

SAMPLE CHAPTER



JAVA TESTING with SPOCK

Konstantinos Kapelonis

FOREWORD BY Luke Daley

 MANNING



Java Testing with Spock

by Konstantinos Kapelonis

Sample Chapter 3

Copyright 2016 Manning Publications

brief contents

PART 1 FOUNDATIONS AND BRIEF TOUR OF SPOCK 1

- 1 ■ Introducing the Spock testing framework 3
- 2 ■ Groovy knowledge for Spock testing 31
- 3 ■ A tour of Spock functionality 62

PART 2 STRUCTURING SPOCK TESTS 89

- 4 ■ Writing unit tests with Spock 91
- 5 ■ Parameterized tests 127
- 6 ■ Mocking and stubbing 157

PART 3 SPOCK IN THE ENTERPRISE 191

- 7 ■ Integration and functional testing with Spock 193
- 8 ■ Spock features for enterprise testing 224

A tour of Spock functionality

This chapter covers

- Understanding the given-when-then Spock syntax
- Testing datasets with data-driven tests
- Introducing mocks/stubs with Spock
- Examining mock behavior

With the Groovy basics out of the way, you're now ready to focus on Spock syntax and see how it combines several aspects of unit testing in a single and cohesive package.

Different applications come with different testing needs, and it's hard to predict what parts of Spock will be more useful to you beforehand. This chapter covers a bit of all major Spock capabilities to give you a bird's-eye view of how Spock works. I won't focus on all the details yet because these are explained in the coming chapters.

The purpose of this chapter is to act as a central hub for the whole book. You can read this chapter and then, according to your needs, decide which of the coming chapters is of special interest to you. If, for example, in your current application you have tests with lots of test data that spans multiple input variables, you can skip straight to the chapter that deals with data-driven tests (chapter 5).

The following sections briefly touch on these three aspects of Spock:

- Core testing of Java code (more details in chapter 4)
- Parameterized tests (more details in chapter 5)
- Isolation of the class under test (more details in chapter 6)

To illustrate these concepts, a series of increasingly complex, semi-real scenarios are used, because some Spock features aren't evident with trivial unit tests. For each scenario, I'll also compare the Spock unit test with a JUnit test (if applicable).

3.1 Introducing the behavior-testing paradigm

Let's start with a full example of software testing. Imagine you work as a developer for a software company that creates programs for fire-control systems, as shown in figure 3.1.

The processing unit is connected to multiple fire sensors and polls them continuously for abnormal readings. When a fire is discovered, the alarm sounds. If the fire starts spreading and another detector is triggered, the fire brigade is automatically called. Here are the complete requirements of the system:

- If all sensors report nothing strange, the system is OK and no action is needed.
- If one sensor is triggered, the alarm sounds (but this might be a false positive because of a careless smoker who couldn't resist a cigarette).
- If more than one sensor is triggered, the fire brigade is called (because the fire has spread to more than one room).

Your colleague has already implemented this system, and you're tasked with unit testing. The skeleton of the Java implementation is shown in listing 3.1.

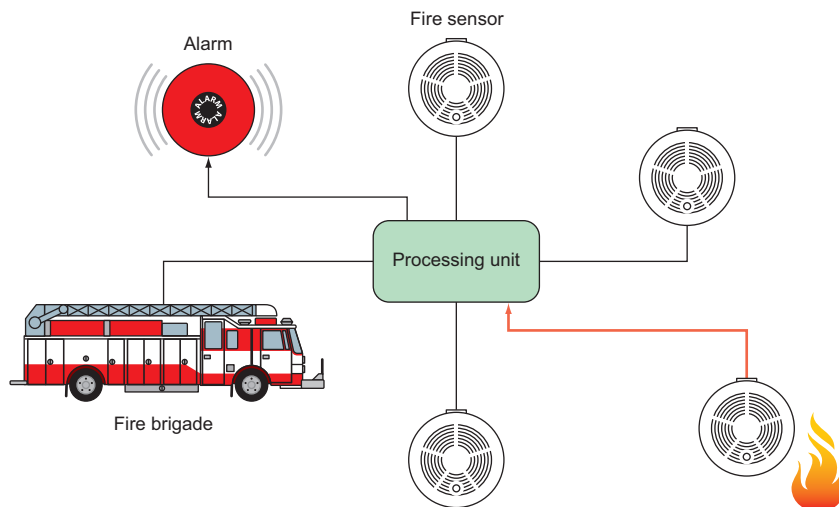


Figure 3.1 A fire-monitoring system controlling multiple detectors

How to use the code listings

You can find almost all code listings for this book at <https://github.com/kkapelon/java-testing-with-spock>.

For brevity, the book sometimes points you to the source code (especially for long listings). I tend to use the Eclipse IDE in my day-to-day work. If you didn't already install Spock and Eclipse in chapter 2, you can find installation instructions in appendix A.

This fire sensor is regularly injected with the data from the fire sensors, and at any given time, the sensor can be queried for the status of the alarm.

Listing 3.1 A fire-control system in Java

```

public class FireEarlyWarning {
    public void feedData(int triggeredFireSensors)
    {
        [...implementation here...]
    }

    public WarningStatus getCurrentStatus()
    {
        [...implementation here...]
    }
}

public class WarningStatus {
    public boolean isAlarmActive() {
        [...implementation here...]
    }

    public boolean isFireDepartmentNotified() {
        [...implementation here...]
    }
}

```

Method called every second by sensor software

The main class that implements monitoring

Redacted for brevity—see source code for full code

Status report getter method

Contents of status report (status class)

If true, the alarm sounds.

If true, the fire brigade is called.

The application uses two classes:

- The *polling class* has all the intelligence and contains a getter that returns a status class with the present condition of the system.
- The *status class* is a simple object that holds the details.¹

¹ This is only the heart of the system. Code for contacting the fire brigade or triggering the alarm is outside the scope of this example.

Your colleague has finished the implementation code, and has even written a JUnit test² as a starting point for the test suite you're supposed to finish. You now have the full requirements of the system and the implementation code, and you're ready to start unit testing.

3.1.1 The setup-stimulate-assert structure of JUnit

You decide to look first at the existing JUnit test your colleague already wrote. The code is shown in the following listing.

Listing 3.2 A JUnit test for the fire-control system

```

@Test
public void fireAlarmScenario() {
    FireEarlyWarning fireEarlyWarning = new FireEarlyWarning();
    int triggeredSensors = 1;

    fireEarlyWarning.feedData(triggeredSensors);
    WarningStatus status = fireEarlyWarning.getCurrentStatus();

    assertTrue("Alarm sounds", status.isAlarmActive());
    assertFalse("No notifications", status.isFireDepartmentNotified());
}

```

Setup needed for the test

JUnit test case

Create an event.

Examine results of the event.

This unit test covers the case of a single sensor detecting fire. According to the requirements, the alarm should sound, but the fire department isn't contacted yet. If you closely examine the code, you'll discover a hidden structure between the lines. All good JUnit tests have three code segments:

- 1 In the *setup phase*, the class under test and all collaborators are created. All initialization stuff goes here.
- 2 In the *stimulus phase*, the class under test is tampered with, triggered, or otherwise passed a message/action. This phase should be as brief as possible.
- 3 The *assert phase* contains only read-only code (code with no side effects), in which the expected behavior of the system is compared with the actual one.

Notice that this structure is *implied* with JUnit. It's never enforced by the framework and might not be clearly visible in complex unit tests. Your colleague is a seasoned developer and has clearly marked the three phases by using the empty lines in listing 3.2:

- The setup phase creates the `FireEarlyWarning` class and sets the number of triggered sensors that will be evaluated (the first two statements in listing 3.2).
- The stimulus phase passes the triggered sensors to the fire monitor and also asks it for the current status (the middle two statements in listing 3.2).
- The assert phase verifies the results of the test (the last two statements).

² Following the test-driven development (TDD) principles of writing a unit test for a feature before the feature implementation.

This is good advice to follow, but not all developers follow this technique. (It's also possible to demarcate the phases with comments.)

Because JUnit doesn't clearly distinguish between the setup-stimulate-assert phases, it's up to the developer to decide on the structure of the unit test. Understanding the structure of a JUnit test isn't always easy when more-complex testing is performed. For comparison, the following listing shows a real-world result.³

Listing 3.3 JUnit test with complex structure (real example)

```
private static final String MASTER_NAME = "mymaster";
private static HostAndPort sentinel = new HostAndPort("localhost", 26379);

@Test
public void sentinelSet() {
    Jedis j = new Jedis(sentinel.getHost(), sentinel.getPort());

    try {
        Map<String, String> parameterMap = new HashMap<String,
            String>();
        parameterMap.put("down-after-milliseconds",
            String.valueOf(1234));
        parameterMap.put("parallel-syncs", String.valueOf(3));
        parameterMap.put("quorum", String.valueOf(2));
        j.sentinelSet(MASTER_NAME, parameterMap);

        List<Map<String, String>> masters = j.sentinelMasters();
        for (Map<String, String> master : masters) {
            if (master.get("name").equals(MASTER_NAME)) {
                assertEquals(1234, Integer.parseInt(master
                    .get("down-after-milliseconds")));
                assertEquals(3,
                    Integer.parseInt(master.get("parallel-
                        syncs")));
                assertEquals(2,
                    Integer.parseInt(master.get("quorum")));
            }
        }

        parameterMap.put("quorum", String.valueOf(1));
        j.sentinelSet(MASTER_NAME, parameterMap);
    } finally {
        j.close();
    }
}
```

After looking at the code, how long did it take you to understand its structure? Can you easily understand which class is under test? Are the boundaries of the three

³ This unit test is from the jedis library found on GitHub. I mean no disrespect to the authors of this code, and I congratulate them for offering their code to the public. The rest of the tests from jedis are well-written.

phases really clear? Imagine that this unit test has failed, and you have to fix it immediately. Can you guess what has gone wrong simply by looking at the code?

Another problem with the lack of clear structure of a JUnit test is that a developer can easily mix the phases in the wrong⁴ order, or even write multiple tests into one. Returning to the fire-control system in listing 3.2, the next listing shows a bad unit test that tests two things at once. The code is shown as an antipattern. Please don't do this in your unit tests!

Listing 3.4 A JUnit test that tests two things—don't do this

```

@Test
public void sensorsAreTriggered() {
    FireEarlyWarning fireEarlyWarning = new FireEarlyWarning(); ← Setup phase
    fireEarlyWarning.feedData(1);
    WarningStatus status = fireEarlyWarning.getCurrentStatus();

    assertTrue("Alarm sounds", status.isAlarmActive()); ← First assert phase
    assertFalse("No notifications", status.isFireDepartmentNotified());
    fireEarlyWarning.feedData(2);

    WarningStatus status2 = fireEarlyWarning.getCurrentStatus();
    assertTrue("Alarm sounds", status2.isAlarmActive()); ← Second assert phase
    assertTrue("Fire Department is notified",
               status2.isFireDepartmentNotified());
}

```

Stimulus phase →

Another stimulus phase—this is bad practice. →

This unit test asserts two different cases. If it breaks and the build server reports the result, you don't know which of the two scenarios has the problem.

Another common antipattern I see all too often is JUnit tests with no assert statements at all! JUnit is powerful, but as you can see, it has its shortcomings. How would Spock handle this fire-control system?

3.1.2 The given-when-then flow of Spock

Unlike JUnit, Spock has a clear test structure that's denoted with labels (*blocks* in Spock terminology), as you'll see in chapter 4, which covers the lifecycle of a Spock test. Looking back at the requirements of the fire-control system, you'll see that they can have a one-to-one mapping with Spock tests. Here are the requirements again:

- If all sensors report nothing strange, the system is OK and no action is needed.
- If one sensor is triggered, the alarm sounds (but this might be a false positive because of a careless smoker who couldn't resist a cigarette).
- If more than one sensor is triggered, the fire brigade is called (because the fire has spread to more than one room).

⁴ Because "everything that can go wrong, will go wrong," you can imagine that I've seen too many antipatterns of JUnit tests that happen because of the lack of a clear structure.

Spock can directly encode these sentences by using full English text inside the source test of the code, as shown in the following listing.

Listing 3.5 The full Spock test for the fire-control system

```

class FireSensorSpec extends spock.lang.Specification{
    def "If all sensors are inactive everything is ok"() {
        given: "that all fire sensors are off"
        FireEarlyWarning fireEarlyWarning = new FireEarlyWarning()
        int triggeredSensors = 0

        when: "we ask the status of fire control"
        fireEarlyWarning.feedData(triggeredSensors)
        WarningStatus status = fireEarlyWarning.getCurrentStatus()

        then: "no alarm/notification should be triggered"
        !status.alarmActive
        !status.fireDepartmentNotified
    }

    def "If one sensor is active the alarm should sound as a precaution"() {
        given: "that only one fire sensor is active"
        FireEarlyWarning fireEarlyWarning = new FireEarlyWarning()
        int triggeredSensors = 1

        when: "we ask the status of fire control"
        fireEarlyWarning.feedData(triggeredSensors)
        WarningStatus status = fireEarlyWarning.getCurrentStatus()

        then: "only the alarm should be triggered"
        status.alarmActive
        !status.fireDepartmentNotified
    }

    def "If more than one sensor is active then we have a fire"() {
        given: "that two fire sensors are active"
        FireEarlyWarning fireEarlyWarning = new FireEarlyWarning()
        int triggeredSensors = 2

        when: "we ask the status of fire control"
        fireEarlyWarning.feedData(triggeredSensors)
        WarningStatus status = fireEarlyWarning.getCurrentStatus()

        then: "alarm is triggered and the fire department is notified"
        status.alarmActive
        status.fireDepartmentNotified
    }
}

```

Setup phase

Clear explanation of what this test does

Stimulus phase

Assert phase

Spock follows a given-when-then structure that's enforced via labels inside the code. Each unit test can be described using plain English sentences, and even the labels can be described with text descriptions.




FireSensorSpec		
	If all sensors are inactive everything is ok	0.008
	If one sensor is active the alarm should sound as a precaution	0.001
	If more than one sensor is active then we have a fire	0.001

Figure 3.2 Surefire report with Spock test description

This enforced structure pushes the developer to think before writing the test, and also acts as a guide on where each statement goes. The beauty of the English descriptions (unlike JUnit comments) is that they're used directly by reporting tools. A screenshot of a Maven Surefire report is shown in figure 3.2 with absolutely no modifications (Spock uses the JUnit runner under the hood). This report can be created by running `mvn surefire-report:report` on the command line.

The first column shows the result of the test (a green tick means that the test passes), the second column contains the description of the test picked up from the source code, and the third column presents the execution time of each test (really small values are ignored). More-specialized tools can drill down in the labels of the blocks as well, as shown in figure 3.3. The example shown is from Spock reports (<https://github.com/renatoathaydes/spock-reports>).

Summary:				
Created on Sun Oct 11 11:40:21 EEST 2015 by Kostis				
Executed features	Failures	Errors	Skipped	Success rate
3	0	0	0	100,0%
Features:				
<ul style="list-style-type: none"> If all sensors are inactive everything is ok If one sensor is active the alarm should sound as a precaution If more than one sensor is active then we have a fire 				
If all sensors are inactive everything is ok				
<i>Given:</i> that all fire sensors are off				
<i>When:</i> we ask the status of fire control				
<i>Then:</i> no alarm/notification should be triggered				
If one sensor is active the alarm should sound as a precaution				
<i>Given:</i> that only one fire sensor is active				
<i>When:</i> we ask the status of fire control				
<i>Then:</i> only the alarm should be triggered				
If more than one sensor is active then we have a fire				
<i>Given:</i> that two fire sensors are active				
<i>When:</i> we ask the status of fire control				
<i>Then:</i> alarm is triggered and the fire department is notified				

Figure 3.3 Spock report with all English sentences of the test

Spock isn't a full BDD tool,⁵ but it certainly pushes you in that direction. With careful planning, your Spock tests can act as living business documentation.

You've now seen how Spock handles basic testing. Let's see a more complex testing scenario, where the number of input and output variables is much larger.

3.2 Handling tests with multiple input sets

With the fire-control system in place, you're tasked with a more complex testing assignment. This time, the application under test is a monitor system for a nuclear reactor. It functions in a similar way to the fire monitor, but with more input sensors. The system⁶ is shown in figure 3.4.

The components of the system are as follows:

- Multiple fire sensors (input)
- Three radiation sensors (input)
- Current pressure (input)
- An alarm (output)

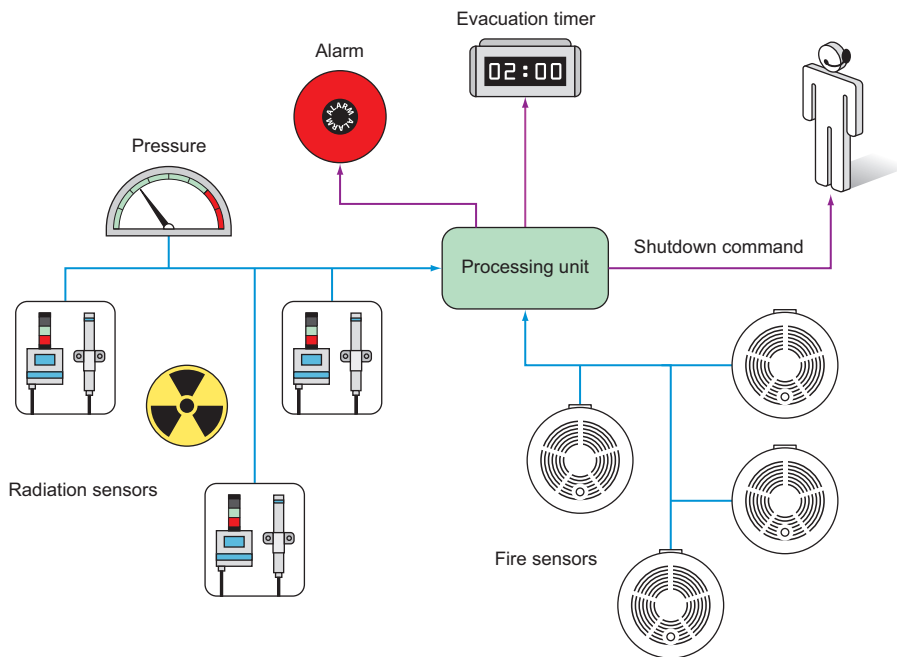


Figure 3.4 A monitor system for a nuclear reactor

⁵ See JBehave (<http://jbehave.org/>) or Cucumber JVM (<http://cukes.info/>) to see how business analysts, testers, and developers can define the test scenarios of an enterprise application.

⁶ This system is imaginary. I'm in no way an expert on nuclear reactors. The benefits of the example will become evident in the mocking/stubbing section of the chapter.

- An evacuation command (output)
- A notification to a human operator that the reactor should shut down (output)

The system is already implemented according to all safety requirements needed for nuclear reactors. It reads sensor data at regular intervals and depending on the readings, it can alert or suggest corrective actions. Here are some of the requirements:

- If pressure goes above 150 bars, the alarm sounds.
- If two or more fire alarms are triggered, the alarm sounds and the operator is notified that a shutdown is needed (as a precautionary measure).
- If a radiation leak is detected (100+ rads from any sensor), the alarm sounds, an announcement is made that the reactor should be evacuated within the next minute, and a notification is sent to the human operator that a shutdown is needed.

You speak with the technical experts of the nuclear reactor, and you jointly decide that a minimum of 12 test scenarios will be examined, as shown in table 3.1.

Table 3.1 Scenarios that need testing for the nuclear reactor

Sample inputs			Expected outputs		
Current pressure	Fire sensors	Radiation sensors	Audible alarm	A shutdown is needed	Evacuation within x minutes
150	0	0, 0, 0	No	No	No
150	1	0, 0, 0	Yes	No	No
150	3	0, 0, 0	Yes	Yes	No
150	0	110.4 ,0.3, 0.0	Yes	Yes	1 minute
150	0	45.3 ,10.3, 47.7	No	No	No
155	0	0, 0, 0	Yes	No	No
170	0	0, 0, 0	Yes	Yes	3 minutes
180	0	110.4 ,0.3, 0.0	Yes	Yes	1 minute
500	0	110.4 ,300, 0.0	Yes	Yes	1 minute
30	0	110.4 ,1000, 0.0	Yes	Yes	1 minute
155	4	0, 0, 0	Yes	Yes	No
170	1	45.3 ,10.f, 47.7	Yes	Yes	3 minutes

The scenarios outlined in this table are a classic example of *parameterized* tests. The test logic is always the same (take these three inputs and expect these three outputs), and the test code needs to handle different sets of variables for only this single test logic.

In this example, we have 12 scenarios with 6 variables, but you can easily imagine cases with much larger test data. The naive way to handle testing for the nuclear reactor

would be to write 12 individual tests. That would be problematic, not only because of code duplication, but also because of future maintenance. If a new variable is added in the system (for example, a new sensor), you'd have to change all 12 tests at once.

A better approach is needed, preferably one that decouples the test code (which should be written once) from the sets of test data and expected output (which should be written for all scenarios). This kind of testing needs a framework with explicit support for parameterized tests.

Spock comes with built-in support for parameterized tests with a friendly DSL⁷ syntax specifically tailored to handle multiple inputs and outputs. But before I show you this expressive DSL, allow me to digress a bit into the current state of parameterized testing as supported in JUnit (and the alternative approaches).

Many developers consider parameterized testing a challenging and complicated process. The truth is that the limitations of JUnit make parameterized testing a challenge, and developers suffer because of inertia and their resistance to changing their testing framework.

3.2.1 Existing approaches to multiple test-input parameters

The requirements for the nuclear-reactor monitor are clear, the software is already implemented, and you're ready to test it. What's the solution if you follow the status quo?

The recent versions of JUnit advertise support for parameterized tests. The official way of implementing a parameterized test with JUnit is shown in the following listing. The listing assumes that -1 in evacuation minutes means that no evacuation is needed.

Listing 3.6 Testing the nuclear reactor scenarios with JUnit

```
@RunWith(Parameterized.class)
public class NuclearReactorTest {
    private final int triggeredFireSensors;
    private final List<Float> radiationDataReadings;
    private final int pressure;

    private final boolean expectedAlarmStatus;
    private final boolean expectedShutdownCommand;
    private final int expectedMinutesToEvacuate;

    public NuclearReactorTest(int pressure, int triggeredFireSensors,
        List<Float> radiationDataReadings, boolean expectedAlarmStatus,
        boolean expectedShutdownCommand, int expectedMinutesToEvacuate) {
        this.triggeredFireSensors = triggeredFireSensors;
        this.radiationDataReadings = radiationDataReadings;
        this.pressure = pressure;
        this.expectedAlarmStatus = expectedAlarmStatus;
        this.expectedShutdownCommand = expectedShutdownCommand;
        this.expectedMinutesToEvacuate = expectedMinutesToEvacuate;
    }
}
```

Specialized runner needed for parameterized tests is created with `@RunWith`.

Outputs become class fields.

Inputs become class fields.

Special constructor with all inputs and outputs

⁷ A DSL is a programming language targeted at a specific problem as opposed to a general programming language like Java. See http://en.wikipedia.org/wiki/Domain-specific_language.

```

    }

    @Test
    public void nuclearReactorScenario() {
        NuclearReactorMonitor nuclearReactorMonitor = new
            NuclearReactorMonitor();

        nuclearReactorMonitor.feedFireSensorData(triggeredFireSensors);
        nuclearReactorMonitor.feedRadiationSensorData(radiationDataReadings);
        nuclearReactorMonitor.feedPressureInBar(pressure);
        NuclearReactorStatus status = nuclearReactorMonitor.getCurrentStatus();

        assertEquals("Expected no alarm", expectedAlarmStatus,
            status.isAlarmActive());
        assertEquals("No notifications", expectedShutdownCommand,
            status.isShutdownNeeded());
        assertEquals("No notifications", expectedMinutesToEvacuate,
            status.getEvacuationMinutes());
    }

    @Parameters
    public static Collection<Object[]> data() {
        return Arrays.asList(new Object[][] {
            { 150, 0, new ArrayList<Float>(), false, false, -1 },
            { 150, 1, new ArrayList<Float>(), true, false, -1 },
            { 150, 3, new ArrayList<Float>(), true, true, -1 },
            { 150, 0, Arrays.asList(110.4f, 0.3f, 0.0f), true,
                true, 1 },
            { 150, 0, Arrays.asList(45.3f, 10.3f, 47.7f), false,
                false, -1 },
            { 155, 0, Arrays.asList(0.0f, 0.0f, 0.0f), true, false,
                -1 },
            { 170, 0, Arrays.asList(0.0f, 0.0f, 0.0f), true, true,
                3 },
            { 180, 0, Arrays.asList(110.4f, 0.3f, 0.0f), true,
                true, 1 },
            { 500, 0, Arrays.asList(110.4f, 300f, 0.0f), true,
                true, 1 },
            { 30, 0, Arrays.asList(110.4f, 1000f, 0.0f), true,
                true, 1 },
            { 155, 4, Arrays.asList(0.0f, 0.0f, 0.0f), true, true,
                -1 },
            { 170, 1, Arrays.asList(45.3f, 10.3f, 47.7f), true,
                true, 3 }, });
    }
}

```

Unit test that will use parameters

Source of test data

Two-dimensional array with test data

If you look at this code and feel it's too verbose, you're right! This listing is a true testament to the limitations of JUnit. To accomplish parameterized testing, the following constraints specific to JUnit need to be satisfied:

- The test class must be polluted with fields that represent inputs.
- The test class must be polluted with fields that represent outputs.

- A special constructor is needed for all inputs and outputs.
- Test data comes into a two-dimensional object array (which is converted to a list).

Notice also that because of these limitations, it's impossible to add a second parameterized test in the same class. JUnit is so strict that it forces you to have a single class for each test when multiple parameters are involved. If you have a Java class that needs more than one parameterized test and you use JUnit, you're out of luck.⁸

The problems with JUnit parameterized tests are so well known that several independent efforts have emerged to improve this aspect of unit testing. At the time of writing, at least three external projects⁹ offer their own syntax on top of JUnit for a friendlier and less cluttered code.

Parameterized tests are also an area where TestNG (<http://testng.org>) has been advertised as a better replacement for JUnit. TestNG does away with all JUnit limitations and comes with extra annotations (`@DataProvider`) that truly decouple test data and test logic.

Despite these external efforts, Spock comes with an even better syntax for parameters (Groovy magic again!). In addition, having all these improved efforts external to JUnit further supports my argument that Spock is a “batteries-included” framework providing everything you need for testing in a single package.

3.2.2 *Tabular data input with Spock*


You've seen the hideous code of JUnit when multiple parameters are involved. You might have also seen some improvements with TestNG or extra JUnit add-ons. All these solutions attempt to capture the values of the parameters by using Java code or annotations.

Spock takes a step back and focuses directly on the original test scenarios. Returning to the nuclear-monitoring system, remember that what you want to test are the scenarios listed in table 3.1 (written in a human-readable format).

Spock allows you to do the unthinkable. You can directly embed this table as-is inside your Groovy code, as shown in the next listing. Again I assume that -1 in evacuation minutes means that no evacuation is needed.

Listing 3.7 Testing the nuclear reactor scenarios with Spock

```
class NuclearReactorSpec extends spock.lang.Specification{
    def "Complete test of all nuclear scenarios"() {
        given: "a nuclear reactor and sensor data"
        NuclearReactorMonitor nuclearReactorMonitor =new
            NuclearReactorMonitor()
```



⁸ There are ways to overcome this limitation, but I consider them hacks that make the situation even more complicated.

⁹ <https://code.google.com/p/fuzztester/wiki/FuzzTester>; <https://github.com/Pragmatists/junitparams>; <https://github.com/piotrtrurski/zohhak>.


```

when: "we examine the sensor data"
nuclearReactorMonitor.feedFireSensorData (fireSensors)
nuclearReactorMonitor.feedRadiationSensorData (radiation)
nuclearReactorMonitor.feedPressureInBar (pressure)
NuclearReactorStatus status = nuclearReactorMonitor.getCurrentStatus()

then: "we act according to safety requirements"
status.alarmActive == alarm
status.shutDownNeeded == shutDown
status.evacuationMinutes == evacuation

where: "possible nuclear incidents are:"
pressure | fireSensors | radiation | alarm | shutDown | evacuation
150      | 0              | []      | false | false   | -1
150      | 1              | []      | true  | false   | -1
150      | 3              | []      | true  | true    | -1
150      | 0 | [110.4f, 0.3f, 0.0f] | true  | true    | 1
150      | 0 | [45.3f, 10.3f, 47.7f] | false | false   | -1
155      | 0 | [0.0f, 0.0f, 0.0f]   | true  | false   | -1
170      | 0 | [0.0f, 0.0f, 0.0f]   | true  | true    | 3
180      | 0 | [110.4f, 0.3f, 0.0f] | true  | true    | 1
500      | 0 | [110.4f, 300f, 0.0f] | true  | true    | 1
30       | 0 | [110.4f, 1000f, 0.0f] | true  | true    | 1
155      | 4 | [0.0f, 0.0f, 0.0f]   | true  | true    | -1
170      | 1 | [45.3f, 10.3f, 47.7f] | true  | true    | 3
}
}

```

Usage of test inputs

Usage of test outputs

Definition of inputs and outputs

Source of parameters

Tabular representation of all scenarios

Spock takes a different approach to parameters. Powered by Groovy capabilities, it offers a descriptive DSL for tabular data. The key point of this unit test is the **where:** label (in addition to the usual given-then-when labels) that holds a definition of all inputs/outputs used in the other blocks.

In the **where:** block of this Spock test, I copied verbatim the scenarios of the nuclear-reactor monitor from the table. The `||` notation is used to split the inputs from outputs. Reading this table is possible even by nontechnical people. Your business analyst can look at this table and quickly locate missing scenarios.

Adding a new scenario is easy:

- You can append a new line at the end of the table with a new scenario, and the test will pick the new scenario upon the next run.
- The parameters are strictly contained inside the test method, unlike JUnit. The test class has no need for special constructors or fields. A single Spock class can hold an unlimited number of parameterized tests, each with its own tabular data.

The icing on the cake is the amount of code. The JUnit test has 82 lines of Java code, whereas the Spock test has 38 lines. In this example, I gained 50% code reduction by using Spock, and kept the same functionality as before (keeping my promise from chapter 1 that Spock tests will reduce the amount of test code in your application).

Chapter 5 shows several other tricks for Spock parameterized tests, so feel free to jump there directly if your enterprise application is plagued by similar JUnit boilerplate code.

We'll close our Spock tour with its mocking/stubbing capabilities.

3.3 *Isolating the class under test*

JUnit doesn't support mocking (faking external object communication) out of the box. Therefore, I usually employ Mockito¹⁰ when I need to fake objects in my JUnit tests.

If you've never used mocking in your unit tests, fear not, because this book covers both theory and practice (with Spock). I strongly believe that mocking is one of the pillars of well-written unit tests and I'm always puzzled when I see developers who neglect or loathe mocks and stubs.

The literature on mocking hasn't reached a single agreement on naming the core concepts. Multiple terms exist, such as these:

- Mocks/stubs
- Test doubles
- Fake collaborators

All these usually mean the same thing: dummy objects that are injected in the class under test, replacing the real implementations.

- A *stub* is a fake class that comes with preprogrammed return values. It's injected in the class under test so that you have absolute control of what's being tested as input.
- A *mock* is a fake¹¹ class that can be examined after the test is finished for its interactions with the class under test (for example, you can ask it whether a method was called or how many times it was called).

Things sometimes get more complicated because a mock can also function as a stub if that's needed.¹² The rest of this book uses the mock/stub naming convention because Spock closely follows this pattern. The next examples show both.

3.3.1 *The case of mocking/stubbing*

After finishing with the nuclear-reactor monitor module, you're tasked with testing the temperature sensors of the same reactor. Figure 3.5 gives an overview of the system.

¹⁰ Many mock frameworks are available for Java, but Mockito is the easiest and most logical in my opinion. Some of its ideas have also found their way into Spock itself. See <https://github.com/mockito/mockito>.

¹¹ Don't sweat the naming rules. In my day job, I name all these classes as mocks and get on with my life.

¹² The two hardest problems in computer science are naming things and cache invalidation.

Even though at first glance this temperature monitor is similar to the previous system, it has two big differences:

- The system under test—the temperature monitor—doesn't directly communicate with the temperature sensors. It obtains the readings from another Java system, the temperature reader (implemented by a different software company than yours).
- The requirements for the temperature monitor indicate that the alarm should sound if the difference in temperature readings (either up or down) is greater than 20 degrees.

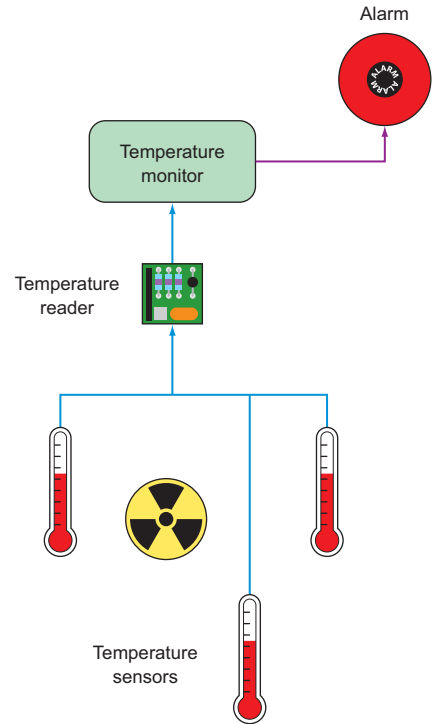


Figure 3.5 A monitor that gets temperatures via another system

You need to write unit tests for the temperature monitor. The implementation code to be tested is shown in the next listing.

Listing 3.8 Java classes for the temperature monitor and reader

```

public class TemperatureReadings {
    private long sensor1Data;
    private long sensor2Data;
    private long sensor3Data;

    [...getters and setters here]
}

public interface TemperatureReader {
    TemperatureReadings getCurrentReadings();
}

public class TemperatureMonitor {
    private final TemperatureReader reader;
    private TemperatureReadings lastReadings;
    private TemperatureReadings currentReadings;
}
    
```

Annotations for Listing 3.8:

- Simple class that contains temperatures (points to TemperatureReadings)
- Current temperature (points to currentReadings field)
- Interface implemented by the reader software (points to TemperatureReader)
- The class under test (points to TemperatureMonitor)
- Injected field of reader (points to reader field)
- Latest temperature readings (points to lastReadings field)
- Method called by the class under test (points to getCurrentReadings() method)
- Previous temperature readings (points to lastReadings field)

```

    public TemperatureMonitor(final TemperatureReader reader)
    {
        this.reader = reader;
    }

    public boolean isTemperatureNormal()
    {
        [...implementation here that compares readings...]
    }

    public void readSensor()
    {
        lastReadings = currentReadings;
        currentReadings = reader.getCurrentReadings();
    }
}

```

Constructor injection → (points to the constructor)

← **Method that needs unit tests** (points to `isTemperatureNormal()`)

← **Called automatically at regular intervals** (points to `readSensor()`)

Communication with temperature reader | (points to the `readSensor()` method body)

The specifications are based on temperature readings. Unlike the previous example that used fixed values (for example, if pressure is more than 150, do this), here you have to test consecutive readings (that is, take an action only if temperature is higher compared to the previous reading).

Reading the specifications, it's obvious you need a way to “trick” the class under test to read temperature readings of your choosing. Unfortunately, the temperature monitor has no way of directly obtaining input. Instead, it calls another Java API from the reader software.¹³ How can you “trick” the `TemperatureMonitor` class to read different types of temperatures?

SOLUTIONS FOR FAKING INPUT FROM COLLABORATING CLASSES

A good start would be to contact the software company that writes the temperature-reader software and ask for a debug version of the module, which can be controlled to give any temperature you choose, instead of reading the real hardware sensors. This scenario might sound ideal, but in practice it's difficult to achieve, either for political reasons (the company won't provide what you ask) or technical reasons (the debug version has bugs of its own).

Another approach would be to write your own dummy implementation of `TemperatureReader` that does what you want. I've seen this technique too many times in enterprise projects, and I consider it an antipattern. This introduces a new class that's used exclusively for unit tests and must be kept in sync with the specifications. As soon as the specifications change (which happens a lot in enterprise projects), you must hunt down all those dummy classes and upgrade them accordingly to keep the stability of unit tests.

The recommended approach is to use the built-in mocking capabilities of Spock. Spock allows you to create a replacement class (or interface implementation) on the

¹³ Notice that in this case I used constructor injection, but setter injection could also work.

spot and direct it to do your bidding while the class under test still thinks it's talking to a real object.

3.3.2 Stubbing fake objects with Spock

To create a unit test for the temperature-monitoring system, you can do the following:

- 1 Create an implementation of the `TemperatureReader` interface.
- 2 Instruct this smart implementation to return fictional readings for the first call.
- 3 Instruct this smart implementation to return other fictional readings for the second call.
- 4 Connect the class under test with this smart implementation.
- 5 Run the test, and see what the class under test does.

In Spock parlance, this “smart implementation” is called a *stub*, which means a fake class with canned responses. The following listing shows stubbing in action, as previously outlined.

Listing 3.9 Stubbing with Spock

```
class CoolantSensorSpec extends spock.lang.Specification{

    def "If current temperature difference is within limits everything is
        ok"() {
        given: "that temperature readings are within limits"
        TemperatureReadings prev = new
            TemperatureReadings(sensor1Data:20,
                sensor2Data:40,sensor3Data:80)
        TemperatureReadings current = new
            TemperatureReadings(sensor1Data:30,
                sensor2Data:45,sensor3Data:73)
        TemperatureReader reader = Stub(TemperatureReader)

        reader.getCurrentReadings() >>> [prev, current]

        TemperatureMonitor monitor = new TemperatureMonitor(reader)

        when: "we ask the status of temperature control"
        monitor.readSensor()
        monitor.readSensor()

        then: "everything should be ok"
        monitor.isTemperatureNormal()

    }

    def "If current temperature difference is more than 20 degrees the
        alarm should sound"() {
        given: "that temperature readings are not within limits"
        TemperatureReadings prev = new
            TemperatureReadings(sensor1Data:20,
                sensor2Data:40,sensor3Data:80)
        TemperatureReadings current = new
```

Dummy interface implementation →

Class under test is injected with dummy interface →

Class under test calls dummy interface |

← **Premade temperature readings**

← **Instructing the dummy interface to return premade readings**

← **Assertion after two subsequent calls**

```

        TemperatureReadings (sensor1Data:30,
            sensor2Data:10, sensor3Data:73) ;
        TemperatureReader reader = Stub(TemperatureReader)
        reader.getCurrentReadings() >>> [prev, current]
        TemperatureMonitor monitor = new TemperatureMonitor(reader)

        when: "we ask the status of temperature control"
            monitor.readSensor()
            monitor.readSensor()

        then: "the alarm should sound"
            !monitor.isTemperatureNormal()
    }
}

```

The magic line is the `Stub()` call, shown here:

```
TemperatureReader reader = Stub(TemperatureReader)
```

Spock, behind the scenes, creates a dummy implementation of this interface. By default the implementation does nothing, so it must be instructed how to react, which is done with the second important line, the `>>>` operator:

```
reader.getCurrentReadings() >>> [prev, current]
```

This line indicates the following:

- The first time the `getCurrentReadings()` method is called on the dummy interface, return the instance named `prev`.
- The second time, return the object named `current`.

The `>>>` operator is normally called an *unsigned shift operator*¹⁴ in Java, but Spock overloads it (Groovy supports operator overloading) to provide canned answers to a stub. Now the dummy interface is complete. The class under test is injected with the Spock stub, and calls it without understanding that all its responses are preprogrammed. As far as the class under test is concerned, the Spock stub is a real implementation.

The final result: you've implemented the unit test for the temperature reader complying with the given requirements, even though the class under test never communicates with the temperature sensors themselves.

3.3.3 Mocking collaborators

For simplicity, all the systems in these examples so far only recommend the suggested action (for example, the alarm should sound). They assume that another external system polls the various monitors presented and then takes the action.

In the real world, systems are rarely this simple. Faking the input data is only half the effort needed to write effective unit tests. The other half is faking the output

¹⁴ <http://docs.oracle.com/javase/tutorial/java/nutsandbolts/op3.html>.

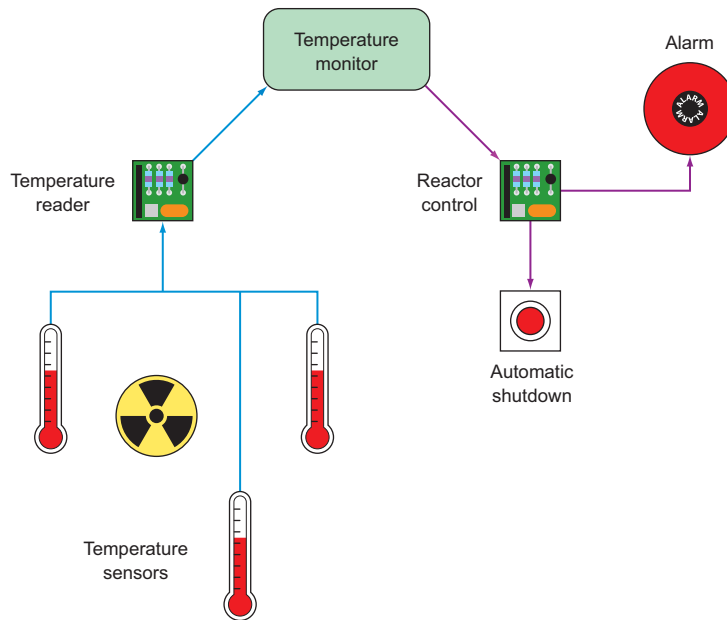


Figure 3.6 A full system with input and output and side effects

parameters. In this case, you need to use mocking in the mix as well. To see how this works, look at the extended temperature-monitor system shown in figure 3.6.

Assume that for this scenario, business analysis has decided that the temperature control of the reactor is mission critical and must be completely automatic. Instead of sounding an alarm and contacting a human operator, the system under test is fully autonomous, and will shut down the reactor on its own if the temperature difference is higher than 50 degrees. The alarm still sounds if the temperature difference is higher than 20 degrees, but the reactor doesn't shut down in this case, allowing for corrective actions by other systems.

Shutting down the reactor and sounding the alarm happens via an external Java library (over which you have no control) that's offered as a simple API. The system under test is now injected with this external API as well, as shown in the following listing.

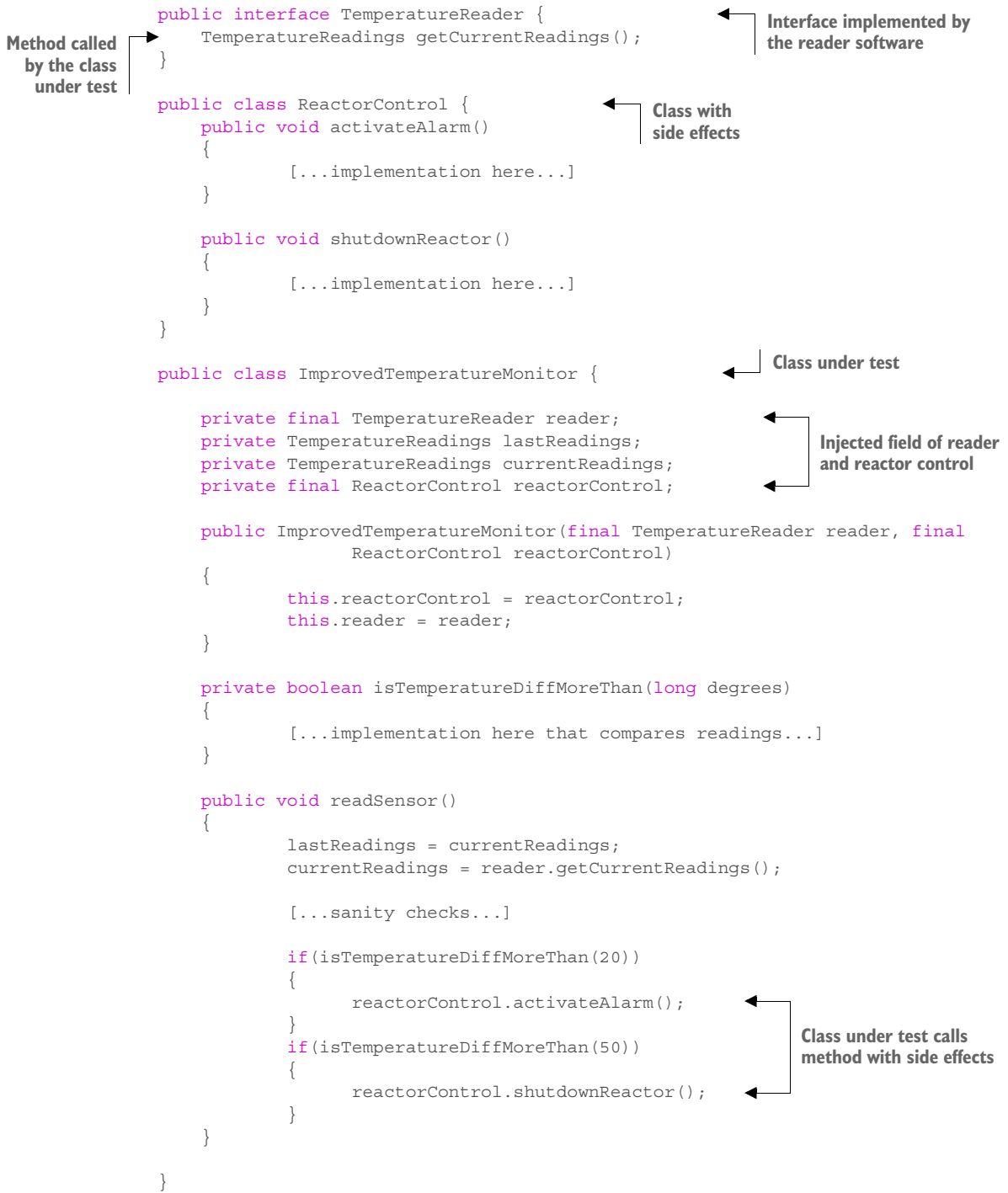
Listing 3.10 Java classes for the temperature monitor, reader, and reactor control

```
public class TemperatureReadings {
    private long sensor1Data;
    private long sensor2Data;
    private long sensor3Data;

    [...getters and setters here]
}
```

Current
temperature

← Simple class that
contains temperatures



Again, you're tasked with the unit tests for this system. By using Spock stubs as demonstrated in the previous section, you already know how to handle the temperature reader. This time, however, you can't easily verify the reaction of the class under test, `ImprovedTemperatureMonitor`, because there's nothing you can assert.

The class doesn't have any method that returns its status. Instead it internally calls the Java API for the external library that handles the reactor. How can you test this?

OPTIONS FOR UNIT TESTING THIS MORE-COMPLEX SYSTEM

As before, you have three options:

- 1 You can ask the company that produces the Java API of the reactor control to provide a "debug" version that doesn't shut down the reactor, but instead prints a warning or a log statement.
- 2 You can create your own implementation of `ReactorControl` and use that to create your unit test. This is the same antipattern as stubs, because it adds extra complexity and an unneeded maintenance burden to sync this fake object whenever the Java API of the external library changes. Also notice that `ReactorControl` is a concrete class and not an interface, so additional refactoring effort is required before you even consider this route.
- 3 You can use mocks. This is the recommended approach.

Let's see how Spock handles this testing scenario.

3.3.4 Examining interactions of mocked objects

As it does for stubbing, Spock also offers built-in mocking support. A *mock* is another fake collaborator of the class under test. Spock allows you to examine mock objects for their interactions after the test is finished. You pass it as a dependency, and the class under test calls its methods without understanding that you intercept all those calls behind the scenes. As far as the class under test is concerned, it still communicates with a real class.

Unlike stubs, mocks can fake input/output, and can be examined after the test is complete. When the class under test calls your mock, the test framework (Spock in this case) notes the characteristics of this call (such as number of times it was called or even the arguments that were passed for this call). You can examine these characteristics and decide if they are what you expect.

In the temperature-monitor scenario, you saw how the temperature reader is stubbed. The reactor control is also mocked, as shown in the next listing.

Listing 3.11 Mocking and stubbing with Spock

```
def "If current temperature difference is more than 20 degrees the alarm
    sounds"() {
    given: "that temperature readings are not within limits"
    TemperatureReadings prev = new TemperatureReadings(sensor1Data:20,
        sensor2Data:40,sensor3Data:80)
    TemperatureReadings current = new TemperatureReadings(sensor1Data:30,
        sensor2Data:10,sensor3Data:73);
```

```

TemperatureReader reader = Stub(TemperatureReader)

reader.getCurrentReadings() >>> [prev, current]

ReactorControl control = Mock(ReactorControl)
ImprovedTemperatureMonitor monitor = new
    ImprovedTemperatureMonitor(reader, control)

when: "we ask the status of temperature control"
monitor.readSensor()
monitor.readSensor()

then: "the alarm should sound"
0 * control.shutdownReactor()
1 * control.activateAlarm()
}

def "If current temperature difference is more than 50 degrees the reactor
    shuts down"() {
    given: "that temperature readings are not within limits"
    TemperatureReadings prev = new TemperatureReadings(sensor1Data:20,
        sensor2Data:40,sensor3Data:80)
    TemperatureReadings current = new TemperatureReadings(sensor1Data:30,
        sensor2Data:10,sensor3Data:160);
    TemperatureReader reader = Stub(TemperatureReader)

    reader.getCurrentReadings() >>> [prev, current]

    ReactorControl control = Mock(ReactorControl)
    ImprovedTemperatureMonitor monitor = new
        ImprovedTemperatureMonitor(reader, control)

    when: "we ask the status of temperature control"
    monitor.readSensor()
    monitor.readSensor()

    then: "the alarm should sound and the reactor should shut down"
    1 * control.shutdownReactor()
    1 * control.activateAlarm()
}

```

← Creating a stub for an interface

← Creating a mock for a concrete class

Class under test is injected with mock and stub.

Mock methods are called behind the scenes.

Verification of mock calls

The code is similar to listing 3.9, but this time the class under test is injected with two fake objects (a stub and a mock). The mock line is as follows:

```
ReactorControl control = Mock(ReactorControl)
```

Spock automatically creates a dummy class that has the exact signature of the `ReactorControl` class. All methods by default do nothing (so there's no need to do anything special if that's enough for your test).

You let the class under test run its way, and at the end of the test, instead of testing Spock assertions, you examine the interactions of the mock you created:

```
0 * control.shutdownReactor()
1 * control.activateAlarm()
```

- The first line says, “After this test is finished, I expect that the number of times the `shutdownReactor()` method was called is zero.”
- The second line says, “After this test is finished, I expect that the number of times the `activateAlarm()` method was called is one.”

This is equivalent to the business requirements that dictate what would happen depending on different temperature variations.

Using both mocks and stubs, you’ve seen how it’s possible to write a full test for the temperature system without shutting down the reactor each time your unit test runs. The reactor scenario might be extreme, but in your programming career, you may already have seen Java modules with side effects that are difficult or impossible to test without the use of mocking. Common examples are as follows:

- Charging a credit card
- Sending a bill to a client via email
- Printing a report
- Booking a flight with an external system

Any Java API that has severe side effects is a natural candidate for mocking. I’ve only scratched the surface of what’s possible with Spock mocks. In chapter 6, you’ll see many more advanced examples that also demonstrate how to capture the arguments of mocked calls and use them for further assertions, or even how a stub can respond differently according to the argument passed.

Mocking with Mockito

For comparison, I’ve included in the GitHub source code the same test with JUnit/Mockito in case you want to compare it with listing 3.11 and draw your own conclusions. Mockito was one of the inspirations for Spock, and you might find some similarities in the syntax. Mockito is a great mocking framework, and much thought has been spent on its API. It sometimes has a strange syntax in more-complex examples (because it’s still limited by Java conventions). Ultimately, however, it’s Java’s verbosity that determines the expressiveness of a unit test, regardless of Mockito’s capabilities.

For example, if you need to create a lot of mocks that return Java maps, you have to create them manually and add their elements one by one before instructing Mockito to use them. Within Spock tests, you can create maps in single statements (even in the same line that stubbing happens), as you’ve seen in Chapter 2.

Also, if you need a parameterized test with mocks (as I’ll show in the next section), you have to combine at least three libraries (JUnit plus Mockito plus JUnitParams) to achieve the required result.

3.3.5 Combining mocks and stubs in parameterized tests

As a grand finale of this Spock tour, I'll show you how to easily combine parameterized tests with mocking/stubbing in Spock. I'll again use the temperature scenario introduced in listing 3.10. Remember the requirements of this system:

- If the temperature difference is larger than 20 degrees (higher or lower), the alarm sounds.
- If the temperature difference is larger than 50 degrees (higher or lower), the alarm sounds and the reactor shuts down automatically.

We have four cases as far as temperature is concerned, and three temperature sensors. Therefore, a full coverage of all cases requires at least 12 unit tests. Spock can combine parameterized tests with mocks/stubs, as shown in the following listing.

Listing 3.12 Mocking/stubbing in a Spock parameterized test

```
def "Testing of all 3 sensors with temperatures that rise and fall"() {
  given: "various temperature readings"
  TemperatureReadings prev =
    new TemperatureReadings(sensor1Data:previousTemp[0],
      sensor2Data:previousTemp[1], sensor3Data:previousTemp[2])
  TemperatureReadings current =
    new TemperatureReadings(sensor1Data:currentTemp[0],
      sensor2Data:currentTemp[1], sensor3Data:currentTemp[2]);

  TemperatureReader reader = Stub(TemperatureReader)
  reader.getCurrentReadings() >>> [prev, current]

  ReactorControl control = Mock(ReactorControl)
  ImprovedTemperatureMonitor monitor = new
    ImprovedTemperatureMonitor(reader,control)

  when: "we ask the status of temperature control"
  monitor.readSensor()
  monitor.readSensor()

  then: "the alarm should sound and the reactor should shut down if
        needed"
  shutDown * control.shutdownReactor()
  alarm * control.activateAlarm()

  where: "possible temperatures are:"
  previousTemp | currentTemp | | alarm | shutDown
  [20, 30, 40] | [25, 15, 43.2] | | 0 | 0
  [20, 30, 40] | [13.3, 37.8, 39.2] | | 0 | 0
  [20, 30, 40] | [50, 15, 43.2] | | 1 | 0
  [20, 30, 40] | [-20, 15, 43.2] | | 1 | 0
  [20, 30, 40] | [100, 15, 43.2] | | 1 | 1
  [20, 30, 40] | [-80, 15, 43.2] | | 1 | 1
  [20, 30, 40] | [20, 55, 43.2] | | 1 | 0
  [20, 30, 40] | [20, 8, 43.2] | | 1 | 0
  [20, 30, 40] | [21, 100, 43.2] | | 1 | 1
}
```

Input temperature with parameters →

Creation of dummy interface →

Class under test is injected with mock and stub →

Instrumenting return value of interface ←

Mocking of concrete class ←

Class under test calls stub and mock behind the scenes ←

Verification of mock using parameters ←

All parameter variations and expected results ←

```

[20, 30, 40] | [22, -40, 43.2]    ||    1 | 1
[20, 30, 40] | [20, 35, 76]     ||    1 | 0
[20, 30, 40] | [20, 31, 13.2]   ||    1 | 0
[20, 30, 40] | [21, 33, 97]     ||    1 | 1
[20, 30, 40] | [22, 39, -22]    ||    1 | 1
}

```

This code combines everything you've learned in this chapter. It showcases the following:

- The expressiveness of Spock tests (clear separation of test phases)
- The easy tabular format of parameters (matching business requirements)
- The ability to fake both input and output of the class under test

As an exercise, try replicating this functionality using Java and JUnit in fewer lines of code (statements). As I promised you at the beginning of the book, Spock is a cohesive testing framework that contains everything you need for your unit tests, all wrapped in friendly and concise Groovy syntax.

3.4 Summary

- Spock tests have a clear structure with explicit given-when-then blocks.
- Spock tests can be named with full English sentences.
- JUnit reporting tools are compatible with Spock tests.
- Spock tests allow for parameterized tests with the `where:` block.
- Parameters in Spock tests can be written directly in a tabular format (complete with header).
- Unlike JUnit, Spock can have an unlimited number of parameterized tests in the same class.
- A stub is a fake class that can be programmed with custom behavior.
- A mock is a fake class that can be examined (after the test is finished) for its interactions with the class under test (which methods were called, what the arguments were, and so forth).
- Spock can stub classes/interfaces and instrument them to return whatever you want.
- The triple-right-shift/unsigned shift (`>>>`) operator allows a stub to return different results each time it's called.
- Spock can mock classes/interfaces and automatically record all invocations.
- Spock can verify the number of times a method of a mock was called.
- Combining stubs, mocks, and multiple parameters in the same Spock test is easy.

JAVA TESTING WITH SPOCK

Konstantinos Kapelonis

Spock combines the features of tools like JUnit, Mockito, and JBehave into a single powerful Java testing library. With Spock, you use Groovy to write more readable and concise tests. Spock enables seamless integration testing, and with the intuitive Geb library, you can even handle functional testing of web applications.

Java Testing with Spock teaches you how to use Spock for a wide range of testing use cases in Java. You'll start with a quick overview of Spock and work through writing unit tests using the Groovy language. You'll discover best practices for test design as you learn to write mocks, implement integration tests, use Spock's built-in BDD testing tools, and do functional web testing using Geb. Readers new to Groovy will appreciate the succinct language tutorial in chapter 2 that gives you just enough Groovy to use Spock effectively.

What's Inside

- Testing with Spock from the ground up
- Write mocks without an external library
- BDD tests your business analyst can read
- Just enough Groovy to use Spock

Written for Java developers. Knowledge of Groovy and JUnit is helpful but not required.

Konstantinos Kapelonis is a software engineer who works with Java daily.

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/books/java-testing-with-spock

“Goes beyond mere exploration of Spock's API and feature set to include general testing practices and real-world application.”

—From the Foreword by
Luke Daley
Spock founding contributor

“An awesome guide to one of the most useful test frameworks for Java.”

—Christopher W. H. Davis, Nike

“Discover the power of Spock and Groovy, step-by-step.”

—David Pardo, Amaron

“Does an excellent job of exploring features of Spock that are seldom, if ever, mentioned in other online resources. If you care about producing quality tests, then this book is for you!”

—Annyce Davis
The Washington Post



ISBN 13: 978-1-61729-253-8
ISBN 10: 1-61729-253-2

