular passenger travels <mileage1> and <mileage2> and <mileage3>")`, meaning this method is executed when the step `When the regular passenger travels <mileage1> and <mileage2> and <mileage3>` from the scenario is executed (2).
- The plugin generates a method annotated with `@Then("the bonus points of the regular passenger should be <points>")`, meaning this method is executed when the step `Then the bonus points of the regular passenger should be <points>` from the scenario is executed (3).
- The rest of the methods generated by the JBehave plugin are similar; we have covered the `Given`, `When`, and `Then` steps of one scenario.

Next, John creates 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, and then implement the methods and make the tests pass.

Listing 21.18 `Mileage` class, with no implementation of the methods

```
public class Mileage {

    public static final int VIP_FACTOR = 10;
    public static final int REGULAR_FACTOR = 20;                (1)

    private Map<Passenger, Integer> passengersMileageMap =
                                    new HashMap<>();
    private Map<Passenger, Integer> passengersPointsMap =       (2)
                                    new HashMap<>();
```

```
    public void addMileage(Passenger passenger, int miles) {

    }

    public void calculateGivenPoints() {

    }

}
```

In this listing:

- John declares the `VIP_FACTOR` and `REGULAR_FACTOR` constants corresponding to the factor by which the mileage is divided for each type of passenger in order to get the bonus points (1).
- He declares `passengersMileageMap` and `passengersPointsMap`, two maps having the passenger as a key and keeping as a value the mileage and points for that passenger, respectively (2).
- He declares the `addMileage` method, which populates `passengersMileageMap` with the mileage for each passenger (3). The method does not do anything for now; it will be written later to fix the tests.
- He declares the `calculateGivenPoints` method, which populates `passengersPointsMap` with the bonus points for each passenger (4). The method does not do anything for now; it will be written later to fix the tests.

John now turns his attention to writing the unimplemented tests from the `BonusPolicy` class to follow the feature's business logic.

Listing 21.19 Business logic of the steps from `BonusPolicy`

```
[...]
public class BonusPolicy {
    private Passenger mike;
    private Mileage mileage;
    [...]

    @Given("we have a regular passenger with a mileage")
    public void givenWeHaveARegularPassengerWithAMileage() {
        mike  = new Passenger("Mike", false);
        mileage  = new Mileage();
    }

    @When("the regular passenger travels <mileage1> and <mileage2> and
                                        <mileage3>")
    public void the_regular_passenger_travels_and_and(@Named("mileage1")
                int mileage1, @Named("mileage2") int mileage2,
                @Named("mileage3") int mileage3) {
        mileage.addMileage(mike, mileage1);
        mileage.addMileage(mike, mileage2);
        mileage.addMileage(mike, mileage3);
    }
```

```
    @Then("the bonus points of the regular passenger should be <points>")
    public void the_bonus_points_of_the_regular_passenger_should_be
                (@Named("points") int points) {
        mileage.calculateGivenPoints();
        assertEquals(points,
           mileage.getPassengersPointsMap().get(mike).intValue());
    }
    [...]
}
```

In this listing:

- John declares the instance variables for the test, including `mileage`, and `mike` as a `Passenger` (1).
- He writes the method corresponding to the `Given we have a regular passenger with a mileage` business logic step (2) by initializing the passenger and the mileage (3).
- He writes the method corresponding to the `When the regular passenger travels <mileage1> and <mileage2> and <mileage3>` business logic step (4) by adding mileages to the regular passenger `mike` (5).
- He writes the method corresponding to the `Then the bonus points of the regular passenger should be <points>` business logic step (6) by calculating the points (7) and checking that the calculated value is as expected (8).
- The rest of the methods are implemented in a similar way; we have covered the `Given`, `When`, and `Then` steps of one scenario.

To run these tests, John needs a new special class that represents the test configuration. He names this class `BonusPolicyStory`.

Listing 21.20 `BonusPolicyStory` class

```
[...]
public class BonusPolicyStory extends JUnitStory {

    @Override
    public Configuration configuration() {
        return new MostUsefulConfiguration()
                   .useStoryReporterBuilder(
                      new StoryReporterBuilder().
                             withDefaultFormats().
                          withFormats(Format.CONSOLE));
    }

    @Override
    public InjectableStepsFactory stepsFactory() {
        return new InstanceStepsFactory(configuration(),
                       new BonusPolicy());
    }
}
```

In this listing:

- John declares the `BonusPolicyStory` class that extends `JUnitStory` (1).
- He overrides the `configuration` method (2) and specifies that the configuration of the report is the one that works for the most situations that users are likely to encounter (3), and that the report will be displayed on the console (4).
- He overrides the `stepsFactory` method (5) and specifies that the step definition is found in the `BonusPolicy` class (6).

If we run the bonus point tests now, they fail (figure 21.22), because the business logic is not yet implemented (the `addMileage` and `calculateGivenPoints` methods are empty).

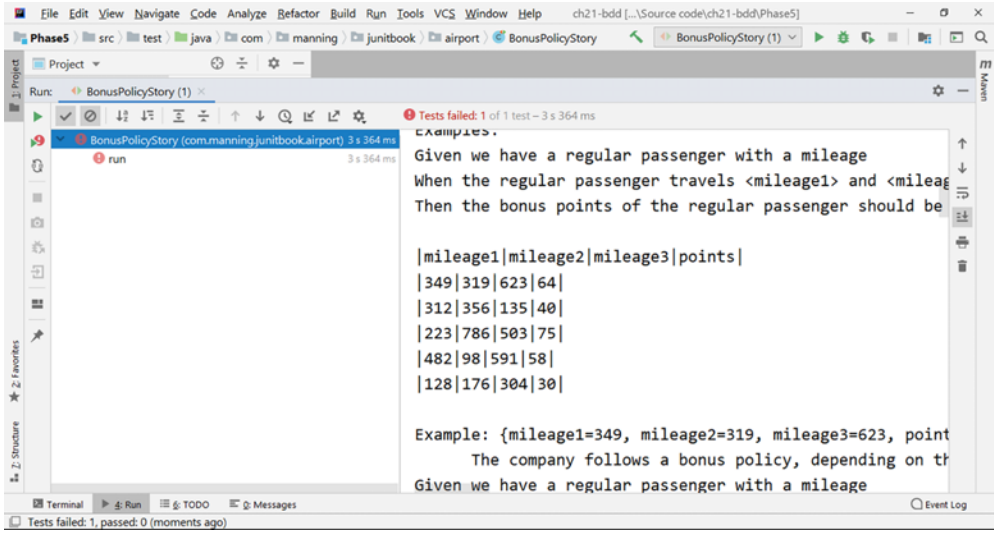


Figure 21.22 The JBehave bonus point tests fail before the business logic is implemented.

John goes back to the implementation of the two remaining business logic methods from the `Mileage` class (`addMileage` and `calculateGivenPoints`).

Listing 21.21 Implementing 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);
    }

}
```

(1)

(2)

```
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 this listing:

- In the `addMileage` method, John checks whether `passengersMileageMap` already contains a passenger (1). If that passenger already exists, he adds the mileage to the passenger (2); otherwise, he creates a new entry in the map with that passenger as a key and the miles as the initial value (3).
- In the `calculateGivenPoints` method, he browses the passenger set (3) and, for each passenger, if the passenger is a VIP, calculates the bonus points by dividing the mileage by the VIP factor (4). Otherwise, he calculates the bonus points by dividing the mileage by the regular factor (5).

Running the bonus point tests now is successful, as shown in figure 21.23. John has successfully implemented the bonus policy feature while working BDD style with JUnit 5 and JBehave.

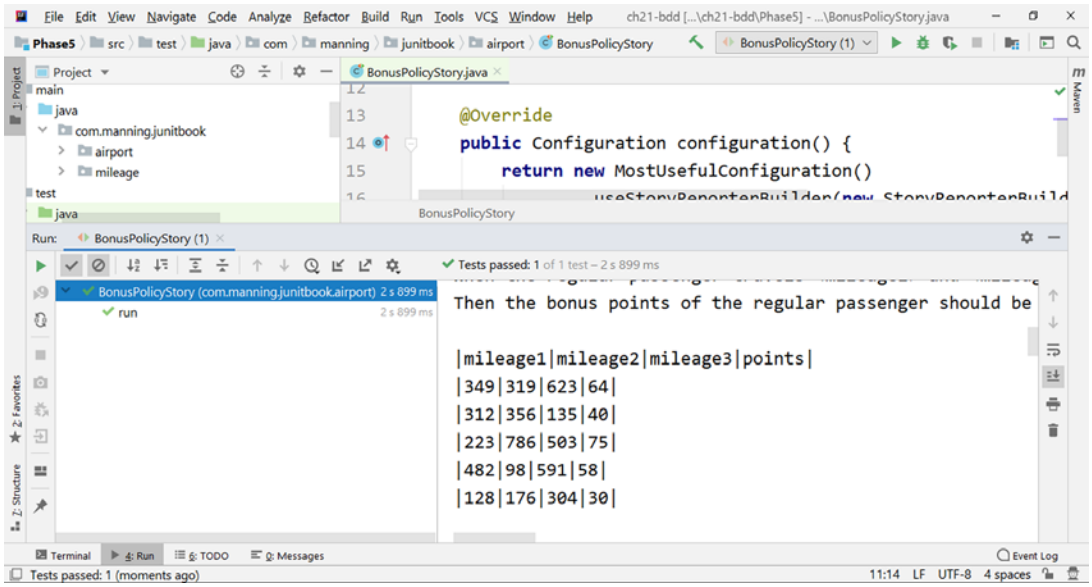


Figure 21.23 The JBehave bonus point tests succeed after the business logic has been implemented.

21.4 Comparing Cucumber and JBehave

Cucumber and JBehave have similar approaches and supporting BDD concepts. They are different frameworks but are 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. They use some of the same BDD keywords (`Given`, `When`, `Then`) but also have some variations in terms like `Scenario Outline` for Cucumber and `Scenario with Examples` for JBehave.

IntelliJ IDE support for both Cucumber and JBehave is provided through plugins that help from generating steps in Java code to checking code coverage. The Cucumber plugin produces nicer output, allowing us to follow the full testing hierarchy at a glance and displaying everything with significant colors. It also lets us run a test directly from the feature text file, which is easier to follow, especially for nontechnical people.

JBehave reached its maturity phase some time ago, while the Cucumber code base is still updated very frequently. For example, at the time of writing, the Cucumber GitHub displays tens of commits from the previous seven days, whereas the JBehave GitHub displays a single commit in the last seven days. Cucumber also has a more active community at the time of writing; articles on blogs and forums are more recent and frequent, which makes troubleshooting easier for developers. In addition, Cucumber is available for other programming languages. In terms of code size compared to the pre-BDD situation, both Cucumber and JBehave show similar performance, reducing the size of the initial code by the same proportion.

Choosing one of these frameworks is a matter of habit or preference (personal preference or project preference). We have shown them side by side in a very practical and comparable way so that you can eventually make your own choice.

The next chapter is dedicated to building a test pyramid strategy, from the low level to the high level, and applying it while working with JUnit 5.

Summary

This chapter has covered the following:

- Introducing BDD, a software development technique that encourages teams to deliver software that matters and supports cooperation between stakeholders
- Analyzing the benefits of BDD: addressing user needs, clarity, change support, automation support, focus on adding business value, and cost reduction
- Analyzing the challenges of BDD: it requires engagement and strong collaboration, interaction, direct communication, and constant feedback
- Moving a TDD application to BDD
- Developing business logic 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 business logic 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 the BDD principles, ease of use, and code size needed to implement functionality

22 Implementing a test pyramid strategy with JUnit 5

This chapter covers

- Building unit tests for components in isolation
- Building integration tests for units as a group
- Building system tests for complete software
- Building acceptance tests and making sure software is compliant with business requirements

> *The test pyramid is a way of thinking about how 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 several purposes. Tests make us interact with the application and understand how it works. Testing helps us deliver 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 very important.

22.1 Software testing levels

The different levels of software tests can be regarded as a pyramid, as shown in figure 22.1. In this chapter, we'll discuss the following levels of software testing (from lowest to highest):

- *Unit testing*—Unit testing is at the foundation of the pyramid. It focuses on methods or classes (individual units) by testing each one in isolation to determine whether it works according to expectations.
- *Integration testing*—Individual, verified software components are combined in larger aggregates and tested together.
- *System testing*—Testing is performed on a complete system to evaluate its compliance with the specification. System testing requires no knowledge of the design or code but focuses on the functionality of the entire system.
- *Acceptance testing*—Acceptance testing uses scenarios and test cases to check whether the application satisfies the expectations of the end user.

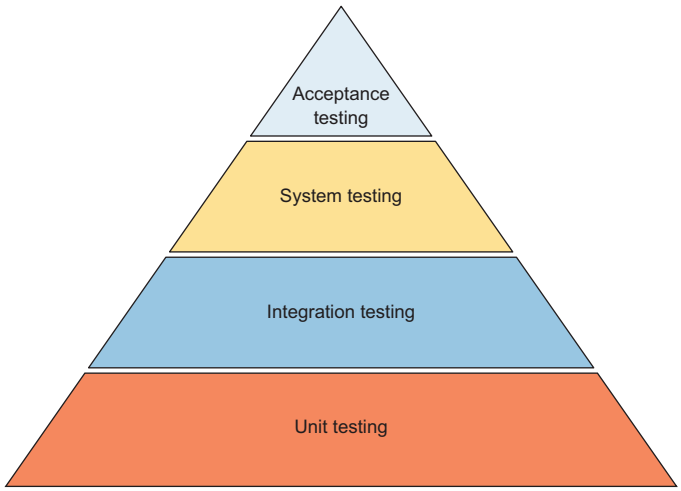


Figure 22.1 The testing pyramid has many simple units at the bottom level, and fewer, more complicated units at the top.

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. Low-level testing addresses individual components; it is concerned more with details and less with the broader view. High-level testing is more abstract; it verifies the overall goals and features of the system and is more focused on the user's interaction with the GUI and how the system works as a whole.

When it comes to what to test, we can identify the following:

- *Business logic*—How the program translates the business rules from the real world.
- *Bad input values*—For example, in our example flight-management application, we cannot assign a negative number of seats on a flight.

- *Boundary conditions*—Extremes of an input domain, such as the maximum or minimum. We might test flights having zero passengers or the maximum allowed number of passengers.
- *Unexpected conditions*—Conditions that are not part of the normal operation of a program. For instance, a flight cannot change its origin once it has taken off.
- *Invariants*—Expressions whose values do not change during program execution. For example, a person's identifier cannot change during the execution of the program.
- *Regressions*—Bugs introduced in an existing system after upgrades or patches.

We'll start now by analyzing the implementation of each testing level, starting from the bottom.

22.2 Unit testing: Basic components working in isolation

In this chapter, we'll demonstrate how Thomas, a developer with Tested Data Systems, builds a test pyramid strategy for the flight-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 `Passenger` class

```
public class Passenger {

   private String identifier;
   private String name;
   private String countryCode;
   private String ssnRegex =
         "^(?!000|666)[0-8][0-9]{2}-(?!00)[0-9]{2}-(?!0000)[0-9]{4}$";
   private String nonUsIdentifierRegex =
         "^(?!000|666)[9][0-9]{2}-(?!00)[0-9]{2}-(?!0000)[0-9]{4}$";
   private Pattern pattern;


   public Passenger(String identifier, String name, String countryCode){
      pattern = countryCode.equals("US")? Pattern.compile(ssnRegex):
                Pattern.compile(nonUsIdentifierRegex);
      Matcher matcher = pattern.matcher(identifier);
      if(!matcher.matches()) {
         throw new RuntimeException("Invalid identifier");
      }

      if(!Arrays.asList(Locale.getISOCountries()).contains(countryCode)){
         throw new RuntimeException("Invalid country code");
      }

      this.identifier = identifier;
      this.name = name;
      this.countryCode = countryCode;
   }
```

1 2 3 4 5 6 7 8 9

```
    public String getIdentifier() {
        return identifier;
    }

    public void setIdentifier(String identifier) {
        Matcher matcher = pattern.matcher(identifier);
        if(!matcher.matches()) {
            throw new RuntimeException("Invalid identifier");
        }

        this.identifier = identifier;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getCountryCode() {
        return countryCode;
    }

    public void setCountryCode(String countryCode) {
        if(!Arrays.asList(Locale.getISOCountries()).
                                contains(countryCode)){
            throw new RuntimeException("Invalid country code");
        }

        this.countryCode = countryCode;
    }

    @Override
    public String toString() {
        return "Passenger " + getName() + " with identifier: "
                + getIdentifier() + " from " + getCountryCode();
    }
}
```

In this listing:

- Thomas declares the `identifier`, `name`, and `countryCode` instance variables for `Passenger` (1).
- If the passenger is a US citizen, the identifier is their Social Security number (SSN). `ssnRegex` describes the regular expression to be followed by the SSN. The SSN must conform to the following rules: the first three digits cannot be 000, 666, or between 900 and 999 (2).
- If the passenger is not a US citizen, the identifier is generated by the company, following rules similar to those for an SSN. `nonUsIdentifierRegex` only allows identifiers with the first digits between 900 and 999 (3). Thomas also declares a pattern that checks whether an `identifier` is following the rules (4).

- In the constructor ❺, Thomas creates the `pattern` ❻, checks whether the `identifier` matches it ❼, checks whether the country code is valid ❽, and then constructs the passenger ❾.
- He provides getters and setters for the fields ❿, checking the validity of the input data for an identifier that has to match the pattern ⓫ and that the country code exists ⓬.
- He also overrides the `toString` method to include the name, identifier, and country code of the passenger ⓭.

Listing 22.2 `Flight` class

```
public class Flight {

    private String flightNumber;
    private int seats;
    private int passengers;
    private String origin;
    private String destination;
    private boolean flying;
    private boolean takenOff;
    private boolean landed;                                          ❶

    private String flightNumberRegex = "^[A-Z]{2}\\d{3,4}$";         ❷
    private Pattern pattern = Pattern.compile(flightNumberRegex);    ❸

    public Flight(String flightNumber, int seats) {
        Matcher matcher = pattern.matcher(flightNumber);             ❹
        if(!matcher.matches()) {
            throw new RuntimeException("Invalid flight number");
        }
        this.flightNumber = flightNumber;
        this.seats = seats;
        this.passengers = 0;
        this.flying = false;                                         ❺
        this.takenOff = false;
        this.landed = false;
    }

    public String getFlightNumber() {
      return flightNumber;
    }

    public int getSeats() {                                          ❻
        return seats;
    }

    public void setSeats(int seats) {
        if(passengers > seats) {
            throw new RuntimeException("Cannot reduce the number of
                    seats under the number of existing passengers!");
        }                                                            ❼

        this.seats = seats;
    }
```

```
    public int getPassengers() {
        return passengers;
    }

    public String getOrigin() {
        return origin;
    }

    public void setOrigin(String origin) {
        if(takenOff){
            throw new RuntimeException("Flight cannot change its origin
                                        any longer!");
        }

        this.origin = origin;
    }

    public String getDestination() {
        return destination;
    }

    public void setDestination(String destination) {
        if(landed){
            throw new RuntimeException("Flight cannot change its
                                        destination any longer!");
        }

        this.destination = destination;
    }

    public boolean isFlying() {
        return flying;
    }

    public boolean isTakenOff() {
        return takenOff;
    }

    public boolean isLanded() {
        return landed;
    }

    @Override
    public String toString() {
        return "Flight " + getFlightNumber() + " from " + getOrigin()
                         +  " to " + getDestination();
    }

    public void addPassenger() {
        if(passengers >= seats) {
            throw new RuntimeException("Not enough seats!");
        }
        passengers++;
    }
```

6

8

6

9

6

10

11

```
    public void takeOff() {
       System.out.println(this + " is taking off");
       flying = true;
       takenOff = true;
    }

    public void land() {
        System.out.println(this + " is landing");
        flying = false;
        landed = true;
    }

}
```

In this listing:

- Thomas declares the instance variables for a `Flight` (1).
- The flight number needs to match a regular expression: a code for an airline service consists of a two-character airline designator and a three- or four-digit number (2). Thomas also declares a `pattern` that checks whether a flight number is following the rules (3).
- In the constructor, he checks whether the flight number matches the pattern (4) and then constructs the flight (5).
- He provides getters and setters for the fields (6), checking that there are not more passengers than seats (7), that the origin cannot change after the plane has taken off (8), and that the destination cannot change after the plane has landed (9).
- He overrides the `toString` method to display the flight number, origin, and destination (10).
- When he adds a passenger to the plane, he checks that there are enough seats (11).
- When the plane takes off, he prints a message and changes the state of the plane (12). He also prints a message and changes the state of the plane when the plane lands (13).

The functionality of the `Passenger` class is verified in the `PassengerTest` class.

Listing 22.3 `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 testCreatePassengerWithInvalidSsn() {
    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 testCreatePassengerWithInvalidNonUsIdentifier() {
    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 testCreatePassengerWithInvalidCountryCode() {
    assertThrows(RuntimeException.class,
            ()->{
                Passenger passenger = new Passenger("900-45-6789",
                                            "John Smith", "GJ");
            });
}

@Test
public void testSetInvalidSsn() {
    assertThrows(RuntimeException.class,
            ()->{
                Passenger passenger = new Passenger("123-45-6789",
                                            "John Smith", "US");
                passenger.setIdentifier("123-456-789");
            });

}
```

2

3

4

5

6

```
    @Test
    public void testSetValidSsn() {
              Passenger passenger = new Passenger("123-45-6789",
                                        "John Smith", "US");
              passenger.setIdentifier("123-98-7654");
              assertEquals("123-98-7654", passenger.getIdentifier());
    }

    @Test
    public void testSetValidNonUsIdentifier() {
        Passenger passenger = new Passenger("900-45-6789",
                                    "John Smith", "GB");
        passenger.setIdentifier("900-98-7654");
        assertEquals("900-98-7654", passenger.getIdentifier());
    }

    @Test
    public void testSetInvalidCountryCode() {
        assertThrows(RuntimeException.class,
                ()->{
                    Passenger passenger = new Passenger("123-45-6789",
                                                "John Smith", "US");
                    passenger.setCountryCode("GJ");
                });

    }

    @Test
    public void testSetValidCountryCode() {
              Passenger passenger = new Passenger("123-45-6789",
                                          "John Smith", "US");
              passenger.setCountryCode("GB");
              assertEquals("GB", passenger.getCountryCode());
    }

    @Test
    public void testPassengerToString() {
        Passenger passenger = new Passenger("123-45-6789",
                                    "John Smith", "US");
        passenger.setName("John Brown");
        assertEquals("Passenger John Brown with identifier:
                      123-45-6789 from US",  passenger.toString());
    }

}
```

In this listing:

- Thomas checks the correct creation of a US passenger (1) and a non-US passenger (2), with correct identifiers.
- He checks that he cannot set an invalid identifier for a US citizen (3) or a non-US citizen (4).
- He checks that he cannot set an invalid country code (5) or an invalid SSN (6).

- He checks that he can set a valid SSN for a US citizen (7) and a valid identifier for a non-US citizen (8).
- He checks setting an invalid country code (9) and a valid country code (10).
- He checks the behavior of the `toString` method (11).

The functionality of the `Flight` class is verified in the `FlightTest` class.

Listing 22.4 `FlightTest` class

```
public class FlightTest {

    @Test
    public void testFlightCreation() {
        Flight flight = new Flight("AA123", 100);
        assertNotNull(flight);
    }

    @Test
    public void testInvalidFlightNumber() {
        assertThrows(RuntimeException.class,
                ()->{
                    Flight flight = new Flight("AA12", 100);
                });
        assertThrows(RuntimeException.class,
                ()->{
                    Flight flight = new Flight("AA12345", 100);
                });
    }

    @Test
    public void testValidFlightNumber() {
        Flight flight = new Flight("AA345", 100);
        assertNotNull(flight);
        flight = new Flight("AA3456", 100);
        assertNotNull(flight);
    }

    @Test
    public void testAddPassengers() {
        Flight flight = new Flight("AA1234", 50);
        flight.setOrigin("London");
        flight.setDestination("Bucharest");
        for(int i=0; i<flight.getSeats(); i++) {
            flight.addPassenger();
        }
        assertEquals(50, flight.getPassengers());
        assertThrows(RuntimeException.class,
                ()->{
                    flight.addPassenger();
                });

    }
```

```
@Test
public void testSetInvalidSeats() {
    Flight flight = new Flight("AA1234", 50);
    flight.setOrigin("London");
    flight.setDestination("Bucharest");
    for(int i=0; i<flight.getSeats(); i++) {
        flight.addPassenger();
    }
    assertEquals(50, flight.getPassengers());
    assertThrows(RuntimeException.class,
            ()->{
                flight.setSeats(49);
            });
}

@Test
public void testSetValidSeats() {
    Flight flight = new Flight("AA1234", 50);
    flight.setOrigin("London");
    flight.setDestination("Bucharest");
    for(int i=0; i<flight.getSeats(); i++) {
        flight.addPassenger();
    }
       assertEquals(50, flight.getPassengers());
    flight.setSeats(52);
       assertEquals(52, flight.getSeats());
}

@Test
public void testChangeOrigin() {
    Flight flight = new Flight("AA1234", 50);
    flight.setOrigin("London");
    flight.setDestination("Bucharest");
    flight.takeOff();
    assertEquals(true, flight.isFlying());
    assertEquals(true, flight.isTakenOff());
    assertEquals(false, flight.isLanded());
    assertThrows(RuntimeException.class,
            ()->{
                flight.setOrigin("Manchester");
            });
}

@Test
public void testChangeDestination() {
 Flight flight = new Flight("AA1234", 50);
    flight.setOrigin("London");
    flight.setDestination("Bucharest");
    flight.takeOff();
    flight.land();
    assertThrows(RuntimeException.class,
            ()->{
                flight.setDestination("Sibiu");
            });
}
```

5

6

7

8

```
    @Test
    public void testLand() {
        Flight flight = new Flight("AA1234", 50);
        flight.setOrigin("London");
        flight.setDestination("Bucharest");
        flight.takeOff();
        assertEquals(true, flight.isTakenOff());
        assertEquals(false, flight.isLanded());
        flight.land();
        assertEquals(true, flight.isTakenOff());
        assertEquals(true, flight.isLanded());
        assertEquals(false, flight.isFlying());
    }
}
```

In this listing:

- Thomas checks the creation of a flight (1).
- He checks that he cannot set an invalid flight number (2) but can set a valid one (3).
- He checks that he can add passengers only within the seat limit (4).
- He checks that he cannot set the number of seats less than the number of passengers (5), but he can set it greater than the number of passengers (6).
- He checks that he cannot change the origin after the plane has taken off (7) or the destination after the plane has landed (8).
- He checks that the plane has changed its state after taking off (9) and again after landing (10).

The result of running the unit tests for the `Passenger` and `Flight` classes is successful, and the code coverage is 100%, as shown in figure 22.2.

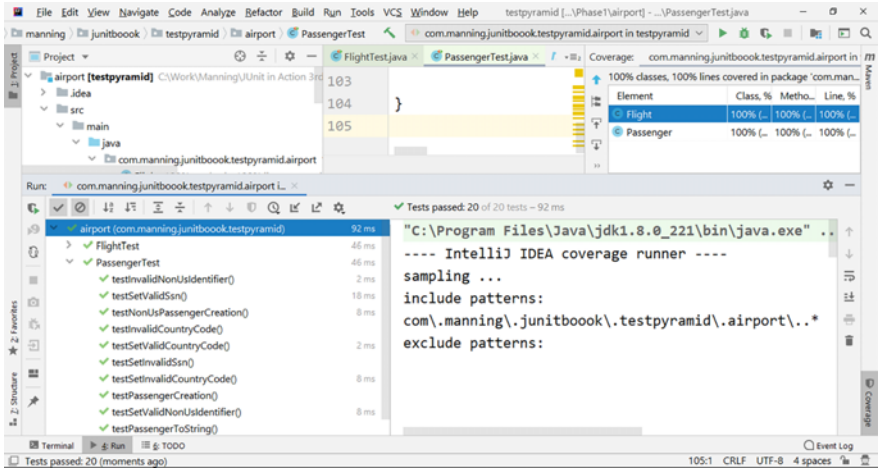


Figure 22.2 The unit tests for `Passenger` and `Flight` check the correct behavior of the individual classes and run successfully.

Thomas has successfully created the application consisting of the `Passenger` and `Flight` classes and has checked that each class works fine individually. Using JUnit 5 tests, he has checked the following:

- Passenger identifier and country code restrictions
- Flight number restrictions (verifying that they start with two letters, followed by three of four digits)
- Bad input values (such as a flight with a negative number of seats)
- Boundary conditions (he cannot add more passengers than there are available seats)

From here, Thomas will move on to integrate the functionality of the two classes.

22.3 Integration testing: Units combined into a group

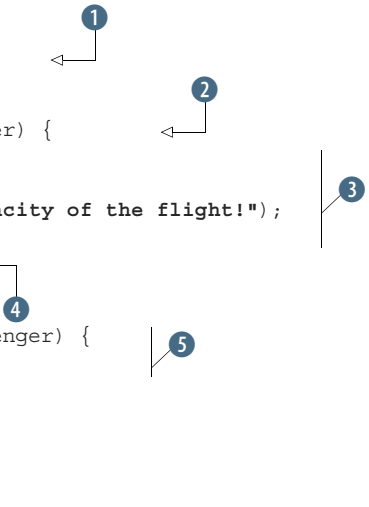
Integration testing combines individual units to check their interactions. The fact that units work fine in isolation does not necessarily mean they work fine together.

Thomas will now investigate how the `Passenger` and `Flight` classes cooperate. They represent two different units; to cooperate, they need to expose appropriate interfaces (APIs). However, interfaces can have defects that prevent interactions, such as missing methods or methods that do not receive the appropriate types of arguments.

When Thomas analyzes the current interfaces, he finds that a passenger should be added to a flight and removed from a flight. Currently, he is only modifying the number of passengers: the current interface is missing an `addPassenger` method as well as a `removePassenger` method. The `Flight` class should be more integrated with `Passenger` and maintain the set of recorded passengers. The changes to the `Flight` class are shown in the following listing.

Listing 22.5 Modified `Flight` class

```
public class Flight {

   Set<Passenger> passengers = new HashSet< >();

   [...]
   public boolean addPassenger(Passenger passenger) {
       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) {
       return passengers.remove(passenger);
   }

   public int getPassengersNumber() {
       return passengers.size();
   }

[...]
}
```



In this listing:

- Thomas adds a `passengers` field to hold the set of passengers (1). This replaces the integer `passengers` field, and consequently the `this.passengers = 0` initialization is no longer needed.
- He adds an `addPassenger` method (2) that checks whether there are enough seats (3) and then adds a passenger to the set of passengers (4).
- He adds a `removePassenger` method that removes a passenger from the set of passengers (5) and the `getPassengersNumber` method that returns the size of the passengers set (6).

To perform integration testing, Thomas decides to use Arquillian—a testing framework for Java that uses 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 integration testing.

Arquillian does not have a JUnit 5 extension at this time, but it is very popular and has been frequently adopted in projects up to JUnit 4. It greatly simplifies the task of managing containers, deployments, and framework initializations.

Arquillian tests Java EE applications. Using it in our examples requires some basic knowledge of Contexts and Dependency Injection (CDI), a Java EE standard for the inversion of control design pattern. We'll explain the most important ideas as we go so that you can quickly adopt Arquillian within your projects.

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 can 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 the following CSV file.

Listing 22.6 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 `FlightBuilderUtil` 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 `FlightBuilderUtil` class

```
public class FlightBuilderUtil {

    public static Flight buildFlightFromCsv() throws IOException {
        Flight flight = new Flight("AA1234", 50);
           flight.setOrigin("London");                                    (1)
           flight.setDestination("Bucharest");

        try(BufferedReader reader =
               new BufferedReader(new FileReader(
                    "src/test/resources/flights_information.csv")))       (2)
        {
```

```
            String line = null;
            do {
                line = reader.readLine();
                if (line != null) {
                 String[] passengerString = line.toString().split(";");
                  Passenger passenger =
                        new Passenger(passengerString[0].trim(),
                                      passengerString[1].trim(),
                                      passengerString[1].trim());
                  flight.addPassenger(passenger);
                }
            } while (line != null);

        }

        return flight;
    }
}
```

In this listing:

- Thomas creates a flight and sets its origin and destination (1).
- He opens the CSV file to parse (2).
- He reads the file line by line (3), splits each line (4), creates a passenger based on the information that has been read (5), and adds the passenger to the flight (6).
- He returns the fully populated flight from the method (7).

So far, all the classes that have been implemented during development of the flight- and passenger-management tasks are pure Java classes; no particular framework or technology has been used. As we mentioned, the Arquillian testing framework executes test cases against a Java container, so using it requires some understanding of notions related to Java EE and CDI. Arquillian abstracts the container or application startup logic and deploys the application to the targeted runtime (an application server, embedded or managed) to execute test cases.

For this example, the following listing shows the dependencies that Thomas needs 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>
                <artifactId>arquillian-bom</artifactId>
                <version>1.4.0.Final</version>
                <scope>import</scope>
                <type>pom</type>
            </dependency>
        </dependencies>
</dependencyManagement>
```

```
<dependencies>
        <dependency>
            <groupId>org.jboss.spec</groupId>
            <artifactId>jboss-javaee-7.0</artifactId>
            <version>1.0.3.Final</version>
            <type>pom</type>
            <scope>provided</scope>
        </dependency>
        <dependency>
            <groupId>org.junit.vintage</groupId>
            <artifactId>junit-vintage-engine</artifactId>
            <version>5.4.2</version>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>org.jboss.arquillian.junit</groupId>
            <artifactId>arquillian-junit-container</artifactId>
           <scope>test</scope>
        </dependency>
        <dependency>
           <groupId>org.jboss.arquillian.container</groupId>
           <artifactId>arquillian-weld-ee-embedded-1.1</artifactId>
           <version>1.0.0.CR9</version>
           <scope>test</scope>
        </dependency>
        <dependency>
           <groupId>org.jboss.weld</groupId>
           <artifactId>weld-core</artifactId>
           <version>2.3.5.Final</version>
           <scope>test</scope>
        </dependency>
</dependencies>
```

This listing adds the following:

- The Arquillian API dependency (1).
- The Java EE 7 API dependency (2).
- The JUnit Vintage Engine dependency (3). As mentioned earlier, at least for the moment, Arquillian is not yet integrated with JUnit 5. Because Arquillian lacks a JUnit 5 extension, Thomas has to use JUnit 4 dependencies and annotations to run the tests.
- The Arquillian JUnit integration dependency (4).
- The container adapter dependencies (5) (6). To execute tests against a container, Thomas 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 next, 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 `FlightWithPassengersTest` class

```
[...]
@RunWith(Arquillian.class)                                          (1)
public class FlightWithPassengersTest {

    @Deployment
    public static JavaArchive createDeployment() {                  (2)
        return ShrinkWrap.create(JavaArchive.class)
                .addClasses(Passenger.class, Flight.class)
                .addAsManifestResource(EmptyAsset.INSTANCE, "beans.xml");
    }

    @Inject
    Flight flight;                                                  (3)

    @Test(expected = RuntimeException.class)
    public void testNumberOfSeatsCannotBeExceeded() throws IOException {   (4)
        assertEquals(50, flight.getPassengersNumber());
        flight.addPassenger(new Passenger("124-56-7890",
                                "Michael Johnson", "US"));
    }

    @Test
    public void testAddRemovePassengers() throws IOException {      (5)
        flight.setSeats(51);
        Passenger additionalPassenger =
             new Passenger("124-56-7890", "Michael Johnson", "US");
        flight.addPassenger(additionalPassenger);
        assertEquals(51, flight.getPassengersNumber());
        flight.removePassenger(additionalPassenger);
        assertEquals(50, flight.getPassengersNumber());
        assertEquals(51, flight.getSeats());
    }
}
```

As this listing shows, an Arquillian test case must have three things:

- A `@RunWith(Arquillian.class)` annotation on the class (1). The `@RunWith` annotation tells JUnit to use Arquillian as the test controller.
- A public static method annotated with `@Deployment` that returns a ShrinkWrap archive (2). The purpose of the test archive is to isolate the classes and resources that the test needs. The archive is defined with ShrinkWrap. The microdeployment 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, Thomas includes only the `Passenger` and `Flight` classes. He tries to inject a `Flight` object as a class member using the CDI `@Inject` annotation (3). The `@Inject` annotation allows us to define injection points in 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` (4) (5). Arquillian looks for a public static method annotated with the `@Deployment` annotation to retrieve the test archive. Then each `@Test`-annotated method is run in 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.

Now that the infrastructure is in place for using Arquillian in the project, we can run the integration tests with its help! If we run the tests now, we get an error (figure 22.3).

```
ıtException: WELD-001408: Unsatisfied dependencies for type Flight with qualifiers @Default
:edField] @Inject com.manning.junitbook.testpyramid.airport.FlightWithPassengersTest.flight
ımid.airport.FlightWithPassengersTest.flight(FlightWithPassengersTest.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 the container is trying to inject the dependency, as it has been instructed to do 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 producer methods that are designed to inject objects that require custom initialization. The solution in the next listing fixes the issue and is easy to put into practice, even for a junior developer.

Listing 22.10 `FlightProducer` class

```
[...]
public class FlightProducer {

    @Produces
    public Flight createFlight() throws IOException {
        return FlightBuilderUtil.buildFlightFromCsv();
    }
}
```

In this listing, Thomas creates the `FlightProducer` class with the `createFlight` method, which invokes `FlightBuilderUtil.buildFlightFromCsv()`. He can use this method to inject objects that require custom initialization: in this case, he injects a flight that has been configured based on the CSV file. Thomas annotates the `createFlight` method with `@Produces`, which is also a Java EE annotation. The container automatically invokes this method to create the configured flight; then it injects the method into the `Flight` field, annotated with `@Inject` from the `FlightWithPassengersTest` class.

Now Thomas adds the `FlightProducer` class to the ShrinkWrap archive.

Listing 22.11 Modified deployment method from `FlightWithPassengersTest`

```
@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 are green, and the code coverage is 100%. The container injected the correctly configured flight (figure 22.4). Thomas has successfully created the integration layer of the test pyramid and is ready to move to the next level.

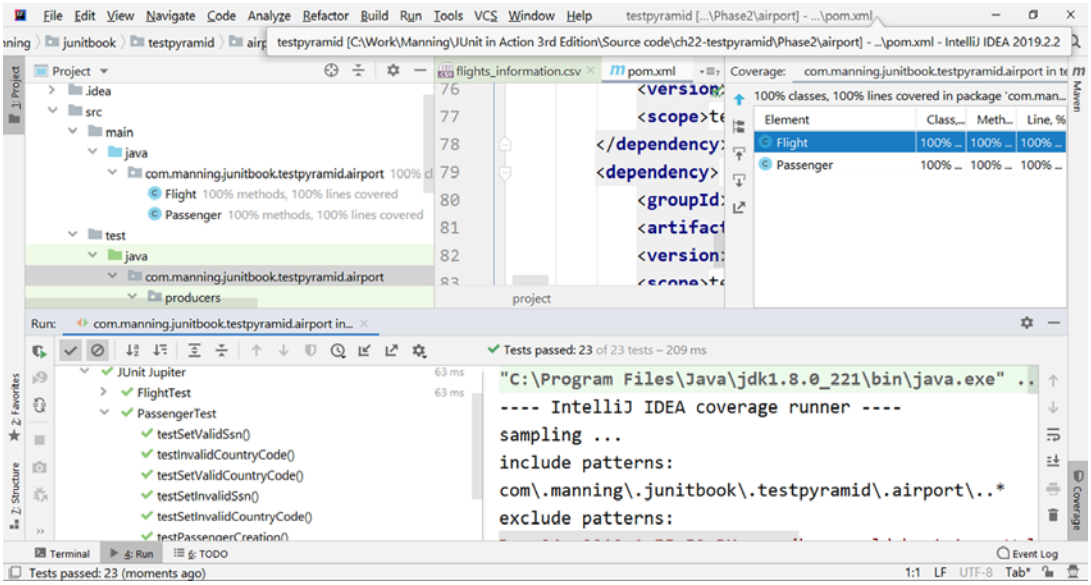


Figure 22.4 Successfully executing the integration tests from `FlightWithPassengersTest`. The code coverage is 100% after Arquillian is fully configured.

22.4 System testing: Looking at the complete software

System testing tests the entire system to evaluate its 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 it is impractical to incorporate a real object (for example, some depended-on component) into a test, or impossible—at least for the moment—as the depended-on component is not yet available (figure 22.5). For example, our system may depend on a device providing measurements of outside conditions (temperature, humidity). The results offered by this device influence our tests and are nondeterministic—we cannot simply decide the meteorological conditions that we need at a given time.

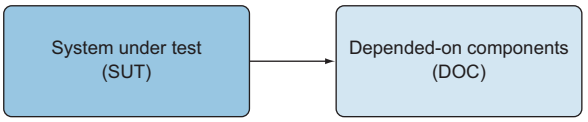


Figure 22.5 The system under test (SUT) needs a depended-on component (DOC) that is not available at the beginning of system development.

When developing a program, we may need to create mock objects that simulate the behavior of complex, real objects 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 saves our tests from having to wait for the availability of that service. We want to make these mock objects an accurate representation of the external service. It is important that the DOC keeps its contract—the expected behavior and an API that our system may use.

When the DOC is created in parallel by a different team, it is useful to use a consumer-driven contract approach. This means the provider must follow an API or a behavior expected by the consumer.

22.4.1 Testing with a mock external dependency

The flight-management application is currently at the level of integration testing. Thomas now needs to add a new feature that will award bonus points to passengers, depending on the distances they travel with the company. The bonus policy is simple: a passenger gets 1 bonus point for every 10 kilometers they travel.

The management of the bonus policy is externalized to another team that will provide the `DistancesManager` class. So far, Thomas knows that the interface forms the contract between the consumer and the provider. The application is following a consumer-driven contract.

He knows that the `DistancesManager` API provides the following methods:

- `getPassengersDistancesMap` provides a map that has the passenger as a key and the traveled distance as a value.
- `getPassengersPointsMap` provides a map that has the passenger as a key and the bonus points as a value.

- `addDistance` adds a traveled distance to the passenger.
- `calculateGivenPoints` calculates the bonus points for a passenger.

Thomas does not know the implementations yet, so he provides dummy implementations and mocks the behavior of the class.

Listing 22.12 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() {
    }

}
```

Thomas has one flight description in a CSV file, but this is not enough. To make sure the calculations are consistent, he needs passengers who have participated in more than one flight. So, he introduces two more CSV files describing two other flights that have passengers in common with the first flight (the following listing shows partial passenger lists).

Listing 22.13 flights_information2.csv and flights_information3.csv files

```
123-45-6789; John Smith; US
900-45-6789; Jane Underwood; GB
123-45-6790; James Perkins; US
[...]

123-45-6790; James Perkins; US
900-45-6790; Mary Calderon; GB
123-45-6792; Oliver Aguilar; US
[...]
The Flight class is modified to include a distance field together with a
     getter and a setter for it.
```

Listing 22.14 The modified `Flight` class

```
private int distance;

public int getDistance() {
    return distance;
}
```

```
public void setDistance(int distance) {
    this.distance = distance;
}
```

To determine that the same passenger is on different flights, Thomas has to override the `equals` and `hashCode` methods in the `Passenger` class. He'll know that two passengers are the same if they have the same identifier.

Listing 22.15 Overridden `equals` and `hashCode` methods in 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);
   }

}
```

To distinguish between different flights, Thomas introduces the `FlightNumber` annotation, which receives the flight number as a parameter.

Listing 22.16 `FlightNumber` annotation

```
[...]
@Qualifier                                  (1)
@Retention(RUNTIME)                         (2)
@Target({FIELD, METHOD})                    (3)
public @interface FlightNumber {            (1)
    String number();                        (4)
}
```

In this listing:

- Thomas creates the `FlightNumber` annotation (1). The information it provides about the annotated elements is kept during runtime (2).
- This annotation can be applied to fields and methods (3) and has a number `parameter` (4).

In the `FlightProducer` class, Thomas annotates the `createFlight` method with the new `FlightNumber` annotation, using `"AA1234"` as an argument to give the flight an identity.

Listing 22.17 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 also changes the `FlightWithPassengersTest` class to annotate the injected flight and writes a test for the distance manager.

Listing 22.18 Modified `FlightWithPassengersTest` class

```
@RunWith(Arquillian.class)
public class FlightWithPassengersTest {
    [...]

    @Inject
    @FlightNumber(number= "AA1234")
    Flight flight;

    @Mock
    DistancesManager distancesManager;

    @Rule
    public MockitoRule mockitoRule = MockitoJUnit.rule();

    private static Map<Passenger, Integer> passengersPointsMap =
        new HashMap<>();

    @BeforeClass
    public static void setUp() {
        passengersPointsMap.put(new Passenger("900-45-6809",
                                "Susan Todd", "GB"), 210);
        passengersPointsMap.put(new Passenger("900-45-6797",
                                "Harry Christensen", "GB"), 420);
        passengersPointsMap.put(new Passenger("123-45-6799",
                                "Bethany King", "US"), 630);
    }

     [...]

    @Test
    public void testFlightsDistances() {
      when(distancesManager.getPassengersPointsMap()).
           thenReturn(passengersPointsMap);

      assertEquals(210, distancesManager.getPassengersPointsMap().
      get(new Passenger("900-45-6809", "Susan Todd", "GB")).longValue());
      assertEquals(420, distancesManager.getPassengersPointsMap().
```

1
2
3
4
5
6
7
8

```
            get(new Passenger("900-45-6797", "Harry Christensen", "GB"))
            .longValue());
        assertEquals(630, distancesManager.getPassengersPointsMap()
            .get(new Passenger("123-45-6799", "Bethany King", "US"))
            .longValue());
    }
}
```

8

In this listing:

- Thomas annotates the `flight` field with the `FlightNumber` annotation (1) to give an identity to the injected flight.
- He provides a mock implementation of the `DistancesManager` object and annotates it with `@Mock` (2).
- He declares a `MockitoRule` `@Rule` annotated object, which is needed to allow the initialization of mocks annotated with `@Mock` (3). Remember that Thomas is using Arquillian, which is not compatible with JUnit 5, and he needs to use the rules from JUnit 4.
- He declares `passengersPointsMap` to keep the bonus points for the passengers (4) and populates it with some expected values (5).
- He creates `testFlightsDistances` (6), where he instructs the mock object to return `passengersPointsMap` when he calls `distancesManager.getPassengersPointsMap()` (7). Then he checks that the bonus points are as expected (8). For now, he only inserts some data into the map and checks that it is correctly retrieved. However, he defines a tests skeleton and expects new functionalities from the provider side.

22.4.2 Testing with a partially implemented external dependency

From the provider side, Thomas receives the partial implementation of the `DistancesManager` class.

Listing 22.19 Modified implementation of the `DistancesManager` class

```
public class DistancesManager {

  private static final int DISTANCE_FACTOR = 10;

  private Map<Passenger, Integer> passengersDistancesMap =
                                  new HashMap<>();
  private Map<Passenger, Integer> passengersPointsMap = new HashMap<>();

  public Map<Passenger, Integer> getPassengersDistancesMap() {
       return Collections.unmodifiableMap(passengersDistancesMap);
  }

  public Map<Passenger, Integer> getPassengersPointsMap() {
       return Collections.unmodifiableMap(passengersPointsMap);
   }
```

1

2

```
    public void addDistance(Passenger passenger, int distance) {
    }

    public void calculateGivenPoints() {
       for (Passenger passenger : getPassengersDistancesMap().keySet()) {
         passengersPointsMap.put(passenger,
           getPassengersDistancesMap().get(passenger)/ DISTANCE_FACTOR);
       }
    }

}
```

In this listing:

- Thomas defines the `DISTANCE_FACTOR` constant by which to divide the distances, in order to get the number of bonus points (1).
- He keeps `passengersDistancesMap` and `passengersPointsMap` and provides getters for them (2).
- The `addDistance` method is still not implemented (3). The `calculateGivenPoints` method has an implementation that browses `passengersDistancesMap` and divides the distances by `DISTANCE_FACTOR` to populate `passengersPointsMap` (4).

In real applications, you may receive some fully implemented packages or classes while another part is still under construction but follows the agreed-on contract. To simplify, we have reduced our example situation to a class with four methods, of which only one does not have an implementation. What is important is that the API contract is respected.

How will this change the tests on the consumer side? Thomas knows how to get the bonus based on the distance, so he doesn't keep `passengersPointsMap`, but rather `passengersDistancesMap`. The changes to `FlightWithPassengersTest` are shown in the following listing.

Listing 22.20 Modified `FlightWithPassengersTest` class

```
[...]
@RunWith(Arquillian.class)
public class FlightWithPassengersTest {
    [...]

    @Spy
    DistancesManager distancesManager;

    private static Map<Passenger, Integer> passengersDistancesMap =
        new HashMap<>();
```

```
    @BeforeClass
    public static void setUp() {
         passengersDistancesMap.put(new Passenger("900-45-6809",
                                    "Susan Todd", "GB"), 2100);
         passengersDistancesMap.put(new Passenger("900-45-6797",
                                    "Harry Christensen", "GB"), 4200);
         passengersDistancesMap.put(new Passenger("123-45-6799",
                                    "Bethany King", "US"), 6300);

    }

    [...]

    @Test
    public void testFlightsDistances() {
      when(distancesManager.getPassengersDistancesMap()).
          thenReturn(passengersDistancesMap);

      distancesManager.calculateGivenPoints();

      assertEquals(210, distancesManager.getPassengersPointsMap().
      get(new Passenger("900-45-6809", "Susan Todd", "GB")).longValue());
      assertEquals(420, distancesManager.getPassengersPointsMap().
           get(new Passenger("900-45-6797", "Harry Christensen", "GB"))
           .longValue());
        assertEquals(630, distancesManager.getPassengersPointsMap()
           .get(new Passenger("123-45-6799", "Bethany King", "US"))
           .longValue());
    }
}
```

In this listing:

- Thomas changes the annotation of the `DistancesManager` object to `@Spy` (1). With the previous `@Mock` annotation, he was mocking the whole `distancesManager` object. In order to indicate that he would like to mock only some methods and to keep the functionality of others, he replaces the `@Mock` annotation with `@Spy`.
- He initializes `passengersDistancesMap` (2) and populates it before the execution of the tests (3).
- In `testFlightsDistances`, he instructs the `distancesManager` object to return `passengersDistancesMap` when he calls `distancesManager.getPassengersDistancesMap()` (4). Then he calls the already implemented `calculateGivenPoints` (5) and checks that the bonus points are as expected after this calculation (6).

If we run the tests now, they are green. The code coverage is not 100%, as we are still waiting for the implementation of a method from `DistancesManager` to be able to test everything (figure 22.6).

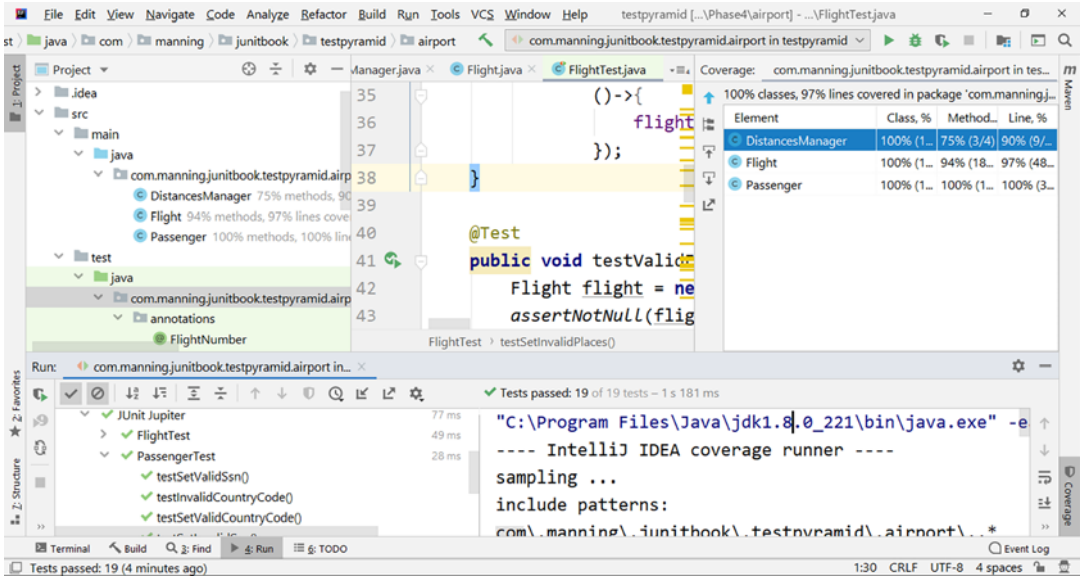


Figure 22.6 `PassengerTest`, `FlightTest`, and `FlightWithPassengersTest` **run successfully with a partial implementation of the** `DistancesManager` **class. The code coverage is not yet 100%, as some functionality is still missing.**

22.4.3 *Testing with the fully implemented external dependency*

On the consumer side, the real system is now fully available. Thomas needs to test the functionality against this real provider service. The full implementation of the `DistancesManager` class has introduced the `addDistance` method.

Listing 22.21 Modified implementation of the `DistancesManager` class

```
public class DistancesManager {
  [...]

  public void addDistance(Passenger passenger, int distance) {
       if (passengersDistancesMap.containsKey(passenger)) {
          passengersDistancesMap.put(passenger,
              passengersDistancesMap.get(passenger) + distance);
       } else {
          passengersDistancesMap.put(passenger, distance);
       }
  }

}
```

(1) (2) (3)

In this listing:

- In the `addDistance` method, Thomas checks whether `passengersDistanceMap` already contains a passenger (1).

- If that passenger already exists, he adds the distance to the passenger (2); otherwise, he creates a new entry in the map with that passenger as a key and the distance as the initial value (3).

Because Thomas will use the real passenger set for each flight and populate `passengersDistancesMap` based on the passengers' information, he introduces the `getPassengers` method into the `Flight` class.

Listing 22.22 Modified implementation of the `Flight` class

```
public class Flight {

    [...]
    private Set<Passenger> passengers = new HashSet<Passenger>();   (1)

    public Set<Passenger> getPassengers() {                         (2)
        return Collections.unmodifiableSet(passengers);
    }
    [...]
}
```

In this listing, Thomas makes `passengers` private (1) and exposes it through a getter (2).

Thomas now goes back to the test that previously mocked a behavior and introduces real behavior. He removes the Mockito dependencies from the pom.xml file, as they are no longer needed. He also removes the initialization of `passengersDistancesMap` before executing the tests. The changes to `FlightWithPassengersTest` are shown in the following listing.

Listing 22.23 Modified `FlightWithPassengersTest` class

```
@RunWith(Arquillian.class)
public class FlightWithPassengersTest {
    @Deployment
    public static JavaArchive createDeployment() {
        return ShrinkWrap.create(JavaArchive.class)
            .addClasses(Passenger.class, Flight.class,                  (1)
                        FlightProducer.class, DistancesManager.class)
            .addAsManifestResource(EmptyAsset.INSTANCE, "beans.xml");
    }

    @Inject
    @FlightNumber(number= "AA1234")
    Flight flight;

    @Inject
    @FlightNumber(number= "AA1235")                                     (2)
    Flight flight2;

    @Inject
    @FlightNumber(number= "AA1236")
    Flight flight3;
```

```
    @Inject
    DistancesManager distancesManager;                    (3)

    [...]
    @Test
    public void testFlightsDistances() {

        for (Passenger passenger : flight.getPassengers()) {
          distancesManager.addDistance(passenger, flight.getDistance());
        }

        for (Passenger passenger : flight2.getPassengers()) {
          distancesManager.addDistance(passenger, flight2.getDistance());   (4)
        }

        for (Passenger passenger : flight3.getPassengers()) {
          distancesManager.addDistance(passenger, flight3.getDistance());
        }

        distancesManager.calculateGivenPoints();          (5)

        assertEquals(210, distancesManager.getPassengersPointsMap()
           .get(new Passenger("900-45-6809", "Susan Todd", "GB"))
           .longValue());
        assertEquals(420, distancesManager.getPassengersPointsMap()
           .get(new Passenger("900-45-6797", "Harry Christensen", "GB"))   (6)
           .longValue());
        assertEquals(630, distancesManager.getPassengersPointsMap()
            .get(new Passenger("123-45-6799", "Bethany King", "US"))
            .longValue());
    }
}
```

In this listing:

- Thomas adds `DistancesManager.class` to the ShrinkWrap archive (1) so that the `DistancesManager` class can be injected into the test.
- He injects three flights into the test and annotates and differentiates them with the `@FlightNumber` annotation. The annotations have different arguments for each flight (2). He also injects a `DistancesManager` field (3).
- He changes the `testFlightsDistance` test to browse all passengers from the three flights and adds the distances they have traveled to `distancesManager` (4). Based on the traveled distances, he calculates the bonus points (5) and checks that the bonus points are as expected after this calculation (6).

If we run the tests now, they are green, and the code coverage is 100% (figure 22.7).

Thomas has successfully tested the entire system using a consumer-driven contract and has moved from a mock implementation of the external functionality to using the real thing. He is ready to proceed to the last step in building the test pyramid strategy: acceptance testing.

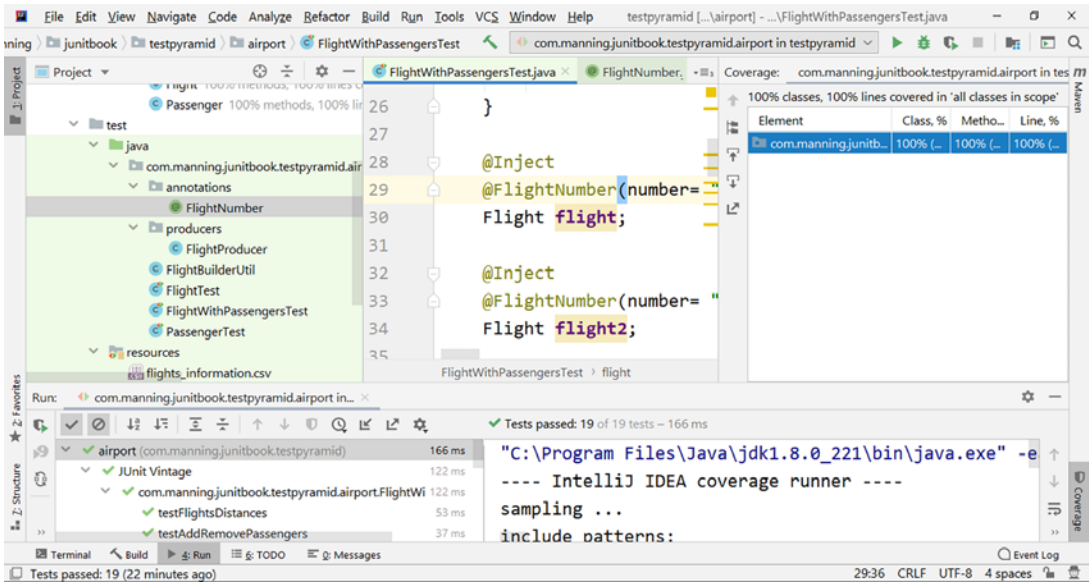


Figure 22.7 `PassengerTest`, `FlightTest`, **and** `FlightWithPassengersTest` **run successfully with a full implementation of the** `DistancesManager` **class, and the code coverage is 100%.**

22.5 Acceptance testing: Compliance with business requirements

Acceptance testing is the level of software testing where a system is tested for compliance with business requirements. Once the system testing has been fulfilled, acceptance testing is executed to confirm that the software is ready for delivery and will satisfy the needs of the end users.

In chapter 21, we discussed the working features that give business value to software, and the communication challenges that exist between customer, business analyst, developer, and 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`, and `Then`:

```
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 passengers to and removing them from 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, whereas VIP passengers cannot be removed from a flight.

This is the business logic that the application must be compliant with to satisfy the end user. To fulfill the acceptance testing, Thomas will use Cucumber, the acceptance testing framework that we used in chapter 21. Cucumber describes application scenarios in plain English, using the Gherkin language. Cucumber is easy for stakeholders to read and understand and allows automation.

To start working with Cucumber, Thomas first introduces the additional dependencies that are needed at the level of the Maven pom.xml file: `cucumber-java` and `cucumber-junit`.

Listing 22.24 Cucumber dependencies in 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 introduce the new feature by working TDD/BDD style, as we demonstrated in chapters 20 and 21. This means he will first write the acceptance tests in Cucumber. Then he will write the tests in Java. Finally, he will write the code that fixes the tests and consequently provides the feature implementation.

Thomas introduces the two types of passengers: regular and VIP. This changes the `Passenger` class by adding a `boolean vip` field, together with a getter and a setter.

Listing 22.25 Modified `Passenger` class

```
public class Passenger {

   [...]
   private boolean vip;

   public boolean isVip() {
      return vip;
   }

   public void setVip(boolean vip) {
      this.vip = vip;
   }
[...]
}
```

Next, to build the scenarios that express the acceptance criteria, Thomas creates the passengers_policy.feature file in the new test/resources/features folder. He reuses the three CSV flight files that were used for the system integration testing. The scenarios that Thomas defines are presented in the following listing and can be read using natural language. To review the capabilities of Cucumber, see chapter 21.

Listing 22.26 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    |
```

To generate the skeleton of the Java tests, Thomas makes sure the Cucumber and Gherkin plugins are installed by accessing the File > Settings > Plugins menu (figures 22.8 and 22.9).

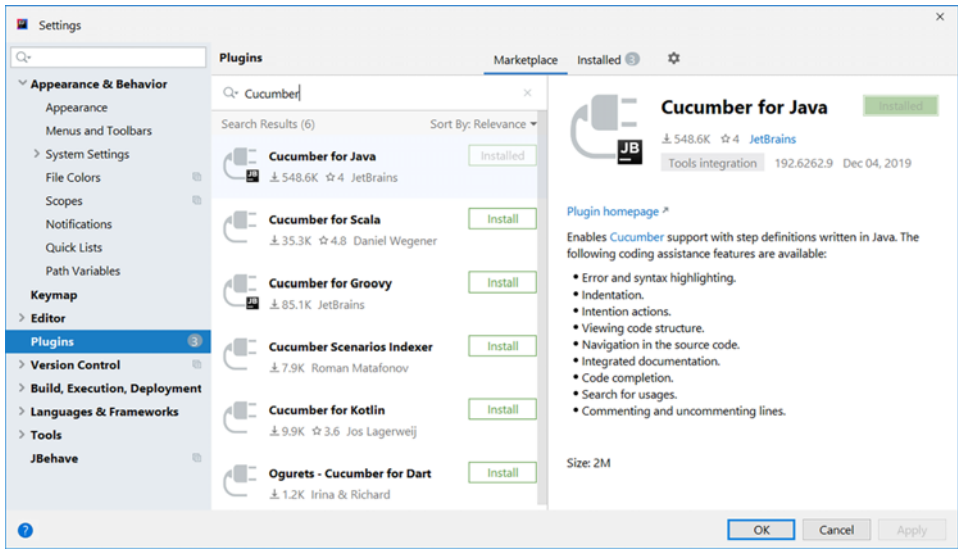


Figure 22.8 Installing the Cucumber for Java plugin from the File > Settings > Plugins menu

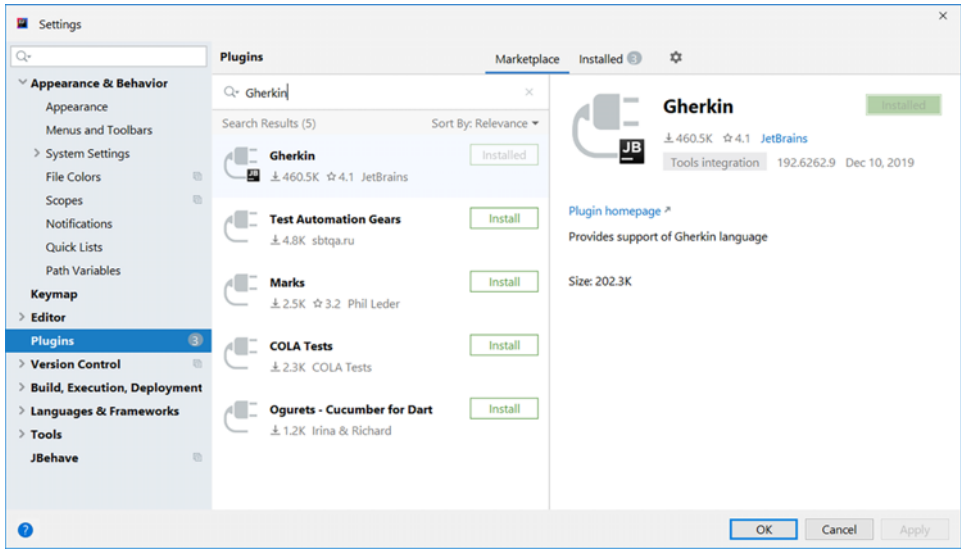


Figure 22.9 Installing the Gherkin plugin from the File > Settings > Plugins menu

Now he must configure the way the feature is run by going to Run > Edit Configurations and choosing a few settings (figure 22.10):

- 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

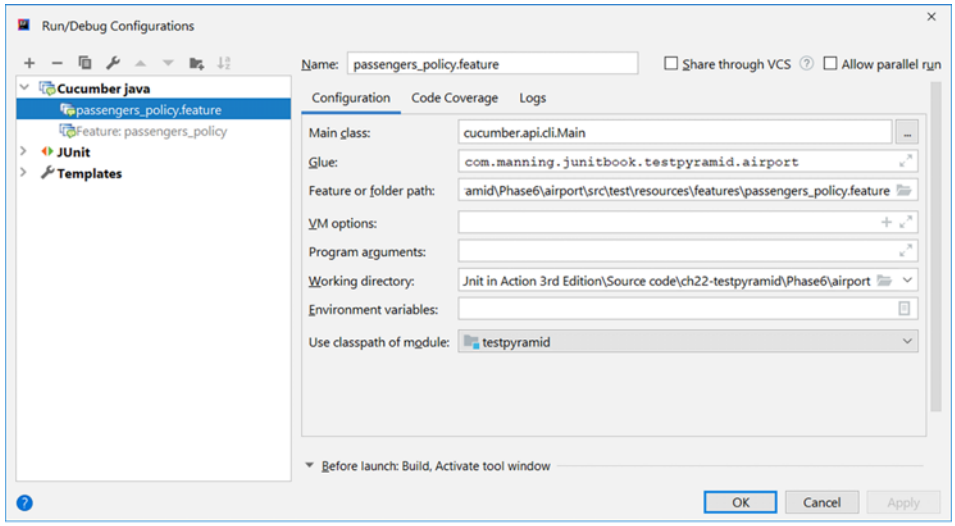


Figure 22.10 Setting the feature configuration by filling in the Main Class, Glue, Feature or Folder Path, and Working Directory fields

After setting this configuration, Thomas can run the feature file directly by right-clicking it (figure 22.11). He gets the skeleton of the Java tests (figure 22.12).

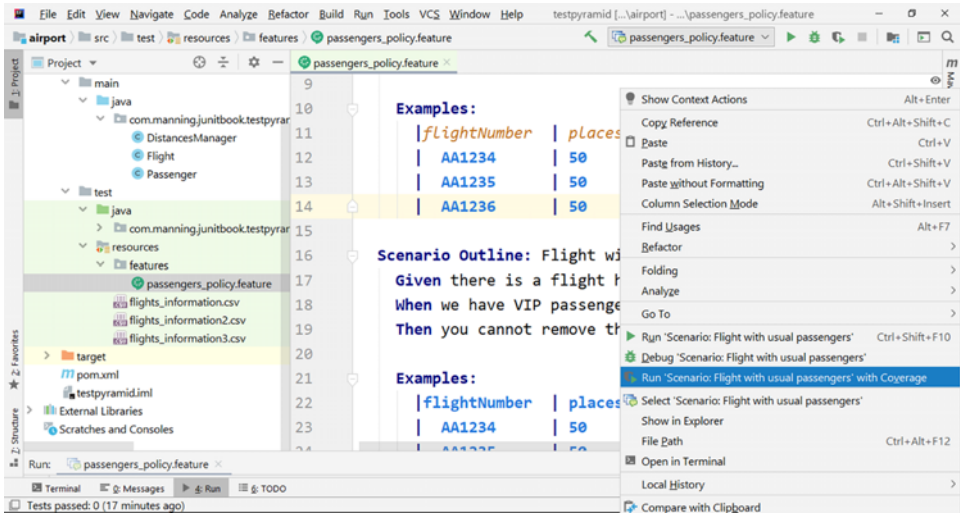


Figure 22.11 Directly running the passengers_policy.feature file by right-clicking it

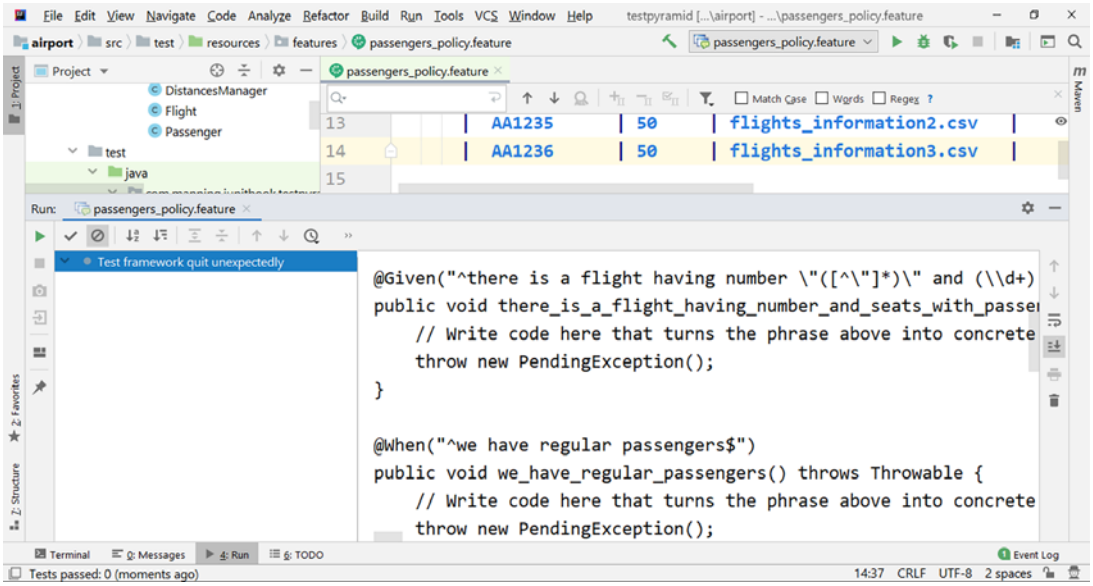


Figure 22.12 Generating the test skeleton for the passengers policy by directly running the feature file

Thomas now creates the `PassengersPolicy` file that contains the skeleton of the tests to be executed.

Listing 22.27 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();
    }
}
```

To run the Cucumber tests, Thomas needs a special class. The name of the class could be anything; he chooses `CucumberTest`.

Listing 22.28 `CucumberTest` class

```
[...]
@RunWith(Cucumber.class)
@CucumberOptions(
    plugin = {"pretty"},
    features = "classpath:features")
public class CucumberTest {
    /**
     * This class should be empty, step definitions should be in separate
           classes.
     */

}
```

In this listing:

- Thomas annotates this class with `@RunWith(Cucumber.class)` (1). Executing it like 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 time of writing, he uses the JUnit 4 runner.
- The `@CucumberOptions` (2) annotation provides the plugin option (3) that is used to specify different formatting options for the output reports. Using `"pretty"` prints the Gherkin source with additional colors. The `features` option (4) helps Cucumber to locate the feature file in the project folder structure. It looks for the features folder on the classpath—and remember that the src/test/resources folder is maintained by Maven on the classpath!

Thomas next turns back to the `PassengersPolicy` class and writes the tests to check the functionality of the `PassengersPolicy` feature to be introduced.

Listing 22.29 `PassengersPolicy` test class

```
public class PassengersPolicy {
    private Flight flight;                                          (1)
    private List<Passenger> regularPassengers = new ArrayList<>();  (2)
    private List<Passenger> vipPassengers = new ArrayList<>();
    private Flight anotherFlight = new Flight("AA7890", 48);        (3)

    @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,  (4)
                 seats,"src/test/resources/" + fileName);
    }

    @When("^we have regular passengers$")
    public void we_have_regular_passengers() {
        for (Passenger passenger: flight.getPassengers()) {          (5)
            if (!passenger.isVip()) {
                regularPassengers.add(passenger);
            }
        }
    }

    @Then("^you can remove them from the flight$")
    public void you_can_remove_them_from_the_flight() {
        for(Passenger passenger: regularPassengers) {                (6)
            assertTrue(flight.removePassenger(passenger));
        }
    }

    @Then("^add them to another flight$")
    public void add_them_to_another_flight() {
        for(Passenger passenger: regularPassengers) {                (7)
            assertTrue(anotherFlight.addPassenger(passenger));
        }
    }

    @When("^we have VIP passengers$")
    public void we_have_VIP_passengers() {
        for (Passenger passenger: flight.getPassengers()) {          (8)
            if (passenger.isVip()) {
                vipPassengers.add(passenger);
            }
        }
    }
```

```
    @Then("^you cannot remove them from the flight$")
    public void you_cannot_remove_them_from_the_flight(){
        for(Passenger passenger: vipPassengers) {
            assertFalse(flight.removePassenger(passenger));
        }
    }
}
```

9

In this listing:

- Thomas defines as fields the flight from which passengers are moved 1, the list of regular and VIP passengers 2, and the second flight to which the passengers are moved 3.
- The step labeled `Given there is a flight having number "<flightNumber>" and <seats> seats with passengers defined into "<file>"` initializes the flight from the CSV file 4.
- Thomas browses the list of all passengers, adds the regular passengers to their list 5, checks that he can remove them from a flight 6, and adds them to another flight 7.
- He browses the list of all passengers, adds the VIP passengers to their list 8, and checks that he cannot remove them from a flight 9.

Running the tests now by executing `CucumberTest`, Thomas obtains the result shown in figure 22.13.

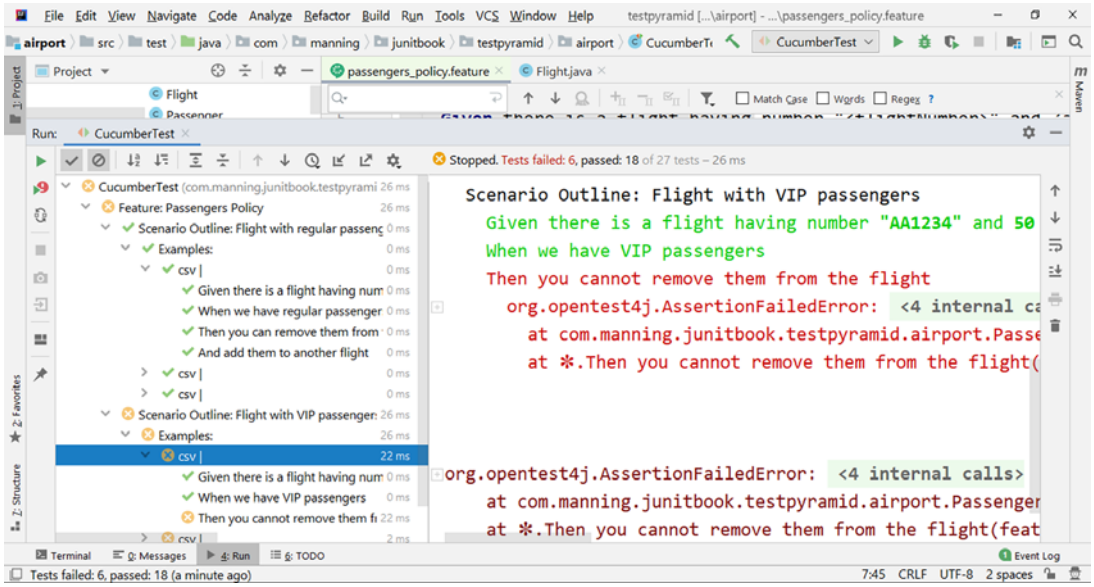


Figure 22.13 Running the newly introduced `PassengersPolicy` tests results in failure only for VIP passengers.

Only the tests concerning VIP passengers fail, so Thomas knows that he only needs to change the code regarding this type of passenger: he will modify the `addPassenger` method from the `Flight` class.

Listing 22.30 Modified `Flight` class

```
public class Flight {

[...]
    public boolean removePassenger(Passenger passenger) {
        if(passenger.isVip()) {                          (1)
            return false;
        }
        return  passengers.remove(passenger);
    }
}
```

In this listing, Thomas introduces the condition that a VIP passenger cannot be removed from a flight (1). Running the tests now by executing `CucumberTest`, Thomas obtains the result shown in figure 22.14—all tests execute successfully.

To check the code coverage, Thomas executes all of the tests that form the test pyramid. When he does, the code coverage is 100% (figure 22.15). He has successfully implemented a test pyramid for the flight-management application including unit tests, integration tests, system tests, and acceptance tests.

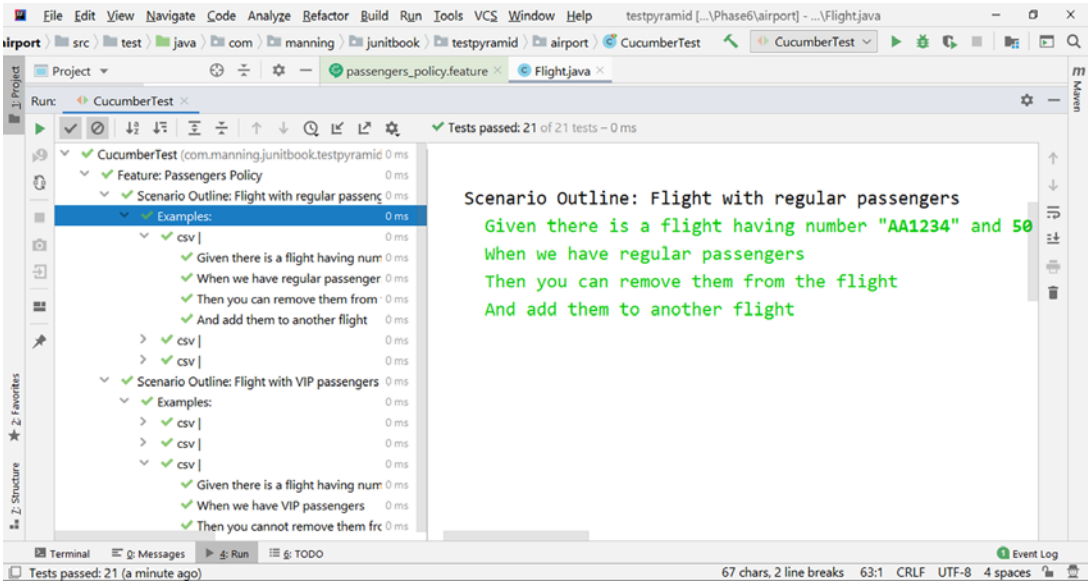


Figure 22.14 Successfully running the newly introduced `PassengersPolicy` test after writing the business logic

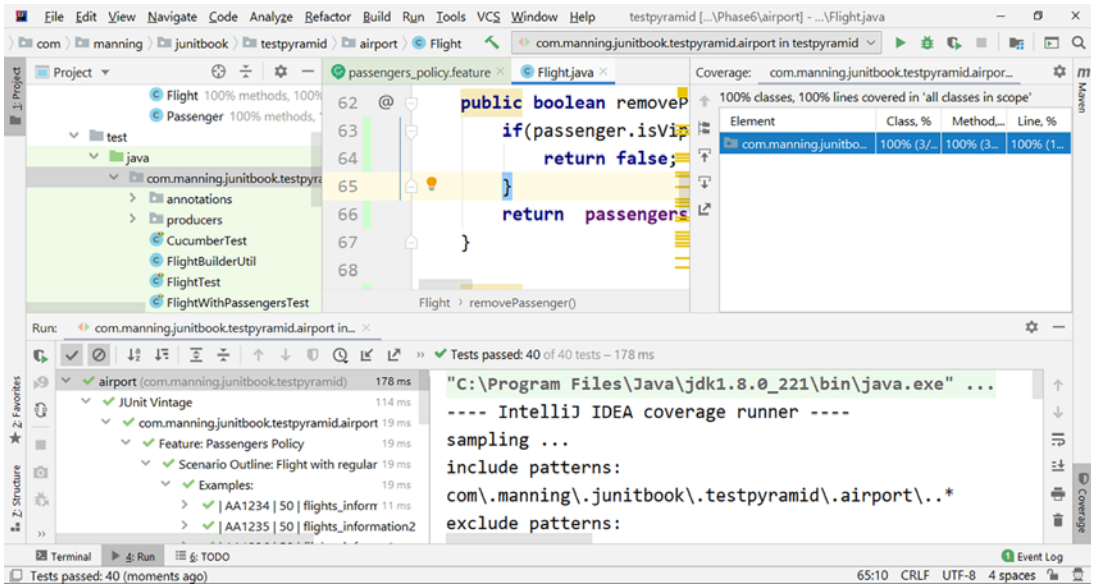


Figure 22.15 The code coverage is 100% after implementing the entire test pyramid.

Summary

This chapter has covered the following:

- Introducing the software testing levels (unit, integration, system, and acceptance) and the concept of a test pyramid (from simpler levels to more complex ones)
- Analyzing what software tests should verify: business logic, bad input values, boundary conditions, unexpected conditions, invariants, and regressions
- Developing unit tests, and testing values restrictions, bad input values, and boundary conditions
- Developing tests to verify the integration between two classes using the Arquillian framework, and testing how these classes interact
- Developing system tests using a consumer-driven contract and moving from a mock implementation of an external functionality to using the real thing
- Developing acceptance tests for a new feature to satisfy external policies with the help of JUnit 5 and Cucumber

appendix A
Maven

Maven (https://maven.apache.org) can be regarded a source-building *environment*. To better understand how Maven works, you need to understand the key points (principles) that stand behind Maven. From the very beginning of the Maven project, certain ground rules were created for software architecture. These rules aimed to simplify development with Maven and make it easier for developers to implement the build system. One of the fundamental ideas of Maven is that the build system should be as simple as possible: software engineers should not spend a lot of time implementing the build system. It should be easy to start a new project from scratch and then rapidly begin developing the software, rather than spending valuable time designing and implementing a build system. This appendix describes the core Maven principles in detail and explains what they mean from a developer's point of view.

A.1 Convention over configuration

Convention over configuration is a software design principle that aims to decrease the number of configurations a software engineer needs to make, instead of introducing conventional rules that we must follow strictly. This way, we can skip the tedious configuration that needs to be done for every single project and focus on the more important parts of our work.

Convention over configuration is one of the strongest principles of the Maven project. One example of its application is the folder structure of the build process. With Maven, all the directories we need are already defined for us. src/main/java/, for example, is the Maven convention for where Java code for the project resides, src/test/java is where the unit tests for the project reside, target is the build folder, and so on.

That sounds great, but aren't we losing the flexibility of the project? What if we want our source code to reside in another folder? Maven is easy to configure: it

provides the convention, but at any point, we can override the convention and use the configuration of our choice.

A.2 Strong dependency management

Strong dependency management is the second key point that Maven introduced. When the Maven project began, the de facto build system for Java projects was another build tool, Ant. With Ant, we have to distribute the dependencies of our project, which means each project must take care of the dependencies it requires, and the dependencies of the same project may be distributed across different locations. Also, the same dependency may be used by different projects but located in a different place for each project, causing duplication of resources.

Maven introduced the notion of a *central repository*: a location on the internet where all kinds of artifacts (dependencies) are stored. The Maven build tool resolves these artifacts by reading a project's build descriptor, downloading the necessary versions of the artifacts, and including them in the classpath of the application. This way, we need to list our dependencies only once, in the dependencies section of our build descriptor. For example:

```
<dependencies>
     <dependency>
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter-api</artifactId>
        <version>5.6.0</version>
        <scope>test</scope>
     </dependency>
     <dependency>
        <groupId>org.jmock</groupId>
        <artifactId>jmock-junit5</artifactId>
        <version>2.12.0</version>
     </dependency>
  </dependencies>
```

Thereafter, we are free to build the software on any other machine. We don't need to bundle the dependencies with our project.

Maven also introduced the concept of the local repository: a folder on a hard disk (~/.m2/repository/ in UNIX and C:\Documents and Settings\<UserName>\.m2\repository\ in Windows) where Maven keeps the artifacts that it just downloaded from the central repository. After we build our project, our artifacts are installed in the local repository for later use by other projects, which is simple and neat.

A developer might join a project managed by Maven and need access only to the sources of the project. Maven downloads the needed dependencies from the central repository and brings them to the local repository, where they will be available for other projects that the same developer may work on.

A.3 Maven build life cycles

Another very strong principle in Maven is the *build life cycle*. The Maven project is built around the idea of defining the process of building, testing, and distributing a particular artifact. A Maven project can produce only one artifact. This way, we can use Maven to build the project artifact, clean the project's folder structure, or generate the project documentation. Following are the three built-in Maven life cycles:

- *Default*—For generating the project artifact
- *Clean*—For cleaning the project
- *Site*—For generating the project documentation

Each of these life cycles is composed of several phases. To navigate a certain life cycle, the build follows its phases (figure A.1).

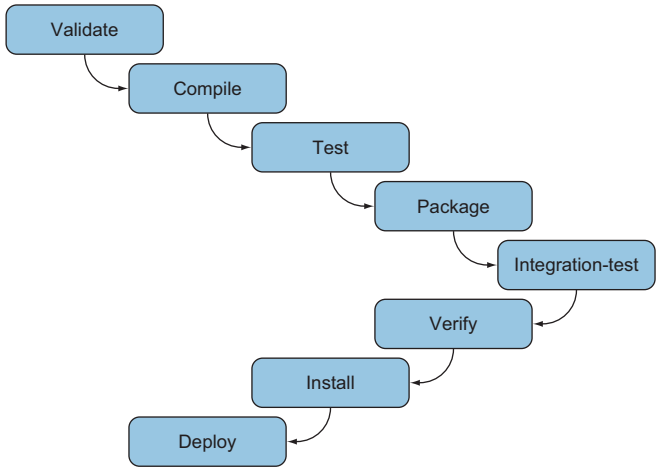


Figure A.1 The phases of Maven's default life cycle, from validate to deploy

Following are the phases of the default life cycle:

1. *Validate*—Validate that the project is correct and all necessary information is available.
2. *Compile*—Compile the source code of the project.
3. *Test*—Test the compiled source code using a suitable unit testing framework (perhaps JUnit 5, in this case). The test should not require the code to be packaged or deployed.
4. *Package*—Package the compiled code in its distributable format, such as a jar file.

5. *Integration-test*—Process and deploy the package (if necessary) into an environment where integration tests can be run.
6. *Verify*—Run any checks to verify that the package is valid and meets quality criteria.
7. *Install*—Install the package in the local repository for use as a dependency in other projects locally.
8. *Deploy*—In an integration or release environment, copy the final package to the remote repository for sharing with other developers and projects.

Here, again, is the convention-over-configuration principle promoted by Maven. These phases are already defined in the order in which they are listed here. Maven invokes these phases in a very strict order; the phases are executed sequentially in the order in which they are listed here, to complete the life cycle. If we invoke any of these phases—if we type

```
mvn compile
```

on the command line in our project home directory, for example—Maven first validates the project and then tries to compile the sources of the project.

One last thing: it is useful to think of all these phases as extension points. We can attach additional Maven plugins to the phases and orchestrate the order and the way in which these plugins are executed.

A.4 Plugin-based architecture

The last feature of Maven that we will mention here is its plugin-based architecture. We mentioned that Maven is a source-building environment. More specifically, Maven is a plugin-execution source-building environment. The core of the project is very small, but the architecture of the project allows multiple plugins to be attached to the core. This way, Maven builds an environment in which different plugins can be executed.

Each phase in a given life cycle has several plugins attached, and Maven invokes them when passing through the given phase in the order in which the plugins are declared. Here are some of the core Maven plugins:

- *Clean*—Cleans up after the build
- *Compiler*—Compiles Java sources
- *Deploy*—Deploys the built artifact to the remote repository
- *Install*—Installs the built artifact in the local repository
- *Resources*—Copies the resources to the output directory for inclusion in the jar file
- *Site*—Generates a site that includes information about the current project
- *Surefire*—Runs the JUnit tests in an isolated classloader
- *Verifier*—Verifies the existence of certain conditions (useful for integration tests)

In addition to these core Maven plugins, other Maven plugins are available for many situations, such as WAR (for packaging a web application) and Javadoc (for generating project documentation).

Plugins are declared in the `plugins` section of the build configuration file, as in this example:

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

A plugin declaration can have a `groupId`, `artifactId`, and `version`. This way, the plugins look like dependencies. In fact, plugins are handled the same way as dependencies; they are downloaded to the local repository like dependencies. When we specify a plugin, the `groupId` and `version` parameters are optional; if we do not declare them, Maven looks for a plugin with the specified `artifactId` and one of the following `groupIds`: `org.apache.maven.plugins` or `org.codehaus.mojo`. As the version is optional, Maven tries to download the latest available plugin version. Specifying the plugin versions is highly recommended to prevent auto-updates and nonreproducible builds. We may have built our project with the most recently updated Maven plugin; but later, if another developer tries to make the same build with the same configuration, and if the Maven plugin has since been updated, using the newest update may result in a nonreproducible build.

A.5 The Maven project object model (POM)

Maven has a build descriptor called pom.xml (short for *project object model*) by default. We do not imperatively specify the things we want to do. We declaratively specify general information for the project itself, as in the following listing.

Listing A.1 Very simple pom.xml

```
<project>
   <modelVersion>4.0.0</modelVersion>
   <groupId>com.manning.junitbook</groupId>
   <artifactId>example-pom</artifactId>
   <packaging>jar</packaging>
   <version>1.0-SNAPSHOT</version>
</project>
```

This code looks really simple, doesn't it? But one big question may arise: how is Maven capable of building source code with so little information?

The answer lies in the inheritance feature of the pom.xml files. Every simple pom.xml inherits most of its functionality from a Super POM. As in Java, in which every class inherits certain methods from the `java.lang.Object` class, the Super POM empowers each pom.xml with Maven features.

We can see the analogy between Java and Maven. To take this analogy even further, Maven pom.xmls can inherit from one another; just as in Java, some classes can act as

parents for others. If we want to use the pom from listing A.1 for our parent, all we have to do is change its packaging value to pom. Parent and aggregation (multimodule) projects can have pom only as a packaging value. We also need to define in our parent which modules are the children.

Listing A.2 Parent pom.xml with a child module

```
<project>
   <modelVersion>4.0.0</modelVersion>
   <groupId>com.manning.junitbook</groupId>
   <artifactId>example-pom</artifactId>
   <packaging>pom</packaging>                    (1)
   <version>1.0-SNAPSHOT</version>
   <modules>
         <module>example-module</module>        (2)
   </modules>
</project>
```

Listing A.2 is an extension of listing A.1. We declare that this pom is an aggregation module by declaring the package to be of pom type (1) and adding a modules section (2). The modules section lists all the children modules that our module has by providing the relative path to the project folder (example-module, in this case). The following listing shows the child pom.xml.

Listing A.3 pom.xml that inherits the parent pom.xml

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>com.manning.junitbook</groupId>
    <artifactId>example-pom</artifactId>
    <version>1.0-SNAPSHOT</version>
  </parent>
  <artifactId>example-child</artifactId>
</project>
```

Remember that this pom.xml resides in the folder that the parent XML has declared (example-module, in this case).

Two things are worth noticing here. First, because we inherit from some other pom, we don't need to specify groupId and version for the child pom; second, Maven expects the values to be the same as in the parent.

Going further with the analogy of Java, it seems reasonable to ask what kinds of objects poms can inherit from their parents. Here are all the elements that a pom can inherit from its parent:

- Dependencies
- Developers and contributors
- Plugins and their configurations
- Reports lists

Each of these elements specified in the parent `pom` is automatically specified in the child `pom`.

A.6 Installing Maven

Installing Maven is a three-step process:

1. Download the latest distribution from https://maven.apache.org, and unzip/untar it in the directory of your choice.
2. Define an `M2_HOME` environment variable pointing to where you have installed Maven.
3. Add `M2_HOME\bin` (`M2_HOME/bin` on `UNIX`) to your `PATH` environment variable so that you can type `mvn` from any directory.

appendix B
Gradle

B.1 Installing Gradle

Gradle runs on all major OSs and requires only an installed Java JDK or JRE version 8 or higher. You can download the Gradle distribution from https://gradle.org/releases. At the time of writing, the latest version was 6.0.1. For our purpose, downloading the binary distribution will be enough. You can choose to make operations with the help of the IDE and the Gradle plugin, but our demonstration will be made through the command line, better touching the functionalities.

As Windows is the most commonly used OS, our example configuration details are on Windows. Concepts such as the path, environment variables, and the command prompt also exist in other OSs. Please follow your documentation guidelines if you run the examples on an OS other than Windows.

Unzip the downloaded file into a folder of your choice; we'll call it GRADLE_HOME. Then add the GRADLE_HOME\bin folder to your path. To do this in Windows, go to This PC, right-click it, and choose Properties > Advanced System Settings. Click Environment Variables, and you will get the window shown in figure B.1.

From here, choose Path and click Edit. This will open the window in figure B.2. Click the New button on the right, and choose to add the GRADLE_HOME\bin folder to your path. On our machine, we unzipped Gradle in the C:\kits\gradle-6.0.1 folder, so this is our GRADLE_HOME. We have added C:\kits\gradle-6.0.1\bin to the path.

To check the results, open a command prompt and type `gradle -version`. On our machine, we got the result shown in figure B.3.

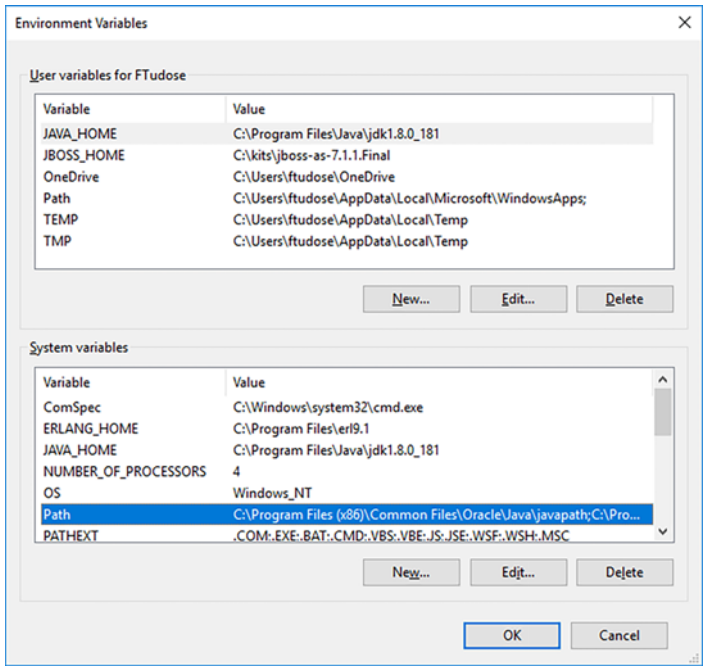


Figure B.1 Accessing the Environment Variables window

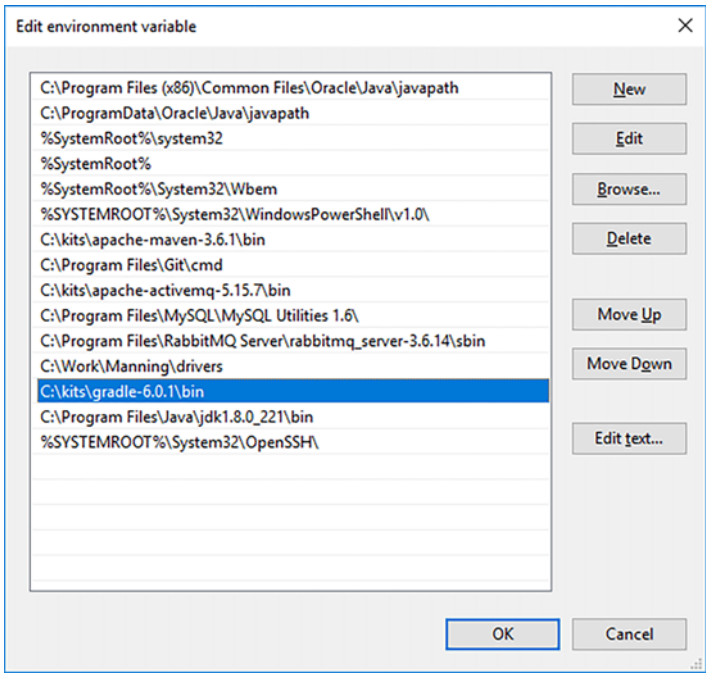


Figure B.2 Adding the GRADLE_HOME\bin folder to the path

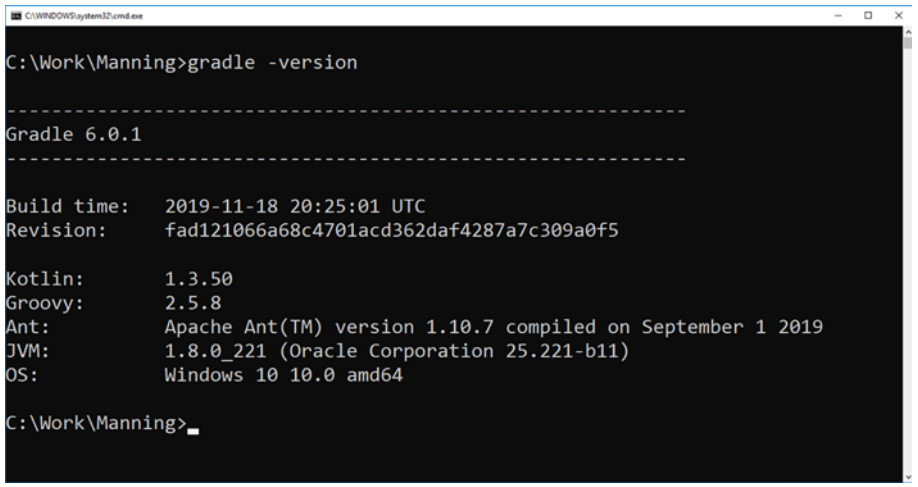


Figure B.3 Executing `gradle -version` at the command prompt

B.2 Creating Gradle tasks

Gradle manages a *build file* to handle projects and tasks. Every Gradle build file represents one or more projects. Further, a project consists of different tasks. A *task* represents a piece of work performed by running a build file. A task may compile some classes, create a JAR, generate a Javadoc, or publish archives to a repository. Tasks give us control over defining and executing the actions that are needed for building and testing the project. A Gradle *closure* is a standalone block of code that can take arguments or return values.

The Gradle build file is named build.gradle by default. To describe builds, Gradle uses a domain-specific language (DSL) based on Groovy. Listing B.1 shows a script defining a simple task named `junit` containing a closure that prints "JUnit in Action." If you create a build.gradle file with the content from this listing and execute the `gradle -q junit` command in the containing folder, you will get the result shown in figure B.4.

Listing B.1 build.gradle file containing one simple task

```
task junit {
    println "JUnit in Action"
}
```

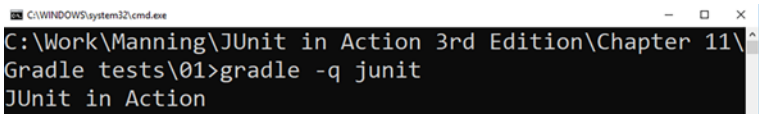


Figure B.4 Executing the command `gradle -q junit`

A task may be dependent on another task. This means a dependent task can start only when the task it depends on is finished. Each task is defined using a task name.

Listing B.2 defines two tasks called `junit` and `third`. The `third` task depends on the `junit` task. Each task contains one closure, each of which prints a message. So, if you save this content into a build.gradle file and execute `gradle -q third` in the containing folder, you will get the result shown in figure B.5.

Listing B.2 build.gradle file containing two dependent tasks

```
task junit {
    print "JUnit in Action"
}

task third (dependsOn: 'junit'){
    println ", third edition"
}
```

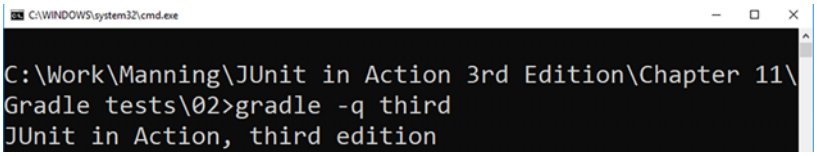


Figure B.5 Executing the command `gradle -q third`

In this listing:

- We try to execute the task `third`. It is dependent on the task `junit`. So, first, the closure 1 from this task is executed.
- The closure 2 of the task `third` is executed.

Working with tasks, Gradle defines different phases. First there is a configuration phase, where the code, which is specified directly in the task closure, is executed. The configuration block is executed for every available task, not only for those tasks that are executed later. After the configuration phase, the execution phase runs the code in the `doFirst` or `doLast` closure of those tasks, which are executed. If you save this content into a build.gradle file and execute the `gradle -q third` command in the containing folder, you will get the result shown in figure B.6.

Listing B.3 build.gradle file with two dependent tasks, each with multiple phases

```
task junit {
    print "JUnit "

    doFirst {
          print "Action"
       }
```

```
        doLast {
           print ", "
        }
}

task third (dependsOn: 'junit') {
    print "in "

    doFirst {
           print "third "
        }

        doLast {
           println "edition"
        }
}
```

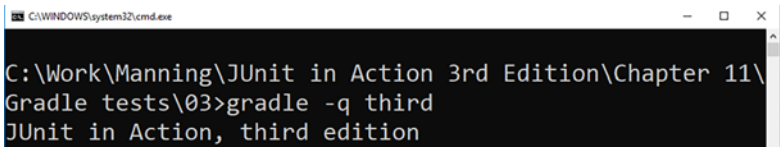


Figure B.6 Executing the command `gradle -q junit` with tasks containing multiple phases

In this listing:

- We try to execute the task `third`. It is dependent on the task `junit`. So, first, the configuration phase ❶ of this task is executed.
- The configuration phase ❹ of the task `third` is executed.
- The `doFirst` phase ❷ of the task `junit` is executed.
- The `doLast` phase ❸ of the task `junit` is executed.
- The `doFirst` phase ❺ of the task `third` is executed.
- The `doLast` phase ❻ of the task `third` is executed.

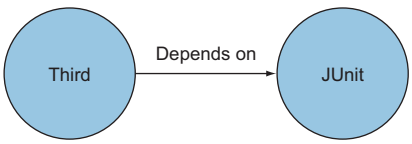


Figure B.7 The directed acyclic graph built by Gradle for listing B.3

Gradle determines the order in which tasks are run by using a directed acyclic graph. For the particular case of listing B.3, it is shown in figure B.7.

Gradle provides many possibilities to create and configure tasks. A detailed discussion of them would exceed the purpose of this appendix. For more information regarding Gradle tasks, we recommend *Gradle in Action* by Benjamin Muschko (Manning, https://www.manning.com/books/gradle-in-action). What we have discussed here is the "just-in-time" information you need to pursue the road to using the Gradle tasks for working with JUnit 5.

appendix C
IDEs

Theoretically, developing Java programs is possible with the help of a simple editor, while compiling and executing can be done at the command line. Practically, this process represents a significant burden and is extremely time consuming. You would have to struggle with the infrastructure and with tedious activities, instead of focusing on writing code.

There still may be some benefit to writing code without an IDE—for learning purposes. Writing very simple applications consisting of one or two short classes and compiling and executing them through the command line may be an excellent introduction for a new Java programmer. Otherwise, you should take advantage of all the benefits of comfortably working with your favorite IDE, which will greatly speed up your development process.

An IDE generally consists of at least a source code editor, build-automation tools, the compiler, and the debugger. A *debugger* is a computer program that is used to test and debug the program under development, usually through step-by-step execution.

Additionally, modern IDEs provide features like these:

- Syntax highlighting
- Code completion
- Navigating between classes
- Easy find and replace
- Automatic generation of pieces of code
- Information about potential problems within the code
- Integration with source control version tools
- Support for refactoring (changing the internal structure of the code while keeping its observable behavior)

IDEs are intended to maximize productivity: all development is done in the same place, including creating, modifying, compiling, deploying, and debugging software.

C.1 Installing IntelliJ IDEA

The official website of IntelliJ IDEA is www.jetbrains.com. You can download either of the two versions—Community (Apache licensed) or Ultimate (proprietary commercial)—from www.jetbrains.com/idea/download. The Community edition is enough for our examples because it includes full support for JUnit 5.

To install IntelliJ, you have to run the downloaded installation kit. IntelliJ IDEA has supported running JUnit 5 tests since version 2016.2. At the time of writing, the latest version is 2019.2, so that's what we will use. The installation will produce a JetBrains/IntelliJ IDEA Community Edition 2019.2 folder, and the executable that will launch the IDE is found in JetBrains/IntelliJ IDEA Community Edition 2019.2/bin. Depending on the OS, you can choose to launch either the `idea` executable (for 32-bits OSs) or `idea64` (for 64-bits OSs). Figure C.1 shows a piece of the installation on the Windows 10 OS.

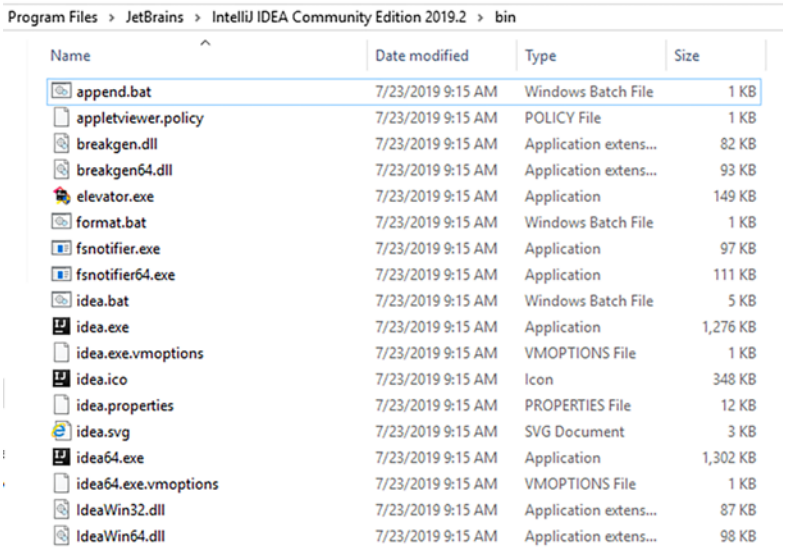


Figure C.1 A piece of the IntelliJ IDEA Community Edition 2019.2 installation

C.2 Installing Eclipse

The official website of Eclipse is www.eclipse.org. You can download it from www.eclipse.org/downloads. To install Eclipse, run the downloaded installation kit. Both Eclipse IDE for Java Developers and Eclipse IDE for Enterprise Java Developers provide support for JUnit 5, starting from Eclipse Oxygen.1a (4.7.1a) release. At the time of writing, the latest version was 2019-06, so that's what we will use. The installation will produce an eclipse/jee-2019-06/eclipse folder, and the executable that will launch the IDE is in this folder. Figure C.2 shows a piece of the installation on the Windows 10 OS.

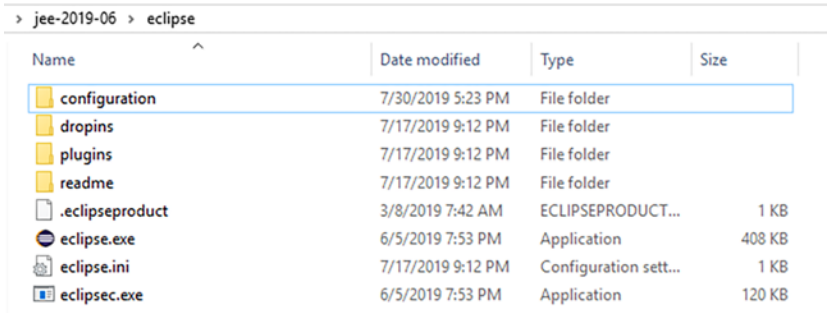


Figure C.2 A piece of the Eclipse 2019-06 installation

C.3 Installing NetBeans

The official website of NetBeans is https://netbeans.org/. You can download it from https://netbeans.apache.org/download/index.html. NetBeans can be downloaded as a zip archive and needs to be unpacked. It provides support for JUnit 5 starting from release 10.0. At the time of writing, the latest version was 11.1, so that's what we will use. Unpacking the archive will produce a netbeans folder, and the executable that will launch the IDE is in netbeans/bin. Figure C.3 shows a piece of the installation on the Windows 10 OS.

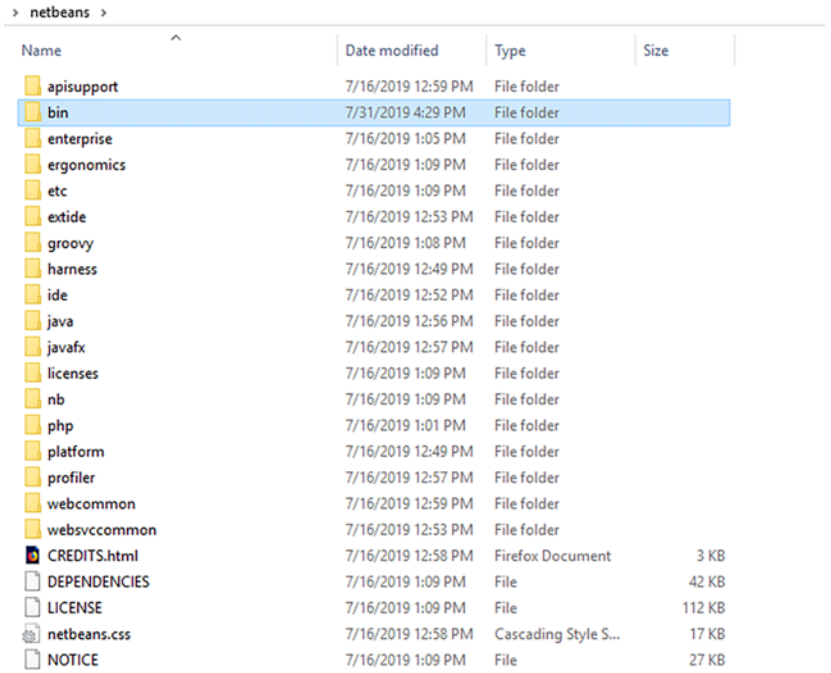


Figure C.3 A piece of the NetBeans 11.1 installation

appendix D
Jenkins

Jenkins (https://jenkins.io/) is an open source project for continuous builds. Like any other software for continuous builds, it relies on the idea of being able to continuously poll the source code from the source control system. If changes are detected, it fires up a build. It is very popular and is used in many projects.

Before installing Jenkins, make sure you have Java version 8 or 11 already installed. Older versions of Java are not supported, and versions 9, 10, and 12 are also not supported. The `JAVA_HOME` environment variable must point to the location where Java is installed.

The installation procedure itself is straightforward. Go to the project website, and download the latest version of Jenkins (figure D.1). At the time of writing, the latest version is 2.176.2.

Figure D.1 The Jenkins continuous integration tool can be downloaded from the official website.

Running on Windows, the Jenkins distribution comes as an installer file. Executing it will launch a wizard that will guide you through the setup (figure D.2). In our example, we are installing Jenkins into its default folder on Windows (figure D.3).

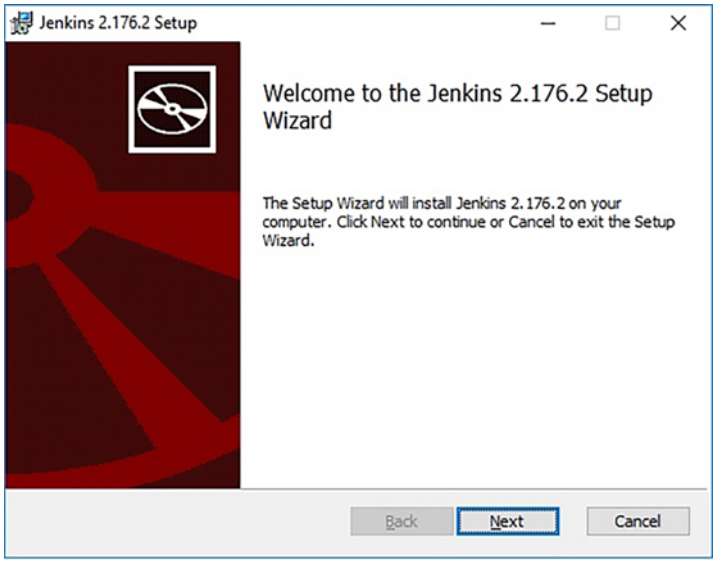


Figure D.2 Launching the Jenkins Setup Wizard

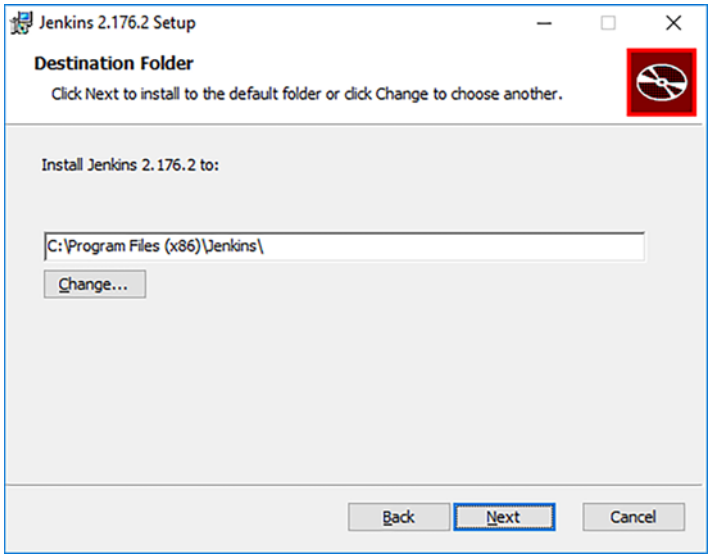


Figure D.3 Installing Jenkins in the Windows default folder

After the installation is completed, Windows creates a folder structure. The most important piece is the jenkins.war file (figure D.4).

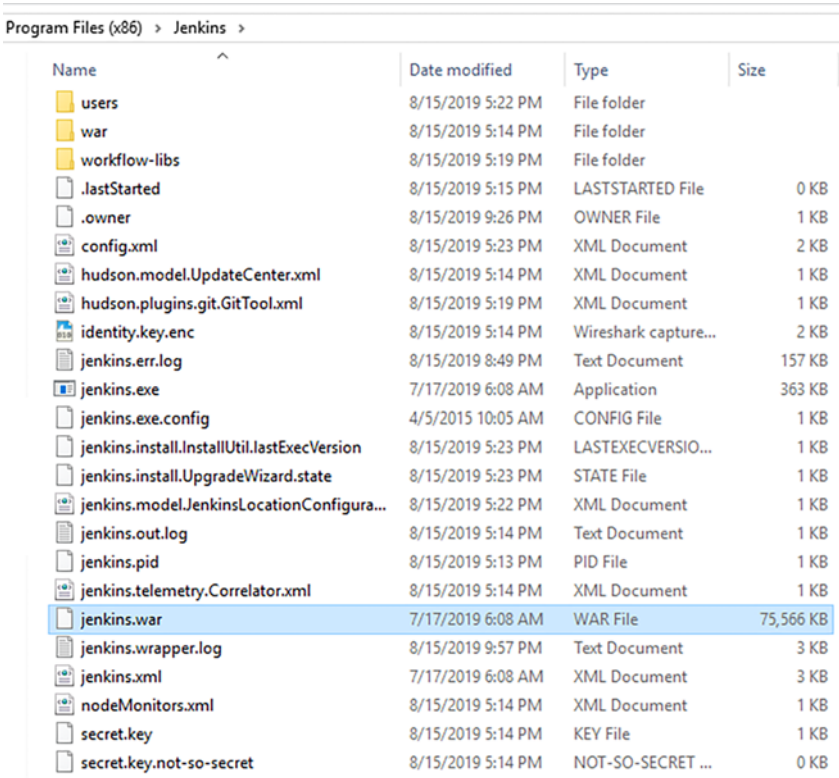


Figure D.4 The Jenkins installation folder with the jenkins.war file

index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

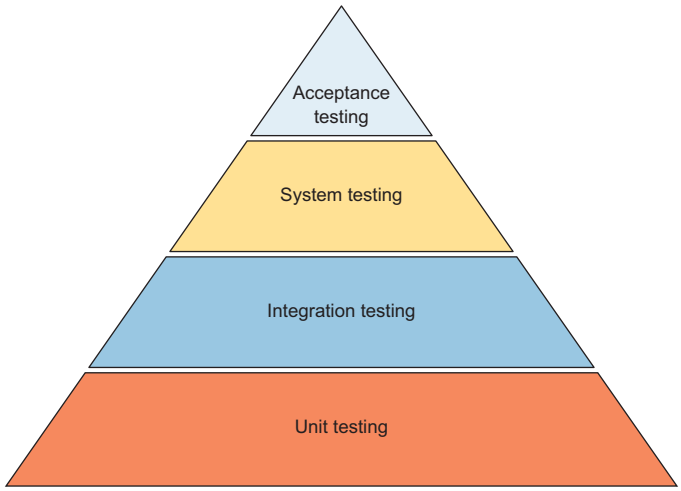
T

U

V

W

X


Acceptance testing
System testing
Integration testing
Unit testing

JAVA/TESTING

JUnit IN ACTION Third Edition

Tudose

The JUnit framework is the gold standard for unit testing Java applications—and knowing it is an essential skill for Java developers. The latest version, JUnit 5, is a total overhaul, now supporting modern Java features like Lambdas and Streams.

JUnit in Action, Third Edition has been completely rewritten for this release. The book is full of examples that demonstrate JUnit's modern features, including its new architecture; nested, tagged, and dynamic tests; and dependency injection. You'll benefit from author Cătălin Tudose's unique "pyramid" testing strategy, which breaks the testing process into layers and sets you on the path to bug-free code creation.

What's Inside

- Migrating from JUnit 4 to 5
- Effective test automation
- Test-driven development and behavior-driven development
- Using mocks for test isolation
- Connecting JUnit 5 with Maven or Gradle

For intermediate Java developers.

Cătălin Tudose has a Ph.D. in Computer Science, and over 15 years of experience as a Senior Java Developer and Technical Team Lead. Previous editions were authored by **Petar Tahchiev**, **Felipe Leme**, **Gary Gregory**, and **Vincent Massol**.

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit www.manning.com/books/junit-in-action-third-edition

"Every tool you need to gain a solid, comprehensive understanding of JUnit."
—Becky Huett, Big Shovel Labs

"A wealth of information about JUnit 5, and about testing!"
—Junilu Lacar
Accenture | SolutionsIQ

"The ultimate reference book you need to migrate your unit tests to JUnit 5!"
—Jean-François Morin
Laval University

"Writing good code takes more than just writing unit tests, and this book shows you the way."
—Burk Hufnagel
Daugherty Business Solutions

ISBN: 978-1-61729-704-5

MANNING $59.99 / Can $79.99 [INCLUDING eBOOK]