JAVA TESTING
with SPOCK

Konstantinos Kapelonis
FOREWORD BY Luke Daley

MANNING

Java Testing with Spock

by Konstantinos Kapelonis

Sample Chapter 1

brief contents

Part 1

Foundations and brief tour of Spock

Spock is a test framework that uses the Groovy programming language. The first part of the book expands on this by making sure that we (you, the reader, and me, the author) are on the same page.

To make sure that we are on the same page in the most gradual way, I first define a testing framework (and why it's needed) and introduce a subset of the Groovy syntax needed for writing Spock unit tests. I know that you're eager to see Spock tests (and write your own), but some features of Spock will impress you only if you've first learned a bit about the goals of a test framework and the shortcomings of current test frameworks (for example, JUnit).

Don't think, however, that this part of the book is theory only. Even at this early stage, this brief tour of Spock highlights includes full code listings and some out-of-the-ordinary examples.

Chapter 1 is a bird's-eye view of Spock, explaining its position in the Java ecosystem, the roles it plays in the testing process, and a brief comparison with JUnit. Feel free to skip this chapter if you're a seasoned Java developer and have already written a lot of JUnit tests.

Chapter 2 is a crash course in the Groovy programming language for Java developers. I promised that I don't assume any Groovy knowledge on your part, and this chapter keeps that promise. In it, I specifically focus only on Groovy features that are useful to Spock tests. By the end of this chapter, you'll be fully primed for reading and writing the Spock Groovy syntax. If you're interested in learning the whole Groovy package (for writing production code and not just

unit tests), you can think of this chapter as a stepping stone to full Groovy nirvana. If you already know your way around Groovy code (and are familiar with closures and expandos), you can safely skip this chapter.

Chapter 3 demonstrates the three major facets of Spock (core testing, parameterized tests, and mocking/stubbing). These are presented via a series of testing scenarios for which the Java production code is already available and you're tasked with the unit tests. All the examples present that same functionality in both Spock and JUnit/ Mockito so that you can draw your own conclusions on the readability and clarity of the test code. Chapter 3 acts as a hub for the rest of the book, as you can see which facet of Spock interests you for your own application.

Let's start your Spock journey together!

Introducing the Spock testing framework

This chapter covers

- Introducing Spock
- Bird's-eye view of the testing process
- Using Groovy to test Java
- Understanding Spock's place in the testing world

We live in the computer revolution. We've reached the point where computers are so commonplace that most of us carry a pocket-sized one all the time: a mobile phone. Mobile phones can now perform real-time face recognition, something that used to require a mainframe or computer cluster. At the same time, access to cheap and "always-on" internet services has created a communication layer that surrounds us.

As we enjoy the benefits of computerized services in our daily lives, our expectations are also changing. We expect information to be always available. Errors and unexpected behavior in a favorite software service leave us frustrated. E-commerce is on the rise, and all brands fight for customer loyalty as we turn to the internet for our shopping needs. Once I ordered a single chair from a well-known furniture

company, and my credit card was charged three times the amount shown on the product page because of a computer error. Naturally, I never bought anything from that online shop again.

These high expectations of error-free software create even more pressure on developers if the "user" of the software is an organization, another company, or even a government agency. Software errors can result in loss of time/money/brand loyalty and, more important, loss of trust in the software.

If you're a software developer at any level, you know that writing programming code is only half of software creation. Testing the programming code is also essential in order to verify its correctness. Software problems (more commonly known as *bugs*) have a detrimental effect on the reliability of an application. A continuous goal of software development is the detection of bugs before the software is shipped or deployed to production.

A bug/issue that reaches production code can have a profound effect, depending on the type of software. For example, if your software is a mobile application for tracking daily intake of calories, you can sleep easily each night knowing that any issues found by users will only inconvenience them, and in the worst case they'll delete your application from their mobile phones (if they get really angry about the problems). But if, for example, you're writing software that manages hotel reservations, consequences are more serious. Critical issues will result in customer anger, brand damage for the hotel, and probable future financial losses.

On the extreme end of the spectrum, consider the severity of consequences for issues with the following:

- Software that controls hospital equipment
- Software that runs on a nuclear reactor
- Software that tracks enemy ballistic missiles and retaliates with its own defensive missiles (my favorite example)

How will you sleep at night if you're not sure these applications are thoroughly tested before reaching production status?

1.1   *What is Spock?*

This book is about Spock, a comprehensive testing framework for Java (and Groovy) code that can help you automate the boring, repetitive, and manual process of testing a software application. Spock is comprehensive because it's a union of existing Java testing libraries, as shown in figure 1.1.
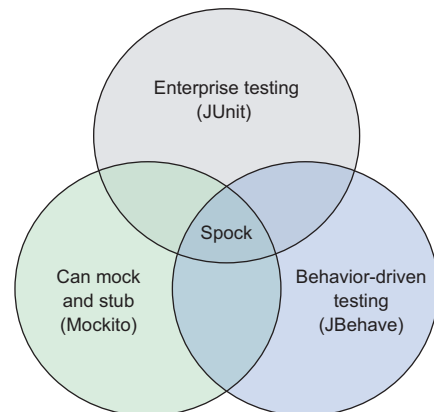
Figure 1.1   Spock among existing Java testing tools

As the figure shows, Spock is a superset of the de facto testing framework for Java: JUnit (http://junit.org/). Spock also comes with built-in capabilities for features that normally require additional libraries. At its core, Spock is a testing framework capable of handling the full lifecycle of a software application.

Spock was initially created in 2008 by Peter Niederwieser, a software engineer with Gradleware.[1] Inspired by existing test tools such as jMock (www.jmock.org) and RSpec (http://rspec.info/), Spock is used by several libraries within the open-source community, including Apache Tapestry (https://github.com/apache/tapestry-5) and MongoDB (https://github.com/mongodb/mongo-java-driver), and by several commercial companies (for instance, Netflix). A second Spock committer is Luke Daley (also with Gradleware), creator of the popular Geb functional testing framework (www.gebish.org) demonstrated in chapter 7. Spock, a new entry in the test framework arena, challenges the undisputed king—JUnit—armed with a bunch of fresh ideas against the legacy techniques of the past. Spock tests are written in Groovy, but they can test either Groovy or Java code.

1.1.1 Mocking and stubbing

The most basic unit tests (called *logic tests* by some) are those that focus on the logic of a single Java class. To test a single class in a controlled environment and isolate it from the other classes it depends on (*collaborators*), Spock comes with built-in support for "faking" external object communication. This capability, known as *mocking and stubbing*, isn't inside vanilla JUnit; you need external libraries—for example, Mockito (https://github.com/mockito/mockito) or jMock (http://www.jmock.org/)—to achieve this isolation of a Java class.

1.1.2 Behavior-driven development

Spock also embraces the paradigm of *behavior-driven development* (BDD), a development process that attempts to unify implementation, testing, and business people inside a software organization, by introducing a central way of documenting requirements and validating software functionality against those requirements, in a clear and repeatable manner. Spock combines these facets into a single convenient package offering a holistic approach to software testing.

1.1.3 Spock's design features

Spock has the following characteristics.

ENTERPRISE-READY
Spock can be easily integrated with the popular build systems, Maven (https://maven.apache.org/) and Gradle (https://gradle.org/). Spock runs as part of a build

---

[1] The same company behind Gradle, a build system in Groovy (a replacement for Maven).

process and produces reports of automated test runs. Spock can be used to test back-end code, web pages, HTTP services, and more.

COMPREHENSIVE

Spock is a one-stop shop when it comes to testing. It has built-in capabilities for mocking and stubbing (creating fake objects), allowing you to decide on the breadth of the testing context. Spock can test a single class, a code module, or a whole application context with ease. You can perform end-to-end testing with Spock (covered in chapter 7) or isolate one class/method for your testing needs without any external libraries (described in chapter 6).

FAMILIAR/COMPATIBLE

Spock runs on top of the JUnit runner, which already enjoys mature support among tools and development environments. You run your Spock tests in the same way as your JUnit tests. You can even mix the two in the same project and get reports on test failures or code coverage in a similar way to JUnit. Run your tests in parallel or in a serial way; Spock doesn't care because it's fully compatible with existing JUnit tools.

INSPIRED

Spock is relatively new and doesn't carry any legacy burden. It's designed from scratch but at the same time it takes the best features of existing testing libraries (and tries to avoid their disadvantages). For example, Spock embraces the given-when-then structure of JBehave (http://jbehave.org/) but also discards the cumbersome record/replay code of older mocking frameworks.

1.1.4 *Spock's coding features*

Spock's coding features are as follows.

CONCISE

Spock uses the Groovy syntax, which is already concise and mixes its simplified syntax on top. No more tests that hide the substance with boilerplate code!

READABLE

Spock follows a close-to-English flow of statements that can be readable even by non-technical people (for example, business analysts). Collaboration among analysis, development, and testing people can be greatly simplified with Spock tests. If you always wanted to name your test methods by using full English sentences, now you can!

METICULOUS

When things go wrong, Spock gives as much detail as possible on the inner workings of the code at the time of failure. In some cases, this is more than enough for a developer to understand the problem without resorting to the time-consuming debugging process.

EXTENSIBLE

Spock allows you to write your own extensions to cater to your specific needs. Several of its "core" features are extensions (or started as extensions).

Listing 1.1 provides a sample test in Spock that illustrates several of these key coding features. The example shows a billing system that emails invoices to customers only if they have provided an email address.

> ### How to use the code listings
>
> You can find almost all code listings of 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 Java listings). I use the Eclipse integrated development environment (IDE) in my day-to-day work, as shown in the screenshots throughout the book. You can find specific instructions for installing Spock and using it via Maven, Gradle, Eclipse, and IntelliJ in appendix A.

Don't be alarmed by unknown keywords at this point. Even if you know absolutely no Groovy at all, you should be able to understand the scenario in question by the presence of full English sentences. The following chapters explain all details of the syntax. All of chapter 2 is devoted to Groovy and how it differs from Java.

Listing 1.1   Sample Spock test

```
class InvoiceMailingSpec extends spock.lang.Specification{          Full English
                                                                    sentences describe
    def "electronic invoices to active email addresses"() {        what the test does.
        given: "an invoice, a customer, a mail server and a printer"
        PrinterService printerService = Mock(PrinterService)        Integrated mocking of
        EmailService emailService = Mock(EmailService)              collaborator classes
        Customer customer = new Customer()
        FinalInvoiceStep finalInvoiceStep = new
                FinalInvoiceStep(printerService, emailService)
        Invoice invoice = new Invoice()

        when: "customer is normal and has an email inbox"           Given-when-then
        customer.hasEmail("acme@example.com")                       declarative style
        finalInvoiceStep.handleInvoice(invoice, customer)           of BDD

        then: "invoice should not be printed. Only an
                        email should be sent"
        0 * printerService.printInvoice(invoice)
        1 * emailService.sendInvoice(invoice,"acme@example.com")
    }
}
```

The Spock specification can be executed by a JUnit runner.

Verifying interactions of mocked objects

As you can see, the Spock test has a clear given-when-then flow denoted with labels (the BDD style of tests), and each label comes fully documented with an English sentence. Apart from the `def` keyword and the `*` symbol in the last two statements, almost all code is Java-like. Note that the `spock.lang.Specification` class is runnable by JUnit, meaning that this class can act as a JUnit test as far as build tools are concerned. Upcoming chapters cover these and several other features of Spock.

Testing is a highly controversial subject among software developers, and often the discussion focuses on testing tools and the number of tests that are needed in an application. Heated discussions always arise on what needs to be tested in a large application and whether tests help with deadlines. Some developers (hopefully, a minority) even think that all tests are a waste of time, or that their code doesn't need unit tests. If you think that testing is hard, or you believe that you don't have enough time to write tests, this book will show you that Spock uses a concise and self-documenting syntax for writing test cases.

If, on the other hand, you've already embraced sound testing practices in your development process, I'll show you how the Spock paradigm compares to established tools such as JUnit and TestNG (http://testng.org/).
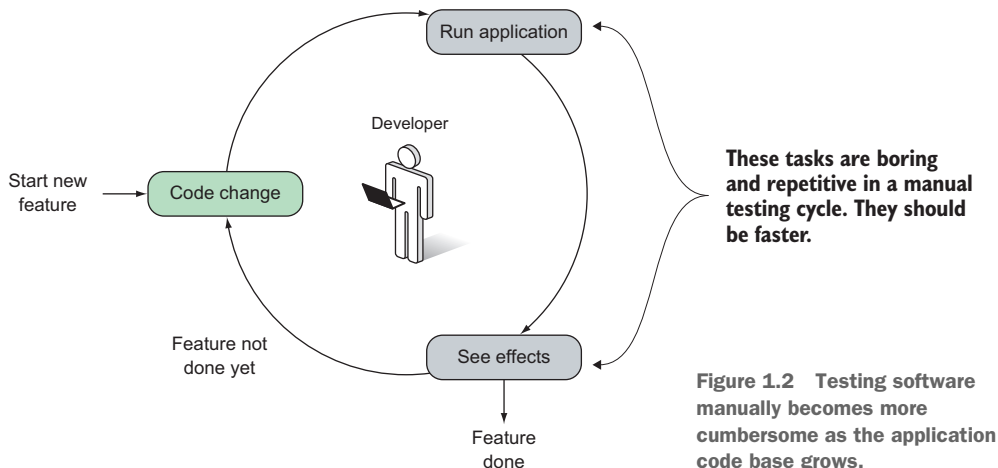
Before getting into the details of using Spock, let's explore why you need a test framework in the first place. After all, you already test your code manually as part of every coding session, when you make sure that what you coded does what you intended.

1.2   *The need for a testing framework*

The first level of testing comes from you. When you implement a new feature, you make a small code change and then run the application to see whether the required functionality is ready. Compiling and running your code is a daily task that happens many times a day as you progress toward the required functionality.

Some features, such as "add a button here that sorts this table of the report," are trivial enough that they can be implemented and tested in one run. But more-complex features, such as "we need to change the policy of approving/rejecting a loan," will need several changes and runs of the application until the feature is marked as complete.

You can see this manual code-run-verify cycle in figure 1.2.

Run application

Developer

Start new
feature   Code change

**These tasks are boring
and repetitive in a manual
testing cycle. They should
be faster.**

Feature not
done yet   See effects

Feature
done

Figure 1.2   Testing software
manually becomes more
cumbersome as the application
code base grows.

Manual testing is enough for small software projects. A quick prototype, a side project, or a weekend coding session can be tested manually by a single person. In order for the cycle to work effectively, a single loop must be quick enough for the developer to see the results of the code change. In an ideal case, a single code change should be verified in seconds. If running the whole application and reaching the point where the new feature is found requires several minutes, developer productivity suffers.

Writing software is a creative process that requires getting into the "zone." Having constant interruptions with lengthy intervals between each code change is a guaranteed way to disrupt the developer's thinking about the code structure (not to mention loss of time/money while waiting for the test to finish).

As the programming code grows past a certain point, this manual cycle gets lengthier, with more time spent running and testing the application than writing code. Soon the run-verify time dominates the "developing" time. Another problem is the time it takes to redeploy software with the new changes. Small software projects can be deployed in seconds, but larger code bases (think bank software) may need several minutes for a complete deployment, further slowing the manual testing cycle.

1.2.1 *Spock as an enterprise-ready test framework*

Spock is marketed as an enterprise-ready test framework, so it's best to explain the need for automated testing in the context of enterprise software—software designed to solve the problems of a large business enterprise. Let's look at an example that reveals why a test framework is essential for large enterprise applications.

Imagine you've been hired as a software developer for a multinational company that sells sports equipment in an online shop. Most processes of the company depend on a monolithic system that handles all daily operations.

You're one of several developers responsible for this central application that has all the characteristics of typical enterprise in-house software:

- The code base is large (more than 200,000 lines of code).
- The development team is 5–20 people.
- No developer knows all code parts of the application.
- The application has already run in production for several years.
- New features are constantly requested by project stakeholders.
- Some code has been written by developers who have left the software department.

The last point is the one that bothers you most. Several areas of the application have nonexistent documentation, and no one to ask for advice.

DEALING WITH NEW REQUIREMENTS IN AN ENTERPRISE APPLICATION

You're told by your boss that because the snow season is approaching, all ski-related materials will get a 25% discount for a limited time period that must also be configurable. The time period might be a day, a week, or any other arbitrary time period.

Your approach is as follows:

1 Implement the feature.
2 Check the functionality by logging manually into the e-shop and verifying that the ski products have the additional discount during checkout.
3 Change the date of the system to simulate a day after the offer has ended.
4 Log in to the e-shop again and verify that the discount no longer applies.

You might be happy with your implementation and send the code change to the production environment, thinking you've covered all possible cases, as shown in figure 1.3.
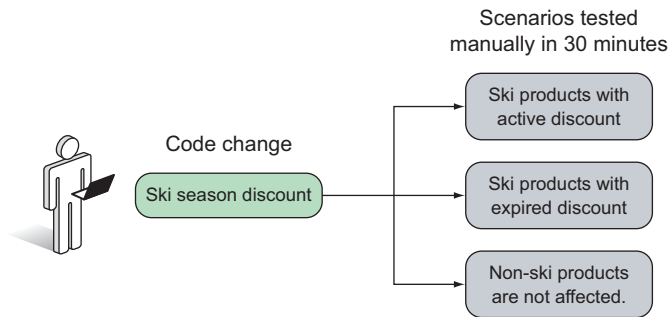
Figure 1.3   Scenarios tested after a simple code change

UNDERSTANDING ENTERPRISE COMPLEXITY: OF MODULES AND MEN

The next morning, your boss frantically tells you to revert the change because the company is losing money! He explains that the e-shop has several VIP customers who always get a 10% percent discount on all products. This VIP discount should never be applied with other existing discounts. Because you didn't know that, VIPs are now getting a total discount of 35%, far below the profit margin of the company. You revert the change and note that for any subsequent change, you have to remember to test for VIP customers as well.

This is a direct result of a large code base with several modules affecting more than one user-visible feature, or a single user-visible feature being affected by more than one code module. In a large enterprise project, some modules affect all user-visible features (typical examples are core modules for security and persistence). This asymmetric relationship is illustrated in figure 1.4.
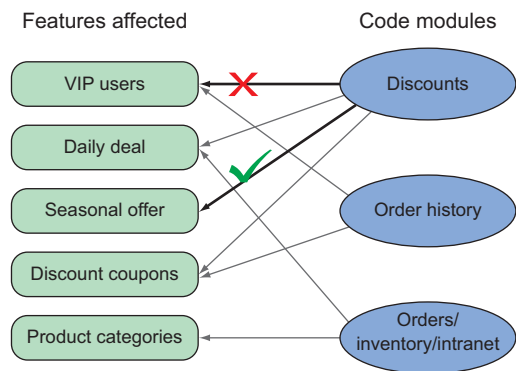
Figure 1.4   A single change in one place has an unwanted effect in another place.

With the change reverted, you learn more about the business requirements of discounts. The final discount of a product is affected by the following:

- Types of customers (first time, normal, silver, VIP)
- Three coupon code types (personal, seasonal, special)
- Ad hoc limited-time offers
- Standard seasonal discounts
- Time of products in the warehouse
- 30+ categories of sports equipment of the company

The next time you tamper with the discount code module, you'll have to manually test more than 100 cases of all the possible combinations. Testing all of them manually would require at least four hours of boring, repetitive work, as shown in figure 1.5.

This enterprise example should make it clear that the complexity of software makes the manual testing cycle slow. Adding a new feature becomes a time-consuming process because each code change must be examined for side effects in all other cases.

Another issue similar to module interaction is the human factor: in a big application, communication between domain experts, developers, testers, system administrators, and so on isn't always free of misunderstandings and conflicting requirements. Extensive documentation, clear communication channels, and an open policy regarding information availability can mitigate the problems but can't completely eliminate them.
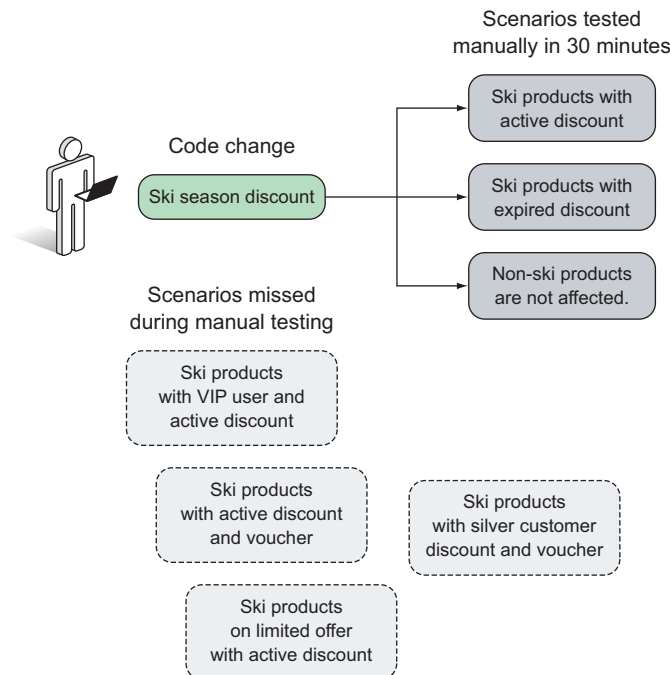
Figure 1.5   Some scenarios were missed by manual testing.
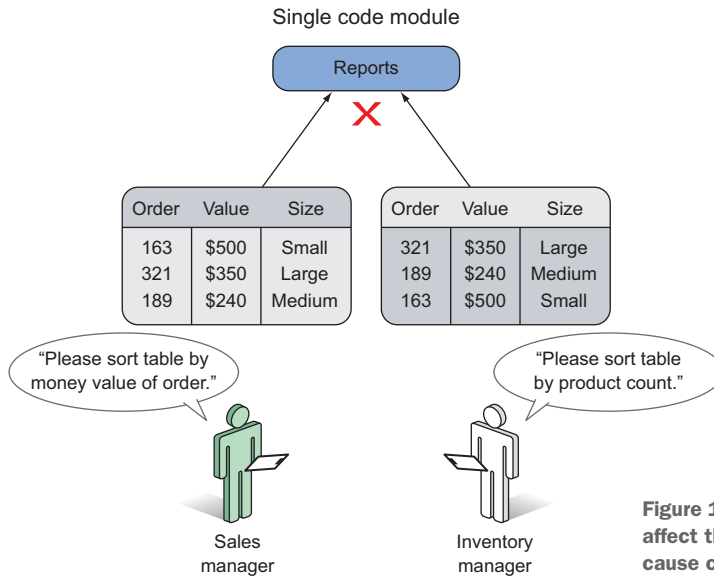
Figure 1.6   Similar features that affect the same code module can cause conflicts.

As an example, a sales manager in the e-shop decides that he wants to see all tables in the back-office application sorted by the value of the order, while at the same time an inventory manager wants to sort the same tables by order size. Two separate developers could be tasked with these cases without knowing that the requirements are conflicting, as shown in figure 1.6.

This enterprise example illustrates firsthand the problems of every large software code base:

- Manually testing every possible combination of data input after a code change is difficult and even impossible in some cases.
- It's hard to predict which parts of the application will be affected by a single code change. Developers are afraid to change existing code, fearing they might break existing functionality.
- Code changes for a new feature can enable previous bugs that have already been fixed to resurface (regressions).
- Understanding all system requirements from the existing code isn't easy. Reading the code provides information only on what happens and not on why it happens.
- Redeploying the application to see the effects of a code change could be a lengthy process on its own and could slow development time even further.

Now you know the major problems faced by a software development team working on a big enterprise project. Next, let's look at various approaches to tackling these problems.

1.2.2 *Common ways to handle enterprise complexity*

All software companies suffer from these problems and deal with them in one of the following three ways or their variations (I've seen them all in real life):

- Developers manually test everything after each code change.
- Big code changes are avoided for fear of unforeseen bugs.
- A layered testing approach is introduced that includes automated testing.

Let's look at each of these solutions in turn.

PERFORMING MINDLESS MANUAL TESTING

In the first case (which is possible with only small- to middle-sized software projects), developers aren't entirely sure what's broken after a code change. Therefore, they manually test all parts of the application after they implement a new feature or fix an issue. This approach wastes a lot of time/money because developers suffer from the repetitive nature of testing (which is a natural candidate for automation).

In addition, as the project grows, testing everything by hand becomes much more difficult. Either the development progress comes to a crawl, as most developers deal with testing instead of adding new features, or (the most common case) developers add features and test only parts of the application that they think might be affected. The result is that bugs enter production code and developers become firefighters; each passing day is a big crisis as the customer discovers missing functionality.

AVOIDING BIG CODE CHANGES

In the second case, the "solution" is to never perform big code changes at all. This paradigm is often embraced by large organizations with big chunks of legacy code (for example, banks). Management realizes that new code changes may introduce bugs that are unacceptable. On the other hand, manual testing of the code is next to impossible because of the depth and breadth of all user scenarios (for example, you can't possibly test all systems of a bank in a logical time frame by hand).

The whole code base is declared sacred. Changing or rewriting code is strictly forbidden by upper management. Developers are allowed to add only small features to the existing infrastructure, without touching the existing code. Local gurus inspect each code change extensively before it enters production status. Code reuse isn't possible. A lot of code duplication is present, because each new feature can't modify existing code. Either you already have what you need to implement your feature, or you're out of luck and need to implement it from scratch.

If you're a developer working in situations that belong to these first two cases (manual testing and the big code base that nobody touches), I feel for you! I've been there myself.

DELEGATING TO AN AUTOMATED TESTING FRAMEWORK

There's a third approach, and that's the one you should strive for. In the third case, an automated test framework is in place that runs after every code change. The framework is tireless, meticulous, and precise. It runs in the background (or on demand) and checks several user features whenever a change takes place. In a well-managed
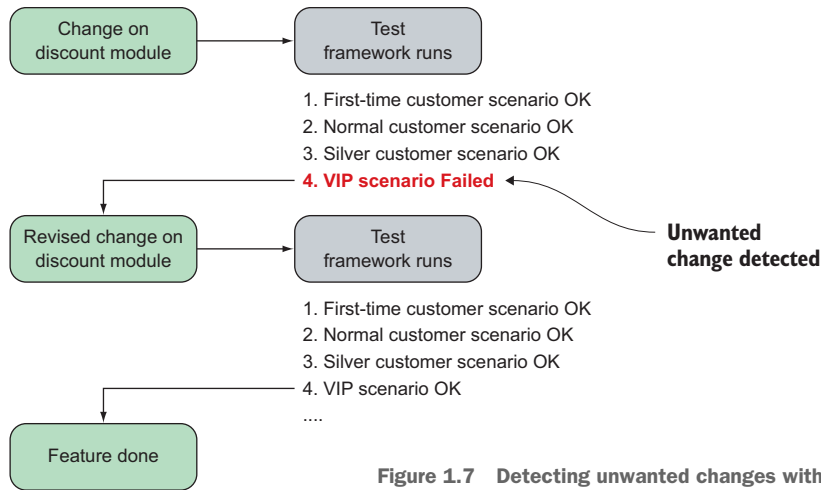
Figure 1.7   Detecting unwanted changes with a test framework

software creation process, the testing framework runs automatically after every developer commit as part of a build pipeline (for example, with the Jenkins build server, available for free at http://jenkins-ci.org/). Results from this automatic run can influence further steps. A common policy is that code modules with failed test results should never be deployed to a production environment.

The test framework acts as an early warning system against unwanted code effects. To illustrate the previous example, if you had a test framework in place, you'd get an automated report after any change, as shown in figure 1.7.

A test framework has the following characteristics.
It reduces

- Feedback time needed to verify the effects of code changes
- Boring, repetitive tasks

It ensures

- Confidence when a new feature is implemented, a bug is fixed, or code is refactored
- The detection of conflicting requirements

It provides

- Documentation for code and an explanation of the reasons behind the current state

Code can be refactored, removed, and updated with ease, because the test framework continuously reports unwanted side effects. Developers are free to devote most of their time to coding new features and fixing existing (known) bugs. Features quickly come into production code, and the customer receives a software package known to be stable and solid for all scenarios supported by the test framework. An initial time investment is required for the testing framework, but after it's in place, the gains

outperform the time it takes to write the test scripts. Catching code regressions and severe bugs before they enter the production environment is much cheaper than allowing them to reach the final users.

A test framework also has other benefits not instantly visible with regard to code quality. The process of making programming code testable enforces several constraints on encapsulation and extensibility that can be easily neglected if the code isn't created with tests in mind. Techniques for making your code testable are covered in chapter 8. But the most important benefit of a test framework is the high developer confidence when performing a deep code change.

Let's dig into how Spock, as a testing framework specializing in enterprise applications, can help you refactor code with such confidence.

1.3 Spock: the groovier testing framework

When I first came upon Spock, I thought that it would be the JUnit alternative to the Groovy programming language. After all, once a programming language reaches a critical mass, somebody ports the standard testing model, known as xUnit (https://en.wikipedia.org/wiki/XUnit), to the respective runtime environment. xUnit frameworks already exist for all popular programming languages.

But Spock is not the xUnit of Groovy! It resembles higher-level testing frameworks, such as RSpec and Cucumber (https://github.com/cucumber/cucumber-jvm), that follow the concepts of BDD, instead of the basic setup-stimulate-assert style of xUnit. BDD attempts (among other things) to create a one-to-one mapping between business requirements and unit tests.

1.3.1 Asserts vs. Assertions

If you're familiar with JUnit, one of the first things you'll notice with Spock is the complete lack of assert statements. *Asserts* are used in unit tests in order to verify the test. You define the expected result, and JUnit automatically fails the test if the expected output doesn't match the actual one.

Assert statements are still there if you need them, but the preferred way is to use Spock *assertions* instead, a feature so powerful that it has been backported to Groovy itself. You'll learn more in chapter 2 about Power asserts and how they can help you pinpoint the causes of a failing test.

1.3.2 Agnostic testing of Java and Groovy

Another unique advantage of Spock is the ability to agnostically test both Java and Groovy code, as shown in figure 1.8.

Figure 1.8   Spock can test both Java and Groovy code.
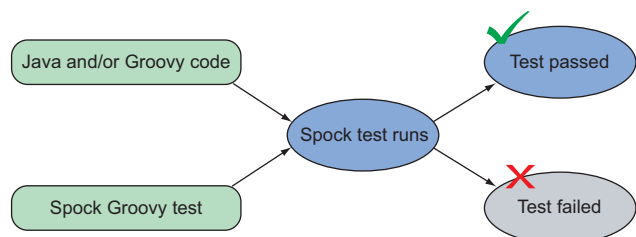
Groovy is a dynamic language that runs in the same Java Virtual Machine (JVM) as Java. Java supporters are proud of the JVM, and some believe that the value of the JVM as a runtime environment is even higher than Java the language. Spock is one example of the power the JVM has to accommodate code from different programming languages.

Spock can test any class that runs on the JVM, regardless of the original source code (Java or Groovy). It's possible with Spock to test either a Java class or a Groovy class in the exact same way. Spock doesn't care about the origin of the class, as long as it's JVM-compatible. You can even verify both Java and Groovy code in the same Spock test if your project is a mix of the two.

1.3.3   *Taking advantage of Groovy tricks in Spock tests*

Finally, you need to know that Groovy is a dynamic language that behaves differently than Java in some important aspects (such as the declaration of variables), as you'll learn in chapter 2. This means that several "tricks" you learn with Spock are in reality a mix of both Groovy and Spock magic, because Spock can extend Groovy syntax in ways that would be difficult with Java (if not impossible). And yes, unlike Java, in Groovy a library/framework can change the syntax of the code as well. Spock is one such library, as you'll learn in chapter 4.

As you become more familiar with Spock and Groovy, the magic behind the curtain will start to appear, and you might even be tempted to use Groovy outside Spock tests as well!

> **AST transformations: changing the structure of the Groovy language**
>
> Several tricks of Groovy magic come from the powerful meta-programming facilities offered during runtime that can change classes and methods in ways impossible with vanilla Java. At the same time, Groovy also supports compile-time macros (*abstract syntax tree*, or AST, transformations in Groovy parlance). If you're familiar with macros in other programming languages, you should be aware of the power they bring to code transformations. By using AST transformations, a programmer can add/change several syntactic features of Groovy code, modifying the syntax in forms that were difficult or impossible in Java.
>
> Spock takes advantage of these compile and runtime code-transformation features offered by Groovy in order to create a pseudo-DSL (domain specific language) specifically for unit tests. All the gory details of Spock syntax are explained in chapter 4.

1.4   *Getting an overview of Spock's main features*

Before starting with the details of Spock code, let's take a bird's-eye view of its major features and how they implement the good qualities of a testing framework, as already explained.

1.4.1 *Enterprise testing*

A test framework geared toward a big enterprise application has certain requirements in order to handle the complexity and possible configurations that come with enterprise software. Such a test framework must easily adapt to the existing ecosystem of build tools, coverage metrics, quality dashboards, and other automation facilities.

Rather than reinventing the wheel, Spock bases its tests on the existing JUnit runner. The runner is responsible for executing JUnit tests and presenting their results to the console or other tools (for example, the IDE). Spock reuses the JUnit runner to get for free all the mature support of external tools already created by Junit:

- Do you want to see code coverage reports with Spock?
- Do you want to run your tests in parallel?
- Do you want to divide your tests into long running and short running?

The answer to all these questions is "Yes, you do, as you did before with JUnit." More details about these topics are presented in chapter 7.

1.4.2 *Data-driven tests*

A common target for unit tests is to handle input data for the system in development. It's impossible to know all potential uses for your application in advance, let alone the ways people are going to use and misuse your application.

Usually a number of unit tests are dedicated to possible inputs of the system in a gradual way. The test starts with a known set of allowed or disallowed input, and as bugs are encountered, the test is enriched with more cases. Common examples include a test that checks whether a username is valid or which date formats are accepted in a web service.

These tests suffer from a lot of code duplication if code is handled carelessly. The test is always the same (for example, Is the username valid?), and only the input changes. Whereas JUnit has some facilities for this type of test (parameterized test), Spock takes a different turn, and offers a special DSL that allows you to embed data tables in Groovy source code. Data-driven tests are covered in chapter 5.

1.4.3 *Mocking and stubbing*

For all its strengths, object-oriented software suffers from an important flaw. The fact that two objects work correctly individually doesn't imply that both objects will also work correctly when connected to each other. The reverse is also true: side effects from an object chain may hide or mask problems that happen in an individual class.

A direct result of these facts is that testing software usually needs to cover two levels at once: the integration level, where tests examine the system as a whole (integration tests), and the class level, where tests examine each individual class (unit tests or logic tests).

To examine the microscopic level of a single class and isolate it from the macroscopic level of the system, a controlled running environment is needed. A developer

has to focus on a single class, and the rest of the system is assumed to be "correct." Attempting to test a single class inside the real system is difficult, because for any bugs encountered, it's not immediately clear whether they happen because of the class under test or the environment.

For this reason, a mocking framework is needed that "fakes" the rest of the system and leaves only the class under test to be "real." The class is then tested in isolation because even though it "thinks" that it's inside a real system, in reality all other collaborating classes (collaborators) are simple puppets with preprogrammed input and output.[2]

In the JUnit world, an external library is needed for mocking. Numerous libraries exist, with both strengths and weaknesses—for example, Mockito, jMock, EasyMock (http://easymock.org/) and PowerMock (www.powermock.org/). Spock comes with its own built-in mocking framework, as you'll see in chapter 6. Combined with the power of Groovy meta-programming (as described in chapter 2), Spock is a comprehensive DSL that provides all the puzzle pieces needed for testing.

Now that we've covered the theory and you know the foundations of a solid testing process and how Spock can test classes written in Java, it's time to delve into code!

1.5　*A first look at Spock in action*

The following examples should whet your appetite, so don't stress over the strange syntax or any unknown keywords. I cover Spock syntax throughout the rest of the book.

1.5.1　*A simple test with JUnit*

When introducing a new library/language/framework, everybody expects a "hello world" example. This section shows what Spock looks like in a minimal, but fully functional example.

The following listing presents the Java class you'll test. For comparison, a possible JUnit test is first shown, as JUnit is the de facto testing framework for Java, still undisputed after more than a decade.

Listing 1.2　Java class under test and JUnit test

```java
public class Adder {                          ◄── A trivial class that will be tested
    public int add(int a, int b) {                (a.k.a. class under test)
            return a+b;
    }
}                                     Test case for the
public class AdderTest {          ◄── class in question
    @Test
    public void simpleTest() {                            JUnit assert statement
            Adder adder = new Adder();                    that compares 2 and
            assertEquals("1 + 1 is 2", 2 ,adder.add(1, 1));  ◄── the result of add(l,l)
```

Initialization of class under test (arrow pointing to `Adder adder = new Adder();`)

---

[2]　I always enjoyed this evil aspect of testing—my own puppet theater, where the protagonist can't see behind the scenes.

```
        }
        @Test
        public void orderTest() {
                Adder adder = new Adder();
                assertEquals("Order does not matter ",5,adder.add(2, 3));
                assertEquals("Order does not matter ",5,adder.add(3, 2));
        }
}
```

Two assert statements that compare 5 with adding 2 and 3

A second scenario for the class under test

You introduce two test methods, one that tests the core functionality of your `Adder` class, and one that tests the order of arguments in your `add` method.

Running this JUnit test in the Eclipse development environment (right-click the .java file and choose Run As > JUnit Test from the menu) gives the results shown in figure 1.9.
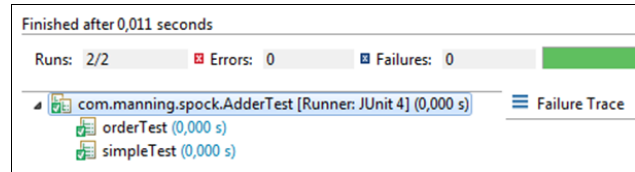
Figure 1.9   Running a JUnit test in Eclipse

1.5.2   *A simple test with Spock*

The next listing shows the same test in Groovy/Spock. Again, this test examines the correctness of the Java class `Adder` that creates the sum of two numbers.

Listing 1.3   Spock test for the `Adder` Java class

```
class AdderSpec extends spock.lang.Specification{
    def "Adding two numbers to return the sum"() {
        when: "a new Adder class is created"
        def adder = new Adder();

        then: "1 plus 1 is 2"
        adder.add(1, 1) == 2
    }
    def "Order of numbers does not matter"() {
        when: "a new Adder class is created"
        def adder = new Adder();

        then: "2 plus 3 is 5"
        adder.add(2, 3) == 5

        and: "3 plus 2 is also 5"
        adder.add(3, 2) == 5
    }
}
```

All Spock tests extend the Specification class.

A Groovy method with a human-readable name that contains a test scenario

A "when" block that sets the scene

Initialization of Java class under test

A "then" block that will hold verification code

A Groovy assert statement

Another test scenario

An "and" block that accompanies the "then" block

If you've never seen Groovy code before, this Spock segment may seem strange. The code has mixed lines of things you know (for example, the first line with the `extends`
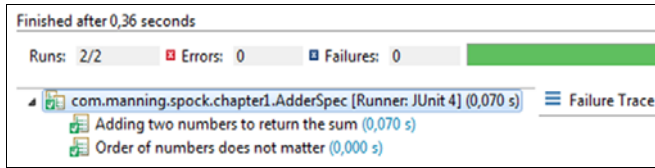
Figure 1.10   Running a
Spock test in Eclipse

keyword) and things completely alien to you (for example, the `def` keyword). Details
on Groovy syntax are explained in chapter 2.

On the other hand, if you're already familiar with BDD, you'll already grasp the
when/then pattern of feature testing.

The upcoming chapters explain Spock syntax in detail. For example, the `def` key-
word (which stands for *define*) is how you declare things in Groovy without explicitly
specifying their type (which is a strict requirement in Java). The Spock blocks (`when:`,
`then:`, `and:`) are covered in chapter 4.

How do you run this test? You run it in the same way as a JUnit test! Again, right-
click the Groovy class and choose Run As > JUnit Test from the pop-up menu. The
result in Eclipse is shown in figure 1.10.

Other than the most descriptive method names, there's little difference between
the JUnit and Spock results in this trivial example. Although I use Eclipse here, Spock
tests can run on all environments/tools that already support JUnit tests (for example,
IntelliJ IDEA).

TAKEAWAYS FROM THESE CODE EXAMPLES

Here's what you need to take away from this code sample:

- The almost English-like flow of the code. You can easily see what's being tested,
  even if you're a business analyst or don't know Groovy.
- The lack of any assert statements. Spock has a declarative syntax, which explains
  what you consider correct behavior.
- The fact that Spock tests can be run like JUnit tests.

Let's move on to one of the killer features of Spock (handling failed tests).

1.5.3   *Inspecting failed tests with Spock*

One of the big highlights of Spock code is the lack of assert statements compared to
JUnit. In the previous section, you saw what happens when all tests pass and the happy
green bar is shown in Eclipse. But how does Spock cope with test failures?

To demonstrate its advantages over JUnit, you'll add another (trivial) Java class that
you want to test:

```
public class Multiplier {
    public int multiply(int a, int b)
    {
            return a * b;
    }
}
```

For this class, you'll also write the respective JUnit test, as shown in the following listing. But as an additional twist (for demonstration purposes), you want to test this class not only by itself, but also in relation to the `Adder` class shown in the previous section.

Listing 1.4   A JUnit test for two Java classes

```java
public class MultiplierTest {
    @Test
    public void simpleMultiplicationTest() {

        Multiplier multi = new Multiplier();
        assertEquals("3 times 7 is 21",21,multi.multiply(3, 7));
    }
    @Test
    public void combinedOperationsTest() {

        Adder adder = new Adder();
        Multiplier multi = new Multiplier();

        assertEquals("4 times (2 plus 3) is 20",
        20,multi.multiply(4, adder.add(2, 3)));
        assertEquals("(2 plus 3) times 4 is also 20",
                      20,multi.multiply(adder.add(2, 3),4));

    }
}
```

A test scenario that will examine two Java classes at the same time → (points to `combinedOperationsTest`)

Creation of the first Java class (points to `Adder adder = new Adder();`)

Creation of the second Java class (points to `Multiplier multi = new Multiplier();`)

Verification of a mathematical result coming from both Java classes (points to assertEquals)

Running this unit test results in a green bar because both tests pass. Now for the equivalent Spock test, shown in the next listing.

Listing 1.5   Spock test for two Java classes

```groovy
class MultiplierSpec extends spock.lang.Specification{
    def "Multiply two numbers and return the result"() {
        when: "a new Multiplier class is created"
        def multi = new Multiplier();

        then: "3 times 7 is 21"
        multi.multiply(3, 7) == 21
    }
    def "Combine both multiplication and addition"() {
        when: "a new Multiplier and Adder classes are created"
        def adder = new Adder();
        def multi = new Multiplier()

        then: "4 times (2 plus 3) is 20"
        multi.multiply(4, adder.add(2, 3)) == 20

        and: "(2 plus 3) times 4 is also 20"
        multi.multiply(adder.add(2, 3),4) == 20

    }
}
```

A test scenario that will examine two Java classes at the same time (points to `def "Combine both multiplication and addition"`)

Creation of the first Java class (points to `def adder = new Adder();`)

Creation of the second Java class (points to `def multi = new Multiplier()`)

Verification of a mathematical result coming from both Java classes (points to `then: "4 times (2 plus 3) is 20"`)

Again, running this test will pass with flying colors. You might start to believe that we gain nothing from using Spock instead of JUnit. But wait!

Let's introduce an artificial bug in your code to see how JUnit and Spock deal with failure. To mimic a real-world bug, you'll introduce it in the `Multiplier` class, but only for a special case (see the following listing).

<div style="background:#a00;color:#fff;padding:4px">

Listing 1.6   Introducing an artificial bug in the Java class under test
</div>

A dummy bug that happens only if the first argument is 4

```java
public class Multiplier {
    public int multiply(int a, int b) {
        if(a == 4)  {
            return 5 * b; //multiply an extra time.
        }
        return a *b;
    }
}
```

Now run the JUnit test and see what happens (figure 1.11).

You have a test failure. But do you notice anything strange here? Because the bug you introduced is subtle, JUnit says this to you:

- Addition by itself works fine.
- Multiplication by itself works fine.
- When both of them run together, we have problem.

But where is the problem? Is the bug on the addition code or the multiplication? We can't say just by looking at the test result (OK, OK, the math might give you a hint in this trivial example).

You need to insert a debugger in the unit test to find out what happened. This is an extra step that takes a lot of time because re-creating the same context environment can be a lengthy process.

SPOCK KNOWS ALL THE DETAILS WHEN A TEST FAILS

Spock comes to the rescue! If you run the same bug against Spock, you get the message shown in figure 1.12.

Spock comes with a super-charged error message that not only says you have a failure, but also calculates intermediate results!
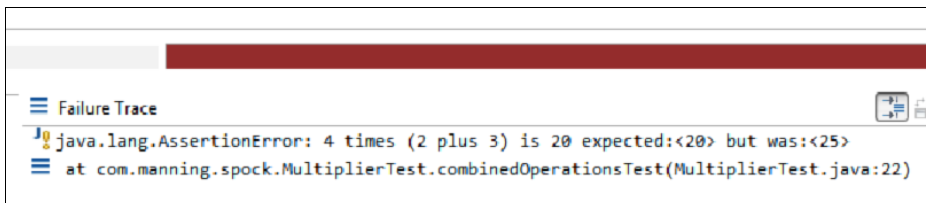

Failure Trace

java.lang.AssertionError: 4 times (2 plus 3) is 20 expected:<20> but was:<25>
  at com.manning.spock.MultiplierTest.combinedOperationsTest(MultiplierTest.java:22)

Figure 1.11   Failure of JUnit test in Eclipse

Figure 1.12 Failure of a Spock test in Eclipse

As you can see, it's clear from the test that the addition works correctly (2 + 3 is indeed 5) and that the bug is in the multiplication code (4 × 5 doesn't equal 25).

Armed with this knowledge, you can go directly to the `Multiplier` code and find the bug. This is one of the killer features of Spock, and may be enough to entice you to rewrite all your JUnit tests in Spock. But a complete rewrite isn't necessary, as both Spock and JUnit tests can coexist in the same code base, which you'll explore next.

1.6 Spock's position in the Java ecosystem

The de facto testing framework in a Java project is JUnit, but *TestNG* is another testing framework for Java that's similar. At one point, TestNG had several extra features that JUnit lacks, resulting in a lot of developers switching over to TestNG (especially for big enterprise projects). But JUnit quickly closed the gap, and TestNG failed to gain a majority in the mindset of Java developers. The throne of JUnit is still undisputed. I've seen junior Java developers who think that JUnit and unit testing are the exact same thing. In reality, JUnit is one of the many ways that unit tests can be implemented.

Unit tests in both JUnit and TestNG are written in Java as well. Traditionally, this has been seen as an advantage by Java developers because they use the same programming language in both production code and testing code. Java is a verbose language (at least by today's standards) with a lot of boilerplate code, several constraints (for example, all code must be part of a class, even static methods), and a heavy syntax requiring everything to be explicitly defined. Newer editions of Java (after version 7) attempt to rectify this issue with mixed success, never reaching the newer "convention-over-configuration" paradigm of other programming languages.

It doesn't have to be this way, though. There's no technical reason to constrain unit tests so that they're in the same programming language as the development code. In fact, production and testing code have completely different requirements. The biggest difference is that testing code runs by definition *before* the application is deployed in production. A good engineer uses the best tool for the job. You can think of Spock as a special domain language created exclusively for testing purposes.

Compilation and running of unit tests is a common task for the developer or the build server inside a software company. Runtime and compile-time errors in unit tests are detected at the same time. Java goes to great lengths to detect several errors during compile time instead of runtime. This effort is wasted in unit tests because these two phases usually run one after the other during the software development lifecycle. The developer still pays the price for the verbosity of Java, even for unit tests. There must be a better way.

Groovy comes to the rescue!

1.6.1 *Making Spock Groovy*

Groovy is a dynamic programming language (similar to Python or Ruby), which means it gives the programmer power to defer several checks until runtime. This might seem like a disadvantage, but this feature is exactly what unit tests should exploit. Groovy also has a much nicer syntax than Java because several programming aspects have carefully selected defaults if you don't explicitly define them (convention over configuration).

As an example, if you omit the visibility modifier of a class in Java, the class is automatically `package private`, which ironically is the least used modifier in Java code. Groovy does the logical thing: if you omit the visibility modifier of a class, the class is assumed to be `public`, which is what you want most times.

The times that I've had to create JUnit tests with `package private` visibility in my programming career: zero! For all these years, I've "paid" the price of declaring all my unit tests (and I guess you have, as well) as `public`, without ever thinking, "There must be a better way!" Groovy has embraced the convention-over-configuration concept, and this paradigm is evident in Spock code as well.

> #### Testing Groovy code with JUnit
>
> The topic of this book is how to test Java code with the Spock framework (which is written in Groovy). The reverse is also possible with JUnit:
>
> - You can write a normal JUnit test in Java, where the class under test is implemented in Groovy.
> - You can also write the JUnit test in Groovy to test Groovy or Java code.
> - Finally, Groovy supports a `GroovyTestCase` class, which extends the standard `TestCase` from JUnit.
>
> Because this is a book about Spock, I don't cover these combinations. See *Making Java Groovy* by Ken Kousen (Manning, 2013) if you're interested in any of these cases.

With Spock, you can gain the best of both worlds. You can keep the tried-and-true Java code in your core modules, and at the same time, you gain the developer productivity of Groovy in the testing code without sacrificing anything in return. Production code is written with verbose and fail-safe Java code, whereas unit tests are written in the

friendlier and lighter Groovy syntax that cuts down on unneeded modifiers and pro-vides a much more compact code footprint. And the best part is that you keep your existing JUnit tests!

1.6.2 Adding Spock tests to existing projects that have JUnit tests

Every new technology faces a big obstacle in its path to adoption: resistance to change. Tradition, inertia, and the projected cost of switching to another technology instead of the mature existing solution are always major factors that affect any proposal for improvement when a better solution comes along.

As an example, *Gradle* is a build system, also written in Groovy, which is in many ways more flexible than the de facto build system of Java (Maven). Using two build sys-tems in a big enterprise project is unrealistic. Gradle has to face the entrenched Maven supporters and convince them that the switch offers compelling advantages.

Spock doesn't suffer from this problem. You can integrate Spock today in your Java project without rewriting or removing a single line of code or configuration. This is a huge win for Spock because it allows a gradual adoption; both old JUnit tests and newer Spock tests can coexist peacefully. It's perfectly possible to implement a gradual Spock adoption strategy in your organization by implementing new tests in Spock dur-ing a trial period without losing anything if you decide to keep implementing JUnit tests as well.

The standard Maven directory structure is flexible in accommodating multiple programming languages. Groovy source code is usually placed in the src/test/groovy folder so that the Groovy compiler plugin can find it. All your Spock tests can go into this directory without affecting your existing JUnit tests located in src/test/java (or other directories), as shown in figure 1.13.
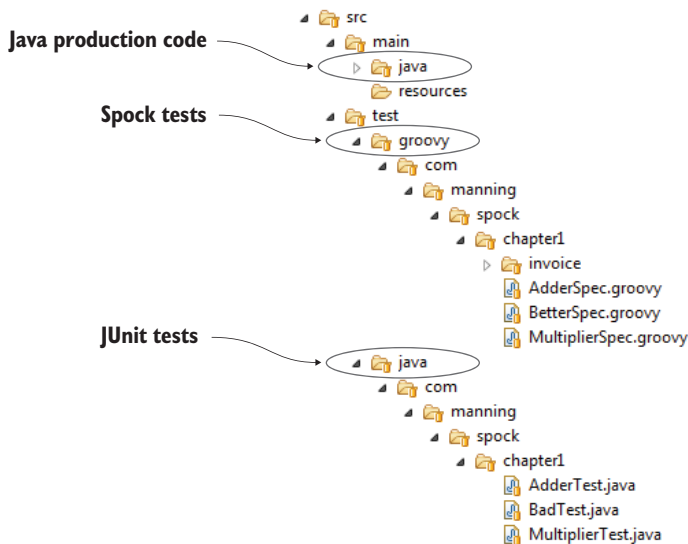
Figure 1.13   Spock tests in a Maven project with existing JUnit tests

For more details on how to set up your IDE for Spock testing, see appendix A.

With the Spock tests in place, the next question you might have is how to run them. You'll be happy to know that Spock comes with a test runner called *Sputnik* (from Spock and JUnit) that runs on top of the existing JUnit runner, thus keeping full backward compatibility.

You can run any Spock test as you run any JUnit test:

- From your development environment
- From the command line
- From Maven/Gradle or any other build system that supports JUnit tests
- From an automated script or build server environment (as explained in chapter 7)

> ### The Spock Web Console
>
> You can also run Spock tests without installing anything at all, with the Spock Web Console. If you visit https://meetspock.appspot.com/, you can play with the Spock syntax and get a feel for how easy it is to write Spock tests by using only your browser.
>
> The Spock Web Console is based on the excellent Groovy Web Console (https://groovyconsole.appspot.com/) that offers a Groovy playground on the web, ready for you to explore from the comfort of your web browser.

1.6.3 Spock adoption path in a Java project

Because Spock is compatible with JUnit runners, it can be introduced gradually in an existing Java code base. Assuming you start with a 100% Java project, as shown at the top left of figure 1.14, Spock can run alongside JUnit tests in the same code base.

It's possible to rewrite all tests in Spock if that's what you want. Spock can work as a superset of JUnit, as you'll see in chapter 3. That situation is shown in the third scenario, depicted at the far right of figure 1.14.
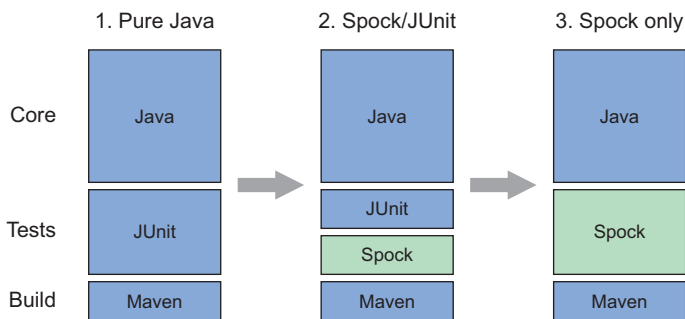

Figure 1.14   Gradual invasion of Spock tests in an existing Java project with JUnit tests

For this book, I assume that you have no prior experience with Groovy. Chapter 2 is fully devoted to Groovy features, and I'll also be careful to explain which new syntax is a feature of Spock and which is a feature of Groovy.

1.7 Comparing Spock and JUnit

Comparing JUnit and Spock in a single section is difficult because both tools have a different philosophy when it comes to testing. JUnit is a Spartan library that provides the absolutely necessary thing you need to test and leaves additional functionality (such as mocking and stubbing) to external libraries.

Spock takes a holistic approach, providing a superset of the capabilities of JUnit, while at the same time reusing its mature integration with tools and development environments. Spock can do everything that JUnit does and more, keeping backward compatibility as far as test runners are concerned.

What follows is a brief tour of some Spock highlights. Chapter 3 compares similar functionality between Spock and JUnit. If you're not familiar with JUnit, feel free to skip the comparisons and follow the Spock examples.

1.7.1 Writing concise code with Groovy syntax

Spock is written in Groovy, which is less verbose than Java. Spock tests are more concise than the respective JUnit tests. This advantage isn't specific to Spock itself. Any other Groovy testing framework would probably share this trait. But at the moment, only Spock exists in the Groovy world. Figure 1.15 shows this advantage in a visual way.

Using Java tools

Core Java code

Tests in JUnit/Mockito

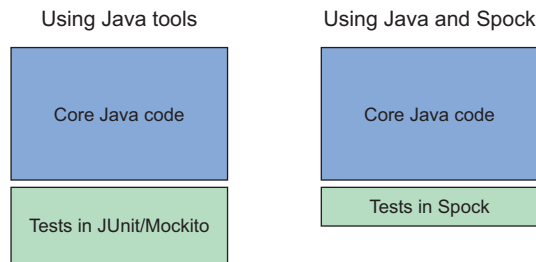Using Java and Spock

Core Java code

Tests in Spock

Figure 1.15　Amount of code in an application with JUnit and Spock tests

Less code is easier to read, easier to debug, and easier to maintain in the long run. Chapter 3 goes into more detail about how Groovy supports less-verbose code than Java.

1.7.2 Mocking and stubbing with no external library

JUnit doesn't support mocking and stubbing on its own. Several Java frameworks fill this position. The main reason that I became interested in Spock in the first place is
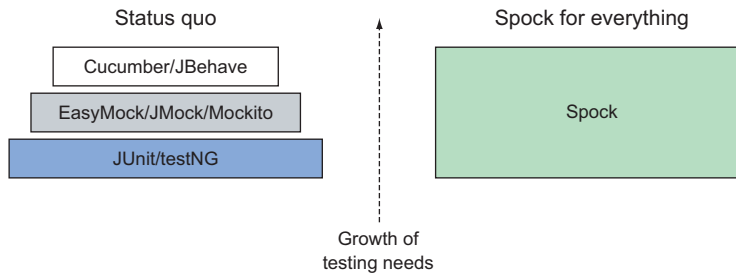
Figure 1.16   **Spock is a superset of JUnit.**

that it comes "batteries included," with mocking and stubbing supported out of the box. As figure 1.16 shows, it does even more than that.

I'll let this example explain:

David goes into a software company and starts working on an existing Java code base. He's already familiar with JUnit (the de facto testing framework for Java). While working on the project, he needs to write some unit tests that need to run in a specific order. JUnit doesn't support this, so David also includes TestNG in the project.

Later he realizes that he needs to use mocking for some special features of the software (for example, the credit card billing module), so he spends time researching all the available Java libraries (there are many). He chooses Mockito and integrates it into the code base.

Months pass, and David learns all about behavior-driven development in his local dev meeting. He gets excited! Again he researches the tools and selects JBehave for his project in order to accomplish BDD.

Meanwhile, Jane is a junior developer who knows only vanilla Java. She joins the same company and gets overwhelmed the first day because she has to learn three or four separate tools just to understand all the testing code.

In an alternate universe, David starts working with Spock as soon as he joins the company. Spock has everything he needs for all testing aspects of the application. He never needs to add another library or spend time researching stuff as the project grows.

Jane joins the same company in this alternate universe. She asks David for hints on the testing code, and he replies, "Learn Spock and you'll understand all testing code." Jane is happy because she can focus on a single library instead of three.

You'll learn more about stubbing/mocking/spying in chapter 6. The semantics of Spock syntax are covered in chapter 4.

1.7.3   *Using English sentences in Spock tests and reports*

The next listing presents a questionable JUnit test (I see these all the time). It contains cryptic method names that don't describe what's being tested.

Listing 1.7   A JUnit test with method names unrelated to business value

```java
public class ClientTest {
    @Test
    public void scenario1()  {

            CreditCardBilling billing = new CreditCardBilling();
            Client client client = new Client();
            billing.chargeClient(client,150);
            assertTrue("expect bonus",client.hasBonus());
    }
    @Test
    public void scenario2()  {

            CreditCardBilling billing = new CreditCardBilling();
            Client client client = new Client();
            billing.chargeClient(client,150);
            client.rejectsCharge();
            assertFalse("expect no bonus",client.hasBonus());
    }
```

Nontechnical people can't understand the test.

A test method with a generic name

Only programmers can understand this code. Also, if the second test breaks, a project manager (PM) will see the report and know that "scenario2" is broken. This report has no value for the PM, because he doesn't know what scenario2 does without looking at the code.

Spock supports an English-like flow. The next listing presents the same example in Spock.

Listing 1.8   A Spock test with methods that explain the business requirements

```groovy
class BetterSpec extends spock.lang.Specification{

    def "Client should have a bonus if he spends more than 100 dollars"() {
            when: "a client buys something with value at least 100"
            def client = new Client();
            def billing = new CreditCardBilling();
            billing.chargeClient(client,150);

            then: "Client should have the bonus option active"
            client.hasBonus() == true
    }
    def "Client loses bonus if he does not accept the transaction"() {
            when: "a client buys something and later changes mind"
            def client = new Client();
            def billing = new CreditCardBilling();
            billing.chargeClient(client,150);
            client.rejectsCharge();

            then: "Client should have the bonus option inactive"
            client.hasBonus() == false
    }
}
```

Business description of test

Human-readable test result

Even if you're not a programmer, you can read the English text in the code (the sentences inside quotation marks) and understand the following:

- The client should get a bonus if he spends more than 100 dollars.
- When a client buys something with a value of at least 100, then the client should have the bonus option active.
- The client loses the bonus if he doesn't accept the transaction.
- When a client buys something and later changes his mind, then the client should have the bonus option inactive.

This is readable. A business analyst could read the test and ask questions about other cases. (What happens if the client spends $99.99? What happens if he changes his mind the next day rather than immediately?)

If the second test breaks, the PM will see in the report a red bar with the title "Client loses bonus if he doesn't accept the transaction." He instantly knows the severity of the problem (perhaps he decides to ship this version if he considers it noncritical).

For more information on Spock reporting and how Spock can be used as part of an enterprise delivery process, see chapter 7.

1.8 Summary

- Spock is an alternative test framework written in the Groovy programming language.
- A test framework automates the boring and repetitive process of manual testing, which is essential for any large application code base.
- Although Spock is written in Groovy, it can test both Java and Groovy code.
- Spock has built-in support for mocking and stubbing without an external library.
- Spock follows the given-when-then code flow commonly associated with the BDD paradigm.
- Both Groovy and Java build and run on the JVM. A large enterprise build can run both JUnit and Spock tests at the same time.
- Spock uses the JUnit runner infrastructure and therefore is compatible with all existing Java infrastructure. For example, code coverage with Spock is possible in the same way as JUnit.
- One of the killer features of Spock is the detail it gives when a test fails. JUnit mentions the expected and actual value, whereas Spock records the surrounding running environment, mentioning the intermediate results and allowing the developer to pinpoint the problem with greater ease than JUnit.
- Spock can pave the way for full Groovy migration into a Java project if that's what you want. Otherwise, it's possible to keep your existing JUnit tests in place and use Spock only in new code.
- Spock tests have the ability to include full English sentences in their code structures, allowing for easy documentation.

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

Free eBook
SEE INSERT

MANNING $44.99 / Can $51.99 [INCLUDING eBOOK]