SAMPLE CHAPTER

BDD IN ACTION

Behavior-Driven Development for the whole software lifecycle

John Ferguson Smart

FOREWORD BY Dan North

MANNING

BDD in Action

by John Ferguson Smart

Chapter 10

brief contents

10 BDD and unit testing

This chapter covers

- The relationship between BDD, TDD, and unit testing
- Going from automated acceptance criteria to implemented features
- Using BDD to discover design and explore low-level requirements
- Tools that help you write BDD unit tests more effectively

So far we've focused on Behavior-Driven Development (BDD) as a tool for discovering, illustrating, and verifying business requirements. But BDD doesn't stop at the business requirements level, or once you've automated your acceptance tests. In this chapter you'll learn how BDD principles and tools can help you write better-designed and better-tested application code (see figure 10.1).

The principles of BDD can be effectively applied at all levels of development, and with significant benefits:

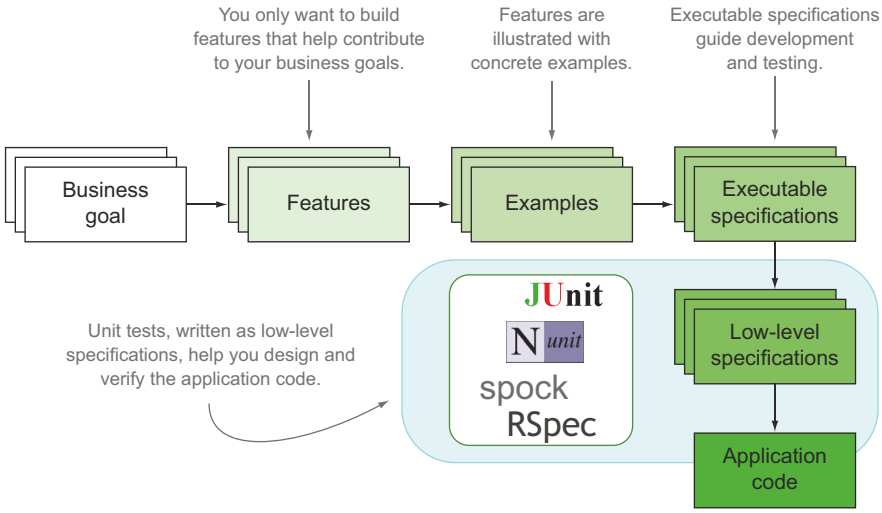


Figure 10.1 In this chapter we'll focus on using BDD practices at the unit-testing level.

- BDD is about writing executable specifications that guide the implementation at all levels of development.
- At a unit-testing level, BDD builds on and extends established TDD practices.
- BDD practitioners use an outside-in approach, using automated acceptance tests and unit tests to drive the implementation of the underlying code.
- You can practice BDD-style unit testing with any tool, but some tools make it easier to write more expressive and more concise unit tests.
- BDD practices at the unit-testing level also help provide living technical documentation of the components and APIs you develop for your application.

Let's start by taking a closer look at the relationship between BDD and TDD.

10.1 BDD, TDD, and unit testing

In this chapter, you'll learn how BDD principles can also help developers write more focused, more effective, more maintainable, and better-documented low-level code, and how unit tests can be used very effectively to express, document, and validate low-level specification and design.

"But isn't that TDD?" I hear you asking. There's often confusion about the distinction between Behavior-Driven Development and Test-Driven Development (TDD). Many developers think of BDD as a technique used for acceptance testing and use "TDD" to refer to lower-level, test-first activities involving unit tests. In fact, things aren't that clear-cut, and the two techniques are deeply intertwined.

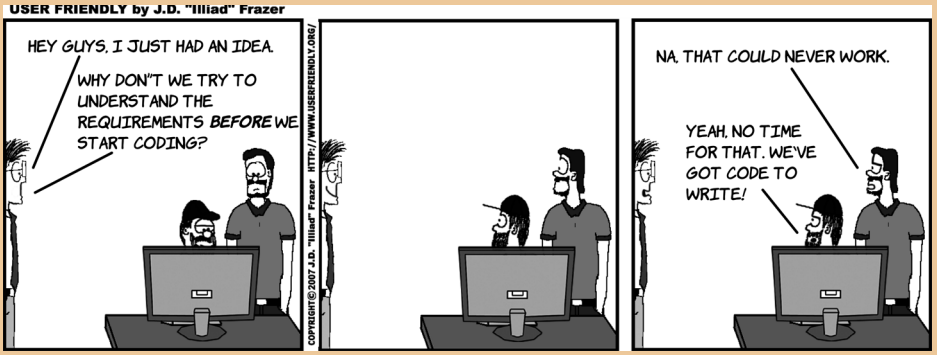
What is Test-Driven Development?

Test-Driven Development (TDD) is a development practice that uses unit tests to specify, design, and verify the code you're writing. Before implementing a piece of functionality, developers write a failing unit test that demonstrates how this functionality should work. At the same time, this failing test also proves that the current implementation doesn't yet support the new functionality. Only then do the developers write the application code. Once the unit test passes, the developers know that the functionality has been successfully implemented. At this stage, they can review their code to tidy things up and fine-tune the design.

TDD has been around for a long time and can be traced back to Extreme Programming practices developed in the late 1990s. TDD relies on two simple principles:

- Don't write any code until you've written a failing test that demonstrates why you need this code.
- Refactor regularly to avoid duplication and keep the code quality high.

The cornerstone of TDD is the idea of writing a unit test before you write the corresponding code. But TDD is much more than just a guarantee that every class has a corresponding set of unit tests. With enough discipline, any experienced developer writing unit tests after writing the code can achieve that outcome as well. TDD's killer feature is that it forces developers to think about the code they're going to write before they write it, in practical and unambiguous terms—you need to understand the functionality before you can write a unit test for it. This way, developers resist the temptation to just start coding something and actively think about what they need to achieve. In this respect, a better term for this practice might be something like "Test-Driven Design."



TDD's killer feature is that it forces developers to think about the requirements before they start to code. This approach isn't always easy for teams to adopt.

This somewhat counterintuitive idea arguably represents one of the single most significant contributions to software engineering quality that has happened over the last 20 years. TDD offers several benefits:

- Cleaner, better designed code
- Code that's easier and less expensive to maintain

(continued)

- Fewer bugs from the outset
- A comprehensive set of regression tests

Experienced developers will typically think about the design of their code before they do any coding, but expressing this design in the form of unit tests makes this process a lot more concrete. Before implementing any code, TDD practitioners imagine the code "they would like to have," which tends to result in cleaner, better-designed APIs. These unit tests also become examples of how to use the application code. And because at least one test is written for every new feature, the code tends to be better tested and have significantly fewer bugs.

Code developed using TDD also benefits from a comprehensive set of regression tests, which, coupled with the clean design that TDD encourages, makes the application easier to change and cheaper to maintain. This, along with the lower bug count, reduces maintenance costs and the total cost of ownership significantly.

On the downside, TDD isn't easy to learn, and it requires a lot of discipline, perseverance, and practice to adopt, especially without the guidance of an experienced TDD practitioner or the support of lead developers and management. As with any new technology or method, productivity will initially take a hit. Teams new to TDD will initially deliver more slowly, though the reduction in defects happens almost immediately.[1]

10.1.1 BDD is about writing specifications, not tests, at all levels

As you've seen, BDD involves discovering and specifying the behavior of a system, but this concept works just as well for the whole application as it does for an individual class: in both cases, you're specifying behavior.

Low-level BDD is a natural continuation of the BDD principles we've been applying to high-level requirements. In the chapters so far, you've seen how BDD practitioners express high-level requirements in the form of executable specifications. High-level requirements deal with the behavior of the system as a whole from the point of view of the business. Low-level requirements deal with the behavior of a component, class, or API from the point of view of the developer working with them. In both cases you're specifying behavior, and in both cases you can describe this behavior in terms of executable specifications. Only the target audience changes: high-level requirements are typically aimed at the broader team, whereas low-level, more technical requirements are aimed at future developers who will have to understand and maintain the application code.

[1] This seems to be supported by a case study from 2008, in which four teams (all new to TDD) reported 40–90% fewer defects, but observed a 15–35% increase in the initial development time. Nagappan, Maximilien, Bhat, and Williams, "Realizing quality improvement through test driven development: results and experiences of four industrial teams," *Empirical Software Engineering*, 13, no. 3 (June 2008): 289–302, http://dl.acm.org/citation.cfm?id=1380664.

10.1.2 BDD builds on established TDD practices

The core practices we're looking at in this chapter are essentially those of TDD, or are deeply rooted in TDD. Indeed, experienced TDD practitioners have been using these techniques for a long time. In many respects, BDD has formalized the way many successful TDD practitioners have been doing things, emphasizing a few key aspects of advanced TDD practice that make the technique more effective. These practices include

- Using outside-in development to ensure that the code you write is delivering real business value
- Using a shared (or "ubiquitous") domain language to encourage closer collaboration and understanding within the team
- Using examples to describe this behavior more clearly
- Describing and specifying the behavior of the system at both a high and a more detailed level

You'll see more of these concepts in the rest of the chapter.

10.1.3 BDD unit-testing tools are there to help

The result of this formalization is that a number of more BDD-flavored unit-testing tools have emerged over recent years that make these techniques easier and more intuitive to practice. Tools like RSpec, NSpec, Spock, and Jasmine have flourished over recent years. While there's no obligation to use these tools (many of our examples will be written simply using JUnit), they can make it easier to write more concise, more expressive, low-level executable specifications.

In the remainder of this chapter, you'll learn how to apply the BDD principles you've seen so far to design and deliver working code. Or, if you prefer, you'll learn to practice TDD with a BDD flavor. If you don't have any experience with TDD, don't worry. I'll introduce all the key concepts as we go.

10.2 Going from acceptance criteria to implemented features

In this section we'll walk through a typical BDD workflow, going from high-level automated acceptance criteria to application code. We'll keep the requirements simple, so that we can focus on the process. But before we start, let's take a look at what we mean by "outside-in" development.

10.2.1 BDD favors an outside-in development approach

One of the core practices used in BDD is *outside-in development*. This involves using acceptance criteria to drive the implementation details you need to build business features. You start with the outcomes you expect and use these outcomes to determine what code you need to write.

The process typically iterates over the following steps:

1. Start with a high-level acceptance criterion that you want to implement.
2. Automate the acceptance criterion as pending scenarios, breaking the acceptance criterion into smaller steps.

3 Implement the acceptance criterion step definitions, imagining the code you'd like to have to make each step work.
4 Use these step definitions to flesh out unit tests that specify how the application code will behave.
5 Implement the application code, and refactor as required.

The last two steps, where you use unit tests to describe low-level behavior and build the corresponding application code, are what many developers would describe as a

Figure 10.2 Going from acceptance criteria to production code with outside-in development

form of TDD. In an outside-in approach, the description of the low-level technical behavior of the application flows naturally from discussing, thinking about, automating, and implementing the high-level acceptance criteria (see figure 10.2).

There are many benefits of outside-in development, but the principle motivations are summarized here:

- Outside-in code focuses on business value.
- Outside-in code encourages well-designed, easy to understand code.
- Outside-in code avoids waste.

OUTSIDE-IN CODE IS FOCUSED ON BUSINESS VALUE

Outside-in code starts with the expected outcomes expressed in business terms, so any code you write can be traced back to some form of business value. The acceptance criteria help keep you focused on where the value is coming from and what you're trying to achieve.

OUTSIDE-IN CODE IS WELL-DESIGNED AND EASIER TO UNDERSTAND

When you write code from the outside in, you write an example of how a method should be used before implementing the method. This makes you think about how the code will be used by other developers, which tends to lead to cleaner API design. These examples also illustrate and document your classes and APIs, making the code much easier to understand for any new developers (or your future self).

OUTSIDE-IN CODE AVOIDS WASTE

If you strictly apply this outside-in approach, any code you write is written with the goal of making an acceptance test pass. If this isn't the case, then either some necessary aspect of the system hasn't been specified in the acceptance criteria, or you're writing code that won't be used in production. BDD won't guarantee that no unnecessary code gets written, but it will highlight potential situations where this might be the case.

Let's see what outside-in development looks like in action with a practical example.

10.2.2 Start with a high-level acceptance criterion

When practicing outside-in development, you usually start with the acceptance criteria of the feature or story that you're currently building. Suppose, for example, that you need to add support for member status levels to your Frequent Flyer application. Flying High Frequent Flyers have different status levels (Bronze, Silver, Gold, and Platinum) depending on how often they fly. Members earn status points with each flight, and when they've earned enough points over the period of a year, they move up to the next level. To encourage travellers to fly with Flying High, members with higher status levels get extra privileges, such as access to lounges, fast lanes, and so forth. At the end of the year, members retain their status level, but their status points are reset to zero.

When implementing a feature, BDD practitioners like to begin with the high-level acceptance criteria, often automating these acceptance criteria using tools like Cucumber, JBehave, and SpecFlow. Using Java and Cucumber, the acceptance criteria might look like this:

```
Feature: Frequent Flyer status is calculated based on points
  As a Frequent Flyer member
  I want my status to be upgraded as soon as I earn enough points
  So that I can benefit from my higher status sooner
```

Recall the business goals behind this requirement.

```
Scenario: New members should start out as BRONZE members
  Given Jill Smith is not a Frequent Flyer member
  When she registers on the Frequent Flyer program
  Then she should have a status of BRONZE
```

This scenario describes what status members should get when they start out.

```
Scenario Outline:
  Given Joe Jones is a <initialStatus> Frequent Flyer member
  And he has <initialStatusPoints> status points
  When he earns <extraPoints> extra status points
  Then he should have a status of <finalStatus>
```

This scenario describes the actual status earning process.

```
  Examples: Status points required for each level
    | initialStatus | initialStatusPoints | extraPoints | finalStatus |
    | Bronze        | 0                   | 300         | Silver      |
    | Bronze        | 100                 | 200         | Silver      |
    | Silver        | 0                   | 700         | Gold        |
    | Gold          | 0                   | 1500        | Platinum    |
```

Illustrate the number of points needed for each level.

This is a simple tangible business requirement with a clear value proposition and two acceptance criteria. Still, it's probably too big to write in one pass, so you'd typically implement the individual acceptance criteria one at a time. In any case, the first step is to automate the step definitions for these scenarios.

10.2.3 Automate the acceptance criteria scenarios

The first acceptance criterion describes the initial status level of a new Frequent Flyer member:

```
Scenario: New members should start out as BRONZE members
  Given Jill Smith is not a Frequent Flyer member
  When she registers on the Frequent Flyer program
  Then she should have a status of BRONZE
```

Background and context

The behavior under test

The expected outcome

You could automate this scenario using the techniques we discussed in the previous chapters. In Cucumber, for example, the step definitions might look something like this:

Background and context

```
@Given("^(\\S*) (\\S*) is not a Frequent Flyer member$")
public void not_a_Frequent_Flyer_member(String name) throws Throwable {...}
```

The behavior under test

```
@When("^(?:s?)he registers on the Frequent Flyer program$")
public void registers_on_the_Frequent_Flyer_program() throws Throwable {...}
```

The expected outcome

```
@Then("^(?:s?)he should have a status of (.*)$")
public void should_have_status_of(FrequentFlyerStatus expectedStatus) {...}
```

But it's when you actually implement the step definitions that you really discover what code you need to write.

10.2.4 Implement the step definitions

Writing a step definition is an exercise in code design. When you write the step definition, you come in contact with your application code for the first time. Although you may have some ideas about the high-level architecture and design, the step definition code is where the rubber meets the road and your design takes the form of real code for the first time.

> **NOTE** Because this chapter focuses on BDD and TDD unit-testing practices, we'll concentrate on acceptance criteria that manipulate the application code directly. Chapter 8 discusses automating acceptance criteria where a UI is involved.

Of course, not all steps are equal. Some step definitions contain relatively simple test code, whereas others involve more forethought and design effort.

In this case, the first step is very simple indeed:

```
String firstName;
String lastName;

@Given("^(\\S*) (\\S*) is not a Frequent Flyer member$")
    public void not_a_Frequent_Flyer_member(String firstName,
                                            String lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
}
```

❶ **Store the new member's name here.**

❷ **Note the name of the new member for use in the following step definitions.**

In the first step ❶, you need to ensure that Jill isn't already a Frequent Flyer member. There's little to do here except record the name of the prospective member so that you can reuse it in the subsequent steps.

The next step definition ❷ is a little more interesting, as you introduce the domain concept of the Frequent Flyer member. But before you start writing code, you'd typically make sure that you have a reasonable understanding of this domain concept.

10.2.5 Understand the domain model

Although incremental and emergent design works well in many situations, a team practicing BDD doesn't exclude more structured design practices where appropriate. In particular, BDD draws many concepts from Domain-Driven Design, including the idea of a common (or *ubiquitous*) language shared by everyone in the team, from business stakeholders to developers. This common language is founded on a high-level domain model, which can be built up incrementally as features are added, as you'll do in this example. This isn't the only option: many teams, particularly larger ones, often draw up a broader domain model supporting several features early on in the iteration. In a similar way, BDD teams will usually give some forethought to the high-level application design and architecture before the coding starts.

In our example, an important business entity has emerged: the Frequent Flyer member. From conversations with the business, you might learn that a Frequent Flyer

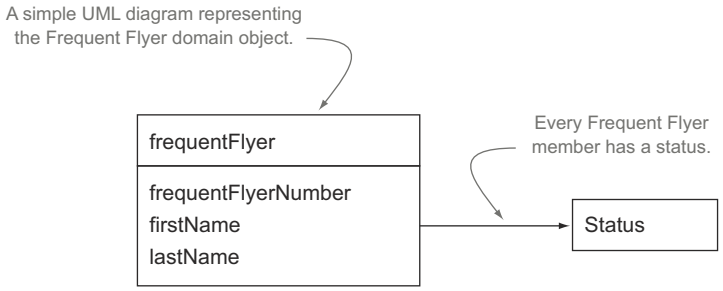


Figure 10.3 The Frequent Flyer domain object as described in discussions with the business

member has a name, a unique Frequent Flyer number, and a current status. Possible status values are Bronze, Silver, Gold, and Platinum. The Frequent Flyer domain model could look like the one in figure 10.3.

Armed with this understanding of the domain, you can now implement the second step definition, which will involve code that manipulates this domain object.

10.2.6 Write the code you'd like to have

When you write a step definition, you write the code you'd like to have if you were a developer using the code. In other words, you imagine the ideal classes and methods for your current needs, and write sample code that illustrates how you'd exercise these classes and methods. This idea of writing the code you'd like to have is an important part of outside-in development.

For example, you might implement the second step definition like this:

```
FrequentFlyer member;

@When("^(?:s?)he registers on the Frequent Flyer program$")
public void registers_on_the_Frequent_Flyer_program() throws Throwable {
    member = FrequentFlyer.withFrequentFlyerNumber("123456789")
                          .named(firstName, lastName);
}
```

Application code illustrating how to create a new Frequent Flyer member ❶

This approach is a great way to design a class, because the code used in the step definitions will be very similar to the code used in the actual application. Writing the code in the step definitions is a way of exploring the objects and methods you'll need to implement a particular acceptance criterion and of discovering the cleanest and most elegant way to do so.

Note that the `FrequentFlyer` class mentioned in the code ❶ doesn't exist yet; you've just specified what it should look like. The next step is to actually implement this code.

10.2.7 Use the step definition code to specify and implement the application code

You can now switch gears and consider the requirements in more granular terms. The high-level requirements expressed in the acceptance criteria often break down into more detailed, low-level requirements.

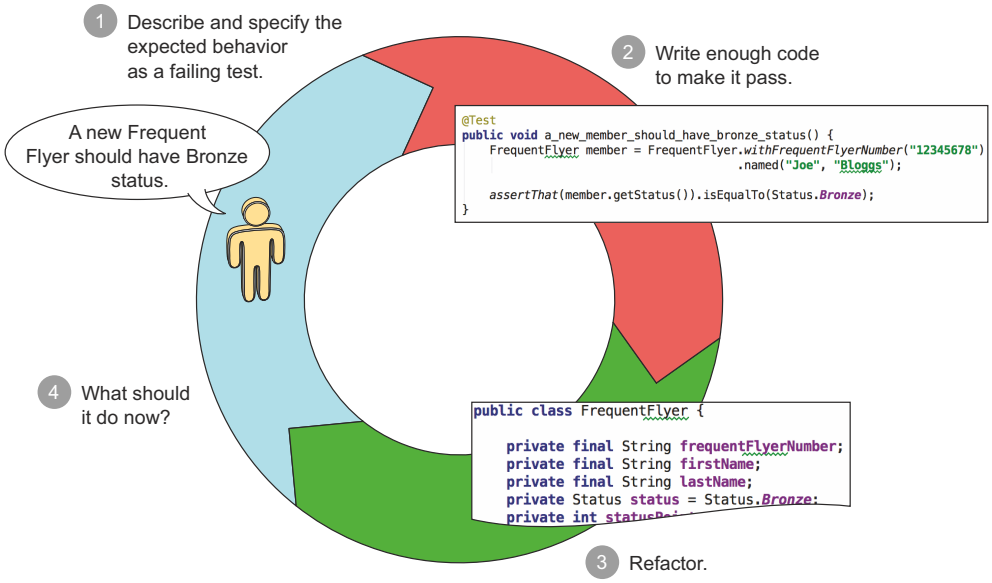


Figure 10.4 Use an incremental approach to progressively describe the behavior of a class.

To flesh out the application code required to implement the high-level step definitions, you write low-level specifications, expressed as unit tests, to progressively describe what your class should do. This approach of using unit tests to drive low-level design is a core TDD practice. The overall process is illustrated in figure 10.4, and can be summarized in the following steps:

1. Describe the behavior you need for a small piece of functionality in the form of a failing test.
2. Write just enough code to make this test pass.
3. Tidy up the code you just wrote, if required, to simplify it, remove duplication, and improve clarity.
4. Ask yourself "What else should the class do?" or "What input could I provide that should produce a different outcome?" and repeat the process.

When your acceptance criteria pass, you're done!

For example, the step definition for registering a new Frequent Flyer member discussed previously looks like this:

```
FrequentFlyer member;

@When("^(?:s?)he registers on the Frequent Flyer program$")
public void registers_on_the_Frequent_Flyer_program() throws Throwable {
    member = FrequentFlyer.withFrequentFlyerNumber("123456789")
                          .named(firstName, lastName);
}
```

Application code illustrating how to create a new Frequent Flyer member (1)

This code illustrates how to create a new Frequent Flyer member ❶, but it lacks precision. You're assuming that the first name comes before the last name, but this isn't demonstrated explicitly. This sort of detail would be useful for a developer using the `FrequentFlyer` class to implement other features, but it would be of little interest to the business. In other words, it's a low-level technical requirement. Let's see how each step of this process works.

DESCRIBE THE EXPECTED BEHAVIOR AS A FAILING TEST

To illustrate and document this technical behavior more clearly, you start by illustrating what the method used in the step definition code (❶ in the previous code snippet) is expected to do, in the form of a unit test:

```
public class WhenRegisteringANewFrequentFlyerMember {

    @Test
    public void should_be_able_to_create_a_new_member() {
        FrequentFlyer member
            = FrequentFlyer.withFrequentFlyerNumber("123456789")
                           .named("Jill", "Smith");

        assertThat(member.getFirstName()).isEqualTo("Jill");
        assertThat(member.getLastName()).isEqualTo("Smith");
        assertThat(member.getFrequentFlyerNumber()).isEqualTo("123456789");#4
    }
}
```

Specify which business requirement you're working on.

❶ Identify which detailed technical requirement or feature you're illustrating.

❷ Create a new Frequent Flyer member.

❸ Verify the expected outcomes.

This method is more than just a unit test; it's a simple and concise executable specification. You've described the requirement ❶, given an example of how you'd like the `FrequentFlyer` API to work ❷, and described the expected outcomes of this operation ❸.

WRITE JUST ENOUGH CODE TO GET THE TEST TO PASS

You can now write some code for the `FrequentFlyer` class, as you see fit, to make this test pass. An important aspect of both TDD and BDD is trying to write the minimum amount of code that will make the test pass. In practice, this may be as little as a single line of code, or it may be a little more for simpler cases or where boilerplate code can be generated by the IDE.

In this case, you could create a simple domain object with a builder class containing the methods required by the API definition:[2]

```
public class FrequentFlyer {

    private String frequentFlyerNumber;
    private String firstName;
    private String lastName;

    protected FrequentFlyer(String frequentFlyerNumber,
                            String firstName,
                            String lastName) {...}
```

Getters omitted for brevity

Private constructor

[2] You can find the tests and application code discussed in this chapter in the sample code for the chapter.

Using a builder pattern

```
    public static FFBuilder withFrequentFlyerNumber(String number) {
        return new FFBuilder(number);
    }

    public static class FFBuilder {

        private String frequentFlyerNumber;

        public FrequentFlyerBuilder(String frequentFlyerNumber) {
            this.frequentFlyerNumber = frequentFlyerNumber;
        }

        public FrequentFlyer named(String firstName, String lastName) {
            return new FrequentFlyer(frequentFlyerNumber,
                                     firstName,
                                     lastName);
        }
    }
}
```

Testing getters and setters

Experienced BDD (and TDD) practitioners generally avoid testing getter and setter methods in themselves. When you need getters, setters, and constructors, they're exercised by the unit tests that describe the behavior that requires them. Remember, your unit tests are executable specifications and living documentation; they describe what the properties of an object are for and how they're used. If you test them explicitly, you're merely stating that your object needs a "name" or a "status" field, for example, but you're not explaining why or how they should be used.

You now have a builder that will allow you to write code like the following to instantiate a new `FrequentFlyer` object:

```
FrequentFlyer member = FrequentFlyer.withFrequentFlyerNumber("12345678")
                                    .named("Joe","Bloggs");
```

REFACTOR IF REQUIRED

Once you've got the test to pass, it's always useful to review your code and see if it can be improved in any way. This includes code-quality considerations such as removing duplicated code, simplifying over-complex algorithms, grouping related lines of code into appropriately named methods, and making sure the variables are well named. But you can also refactor your code from an API perspective: is the API as simple and intuitive as it should be? For example, the `FrequentFlyer` builder might be more readable like this:

```
FrequentFlyer member = FrequentFlyer.withFrequentFlyerNumber("12345678")
                                    .withFirstName("Joe")
                                    .andLastName("Bloggs");
```

You should also include the executable specifications themselves in the refactoring activity; for example, does the name of the unit test reflect the functionality and code being illustrated? Are the variable names clear and accurate? And so on.

WHAT SHOULD THE APPLICATION DO NEXT?

When your unit test (or, more precisely, your executable specification) passes, you need to decide what test to write next. In BDD terms, you ask, "What should the application do that it doesn't already do in order to satisfy the acceptance criteria I'm working on?" The acceptance criteria remind you of what you're trying to achieve and help you avoid being sidetracked.

In this case, with the preceding builder implemented, there's little left to explore in the area of creating new Frequent Flyer members, so you can safely move on to the last step in the acceptance criterion, which checks the status of the Frequent Flyer member. You could implement this step definition as shown here:

```
@Then("^(?:s?)he should have a status of (.*)$")
public void should_have_status_of(Status expectedStatus) {
    assertThat(member.getStatus()).isEqualTo(expectedStatus);
}
```

The Status class represents the concept of member status levels.

You need to be able to get a member's status.

Writing this step definition leads to the discovery of a new domain class (the `Status` class) and a new method to obtain the status of a member. This requirement would probably be simple enough to implement directly from the step definition, but you could also choose to write a more focused unit test to illustrate the initial member status:

```
@Test
public void the_members_initial_status_should_be_bronze() {
    FrequentFlyer member
        = FrequentFlyer.withFrequentFlyerNumber("123456789")
                                    .named("Jill", "Smith");

    assertThat(member.getStatus()).isEqualTo(Status.Bronze);
}
```

Create a Frequent Flyer member.

The initial status should be Bronze.

The implementation for this should be simple. You could implement member status as an enumerated type, like the following:

```
public enum Status { Bronze, Silver, Gold, Platinum; }
```

Then you'd simply add a status attribute and a getter method to the `FrequentFlyer` class:

```
public class FrequentFlyer {

    private String frequentFlyerNumber;
    private String firstName;
    private String lastName;
    private Status status = Status.Bronze;
    ...
    public Status getStatus() {
        return status;
    }
}
```

The first acceptance criteria, as well as the supporting unit tests, should now pass. Once you've verified this and done any necessary refactoring, you're ready to move on to the next acceptance criterion.

The process continues until you've written and implemented a set of low-level specifications for all of the classes involved in the high-level acceptance criteria. At this point, the acceptance criteria should pass.

10.2.8 How did BDD help?

Working this way gives you much finer control over what you're building and more confidence that the code actually works! Although this was a very simple requirement, the process we walked through did illustrate a number of important points.

THE ACCEPTANCE CRITERIA HELP YOU FOCUS ON VALUE

Starting with an acceptance criterion helps you remember what business value you're trying to deliver and provides a clear and convenient indicator so you know when you're done.

SMALL FEEDBACK LOOPS KEEP YOU ON TRACK

Working in very small feedback loops makes a lot of sense. The more code you write without the support of unit tests to back you up, the more chances you have of getting bogged down. The more code you write without a unit test to verify it, the more likely you'll write code that's not covered by a unit test, which both increases the risk of regressions and makes regression issues harder to isolate and fix when they do occur. For an experienced BDD or TDD practitioner, writing the unit tests first is a little like thinking aloud about the design.

THE UNIT TESTS ACT AS LOW-LEVEL SPECIFICATIONS AND DOCUMENTATION

In executable specifications like the previous ones, you're still describing behavior, but your target audience is different. The business stakeholders won't care about how a Frequent Flyer domain object is created, what its default values are, and so forth. But the developers who have to work with and maintain the code will. The unit tests you write are effectively technical specifications and documentation for other developers.

We've just walked through the process of taking a simple acceptance criterion and using it to produce a piece of working, verified production code. But this was a very simple case, and you could almost go directly from the step definitions to the production code. In the next section, we'll look at how to work with more involved acceptance criteria, where you'll need to use unit tests more actively to design and implement the production code.

10.3 Exploring low-level requirements, discovering design, and implementing more complex functionality

The acceptance criterion you implemented in the previous section was a very simple one, and it didn't require much in the way of unit tests or underlying code complexity. But this is usually not the case. In real-world applications, many acceptance criteria

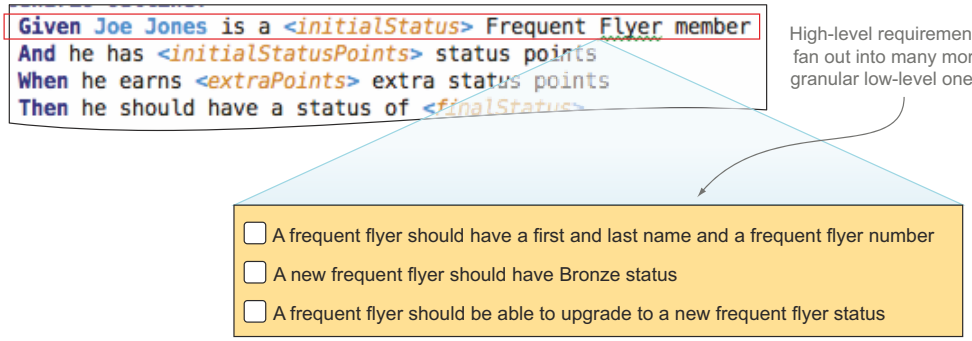


Figure 10.5 High-level requirements in the acceptance criteria fan out into many low-level ones.

hide a large amount of complexity under the hood. In some cases, the automated acceptance test may be sufficient to illustrate and verify this functionality; in other cases, each step in the high-level acceptance criteria may lead to a large number of low-level requirements that you'll express as BDD unit tests (see figure 10.5). This is typical of the BDD work process: as you implement the code required for an acceptance criterion, you'll often discover more low-level requirements that you'll also need to implement.

In this section we'll look at some of the techniques used when you expand high-level acceptance criteria into more detailed unit tests, discovering the classes, methods, and services you need as you go.

The best way to explore this idea is with a practical example. Let's see how this process would play out with the second of the acceptance criteria we introduced earlier in the chapter:

```
Scenario Outline:
  Given Joe Jones is a <initialStatus> Frequent Flyer member
  And he has <initialStatusPoints> status points
  When he earns <extraPoints> status points
  Then he should have a status of <finalStatus>
```

Describe how the application should behave.

```
  Examples: Status points required for each level
    | initialStatus | initialStatusPoints | extraPoints | finalStatus |
    | Bronze        | 0                   | 300         | Silver      |
    | Bronze        | 100                 | 200         | Silver      |
    | Silver        | 0                   | 700         | Gold        |
    | Gold          | 0                   | 1500        | Platinum    |
```

Illustrate the behavior with some basic examples.

Once again, the first task is to automate the steps in the acceptance criteria.

10.3.1 Use step definition code to explore low-level design

In this case, the definition for the first step might look like this:

```
@Given("^(.*) (.*) is a (.*) Frequent Flyer member$")
public void a_Frequent_Flyer_member(String firstName, String lastName, Status
     status) {
```

```
    member = FrequentFlyer.withFrequentFlyerNumber("12345678")
                          .named(firstName,lastName);
    member.setStatus(status);
}
```

❶ Create a new member.

❷ Update the member's status.

Here you reuse the builder method that was introduced in section 10.2.7 to create a new Frequent Flyer member ❶, and then update the status for this member ❷. In doing so, you discover that you need a setter method for the `status` field, which you can simply add to the `FrequentFlyer` class. A simple setter method like this would not justify a separate unit test, as the automated acceptance test is enough to illustrate how this code works.

The next step definition introduces a new concept: you need to keep track of a member's status points:

```
And he has <initialStatusPoints> status points
```

Status points are accumulated for each flight, but the business rules around how many status points are earned for a given flight are complex and vary depending on many factors (the length of the flight, the cabin category, special deals, and so forth). However, a key principle of BDD, and agile development in general, is to keep things simple until you have a good reason to make them more complicated. In this case, the simplest solution for the `FrequentFlyer` class would be to store the current number of status points. Using this approach, the step definition might look like this:

```
@Given("^(?:s?)he has (.*) status points$")
public void earned_status_points(int statusPoints) {
    member.setStatusPoints(statusPoints);
}
```

Again, this is a simple method that doesn't justify any dedicated unit tests.

The next step involves updating the status points of a member account:

```
When he earns <extraPoints> status points
```

Once again, writing the step definition gives you the opportunity to describe the ideal API you'd like to use to perform this operation. For example, you could use a fluent API approach like the one here:

```
@When("^(?:s?)he earns (.*) extra status points$")
public void earn_extra_status_points(int points) {
    member.earns(points).statusPoints();
}
```

This is a more sophisticated API, so you'd typically specify its behavior in more detail in a low-level executable specification, like this one:

```
@Test
public void a_member_should_be_able_to_earn_extra_status_points() {
    FrequentFlyer member =
        = FrequentFlyer.withFrequentFlyerNumber("12345678")
                                    .named("Joe", "Jones");
    member.setStatusPoints(100);
```

Given Joe Jones is a member with 100 status points...

```
    member.earns(150).statusPoints();
    assertThat(member.getStatusPoints()).isEqualTo(250);
}
```

When Joe earns 150 points...

Then he should have a total of 250 points.

This is still pretty simple code. The final step in the acceptance criterion is the interesting part, where you check the business logic around status points and status levels:

```
Then he should have a status of <finalStatus>
```

You've already implemented the step definition for this step:

```
@Then("^(?:s?)he should have a status of (.*)$")
public void should_have_status_of(Status expectedStatus) {
    assertThat(member.getStatus()).isEqualTo(expectedStatus);
}
```

But this is where things get a little more interesting. The acceptance criterion will still not pass, which indicates that the implementation is incomplete. When a member earns additional status points, their status should also be updated if enough points have been accumulated.

You could express this in more technical terms by writing the following unit test:

When a new Frequent Flyer member...

```
@Test
public void should_obtain_a_new_status_when_enough_points_are_earned() {
    FrequentFlyer member
                    = FrequentFlyer.withFrequentFlyerNumber("12345678")
                                   .named("Joe", "Bloggs");
        member.setStatusPoints(100);
        member.earns(200).statusPoints();
        assertThat(member.getStatus()).isEqualTo(Status.Silver);
}
```

With 100 status points...

Earns 200 status points...

Then the member should get Silver status.

This is a low-level example of how the application code updates member status. Note that you've just expressed one of the examples from the acceptance criterion as a unit test. Although this unit test is a little redundant (after all, it tests exactly the same thing as one of the examples in the acceptance criterion), it's still useful, because it acts as the starting point for implementing the status upgrade functionality. It also concisely documents how this piece of functionality is implemented.

10.3.2 Working with tables of examples

Many acceptance criteria use example tables to summarize a number of related scenarios, as you saw in this scenario:

```
Scenario Outline:
  Given Joe Jones is a <initialStatus> Frequent Flyer member
  And he has <initialStatusPoints> status points
  When he earns <extraPoints> status points
  Then he should have a status of <finalStatus>
```

Describe how the application should behave.

Illustrate the behavior with some basic examples.

```
Examples: Status points required for each level
  | initialStatus | initialStatusPoints | extraPoints | finalStatus |
  | Bronze        | 0                   | 300         | Silver      |
  | Bronze        | 100                 | 200         | Silver      |
  | Silver        | 0                   | 700         | Gold        |
  | Gold          | 0                   | 1500        | Platinum    |
```

When you implement this feature, you'll use these examples to guide your implementation. You may even expand on these examples to include edge cases or boundary conditions that are important from a technical perspective but of little interest to the business. At the unit-testing level, there are several approaches you can take.

Some developers write a single unit test to illustrate the technical implementation, and let the acceptance test verify the bulk of the examples. This works fine if the tests run quickly and the implementation is identical or very similar for each example. But when you're incrementally building a solution, it can be more convenient to have a unit-test case for each example, adding a new unit test for each example as you expand your implementation to cater to the different cases.

As you saw in the previous section, you can do this easily enough by writing separate unit tests for every example. While this works fine, it can be long-winded and repetitive, and can make refactoring harder.

Many unit-testing tools support example-driven testing to some extent. For example, both JUnit and NUnit support parameterized tests, which allow you to pass a table of values into a single unit test. The following example is in JUnit.

Listing 10.1 A data-driven unit test in JUnit

```
@RunWith(Parameterized.class)
public class WhenEarningStatusLevels {

    @Parameters
    public static Collection pointsPerStatus() {
        return Arrays.asList(new Object[][]{
                {Bronze, 0,   100,  Bronze},
                {Bronze, 0,   300,  Silver},
                {Bronze, 100, 200,  Silver},
                {Silver, 0,   700,  Gold},
                {Gold,   0,   1500, Platinum}
        });
    }

    Status initialStatus, finalStatus;
    int initialPoints, earnedPoints;

    public WhenEarningStatusLevels(Status initialStatus,
                                   int initialPoints,
                                   int earnedPoints,
                                   Status finalStatus) {
        this.initialStatus = initialStatus;
        this.initialPoints = initialPoints;
        this.earnedPoints = earnedPoints;
        this.finalStatus = finalStatus;
    }
```

The test data goes here.

The test data values are stored in these fields.

The test data is passed into the unit test via the constructor.

```
    @Test
    public void should_earn_new_status_based_on_point_thresholds() {
        FrequentFlyer member
                    = FrequentFlyer.withFrequentFlyerNumber("12345678")
                                   .named("Joe", "Jones")
                                   .withStatusPoints(initialPoints)
                                   .withStatus(initialStatus);

        member.earns(earnedPoints).statusPoints();

        assertThat(member.getStatus()).isEqualTo(finalStatus);

    }
}
```

The test is run once for each row of data.

More modern unit-testing tools like Spock and Spec2 (see section 10.4.3) provide built-in support for data tables. For example, using Spock (which we looked at briefly in chapter 2), the unit test in listing 10.1 might look like the following listing.

Listing 10.2 A data-driven unit test in Spock

```
class WhenEarningStatus extends Specification {

    def "should earn status based on the number of points earned"() {
        given:
        def member = FrequentFlyer.withFrequentFlyerNumber("12345678")
                .named("Joe", "Jones")
                .withStatusPoints(initialPoints)
                .withStatus(initialStatus);

        when:
        member.earns(earnedPoints).statusPoints()

        then:
        member.status == finalStatus

        where:
        initialStatus | initialPoints | earnedPoints | finalStatus
        Bronze        | 0             | 100          | Bronze
        Bronze        | 0             | 300          | Silver
        Bronze        | 100           | 200          | Silver
        Silver        | 0             | 700          | Gold
        Gold          | 0             | 1500         | Platinum
    }
}
```

Create a new Frequent Flyer member.

The member earns some points.

Check the status.

The test data used in the previous steps comes from here.

In both these examples, using data tables makes the unit test easier to read, extend, and maintain.

10.3.3 Discover new classes and services as you implement the production code

So far, we've looked at several ways to describe what you'd like your application code to look like and how it should behave. You've described how you want to be able to upgrade a member's status in the code, but you haven't written any corresponding

application code. To get status upgrades to work, and the tests to pass, you need to write some application code. For example, you could make the following changes to the `FrequentFlyer` class:

```
StatusService statusService;

public void setStatusPoints(int statusPoints) {
    this.statusPoints = statusPoints;
    updateStatusLevel();
}

private void updateStatusLevel() {
    setStatus(statusService.statusLevelFor(statusPoints));
}
```

You need a service to tell you what status can be obtained for a given number of points.

Update the status level to the appropriate level.

Here you've discovered the need for a new service (`StatusService`) and method (`statusLevelFor()`). This service needs to provide the status level that corresponds to a given number of status points.

Notice how the principle of outside-in development isn't limited to tests. When you write implementation code, you can use the same principle, writing the code you'd like to have, and use this process to discover the classes and methods that you need. Whenever you write a piece of code that doesn't yet exist, you're discovering a new low-level requirement.

In general, when you come across something you need from another class or method, you have several choices:

- Implement the class or method immediately.
- Implement a minimum version of the class, and come back to it later.
- Defer implementation by using a "fake" class (a stub or mock) until your current test works, and then go back to implement the class.

Each of these approaches has advantages and trade-offs; experienced practitioners typically know how to use each, and how to pick the most appropriate approach for a given situation. In the following sections we'll look at each of these approaches briefly.

10.3.4 Implement simple classes or methods immediately

Specifying and implementing immediately works well for simple, obvious implementations that can be done quickly. This is what you did in the previous section, where you implemented the `Status` class and the `getStatus()` method of the `FrequentFlyer` class directly, without needing any dedicated unit tests.

Oftentimes when you start to implement an apparently simple method or class, you realize that it's more complicated than you initially thought. If a class has any nontrivial behavior, it's a good idea to describe this behavior through "executable specification"-style unit tests.

To do this, you need to put your current test on standby while you implement the new method or class. For example, in JUnit, you might use the `@Ignore` annotation to temporarily skip this test:

```
@Ignore                                          <-- This test will now be skipped.
@Test
public void should_obtain_a_new_status_when_enough_points_are_earned() {
    ...
}
```

You could then implement the `StatusService` class or interface, and the `statusLevelFor()` method, using a test-driven approach (an example of doing this is illustrated in section 10.3.7). Once the method works, you'd come back and remove the `@Ignore` annotation, to ensure that the method works as expected in its original context.

This approach is simple and intuitive. But it does mean that you'll momentarily have more than one unit test in progress, and you'll need to remember to go back and restore the skipped test later on.

10.3.5 Use a minimal implementation

If the service class looks too complicated to implement in one go, another option is to code a minimal implementation of the real service class, and to use that for your test. This can help you get a feel for an API and how the classes should interact. Once you've finished with the `FrequentFlyer` code, for example, you'd come back and flesh out the status service implementation. Technically, this is an integration test rather than a unit test, which may have performance implications down the track.

10.3.6 Use stubs and mocks to defer the implementation of more complex code

Another approach, commonly used in outside-in development, is to use a stub or mock class to act as a placeholder for the real implementation, and at the same time to describe the behavior you expect of the new class. A stub or mock class is essentially a class you can control for testing purposes that stands in for a real class.

Using stubs and mocks helps you focus on the test at hand and avoids having several unit tests in progress at the same time. It also helps you clarify the roles and responsibilities of the classes and services in your application, which is very useful for larger, more complex applications. But it does add some overhead and complexity to your test classes.

> **USING MOCKS AND STUBS** For a more detailed discussion of using mocks and stubs to discover and define application design, be sure to read the seminal book *Growing Object-Oriented Software, Guided by Tests*, by Steve Freeman and Nat Pryce (Addison Wesley Professional, 2009).

Let's see what this approach would look like for the `StatusService` classes.

MAKE SURE YOU CAN PROVIDE A CUSTOM VERSION OF THE SERVICE TO YOUR CLASS

Remember, you need the `StatusService` class to tell you what status a Frequent Flyer member should achieve with a given number of points:

```
StatusService statusService;
...
private void updateStatusLevel() {
    setStatus(Status.statusLevelFor(statusPoints));
}
```

You've also discovered that the `FrequentFlyer` class needs the `StatusService` to do its job, so you'll need to make sure the `FrequentFlyer` class has an instance of `StatusService` available to use. There are many ways you could do this, such as using dependency injection frameworks like Spring or Guice in Java. But for testing purposes, you need to be able to provide your own custom version of the `StatusService` class:

```
protected FrequentFlyer(String frequentFlyerNumber,
                        String firstName,
                        String lastName,
                        Status status,
                        int statusPoints,
                        StatusService statusService) {
    ...
    this.statusService = statusService;
}
```

Pass a StatusService object in the constructor.

SPECIFY FOCUSED BEHAVIOR USING MOCK SERVICES

When you use this approach, a unit test focuses on specifying the behavior of a particular class or method and describes what it expects of any other classes it needs. This makes it very clear what other services your class needs to work. More interestingly, it gives you a chance to define the "contract" between your class and the service class you're calling.

Mocking libraries exist for all modern testing frameworks, and you can also write your own stubs and mocks if you prefer. The following code uses the popular Mockito library for Java (https://code.google.com/p/mockito/):

```
@Test
public void should_cumulate_points_with_each_flight() {

    // GIVEN
    StatusService statusService = mock(StatusService.class);
    when(statusService.statusLevelFor(300)).thenReturn(Status.Silver)

    FrequentFlyer member = new FrequentFlyer("12345678","Joe", "Bloggs",
                                             statusService);

    // WHEN
    member.earns(100).statusPoints();
    member.earns(200).statusPoints();

    // THEN
    assertThat(member.getStatusPoints(),is(greaterThanOrEqualTo((300))));
    assertThat(member.getStatus(), is(Status.Silver));
}
```

Use a mock version of the status service.

Return "Silver" for a value of 300.

Use the mock status service for this Frequent Flyer member.

Here you assume that the status service works, and that it returns the correct status when you ask about a given number of points. So this test could read, "Assuming that the status service tells us that the status level for 300 points is Silver, when a member

earns 300 points, then they should obtain a status of Silver." Once you have this test working, you could proceed to implement the real version of this class.

This test is now totally independent of any particular implementation of the `StatusService` class. The test will run quickly and require no setup, and if it fails you'll know that the problem is likely to be in the `FrequentFlyer` class and not in the `StatusService` class.

Of course, this now means that you need to write a separate specification for the `StatusService` class. In the next section you'll see an example of some of these low-level specifications.

10.3.7 Expand on low-level technical specifications

In the previous section, you saw how mocks can be used to define contracts between classes. After these contracts are defined, you need to come back to ensure that your service class respects this contract. Likewise, in section 10.3.5 we discussed using a minimal implementation of the status service, but not one that would be sufficient for the full acceptance test. In both cases, you need to come back to complete the implementation.

Suppose you're implementing this as an in-memory service. You might start off by defining how this service behaves for zero points:

```
StatusService statusService = new InMemoryStatusService();

@Test
public void should_stay_on_bronze_for_zero_points() {
    assertThat(statusService.statusLevelFor(0), is(Status.Bronze));
}
```

The status level for zero points should be Bronze.

Low-level specifications often cover cases that aren't included in the acceptance criteria, such as boundary conditions like this one. Acceptance criteria are high-level examples that don't need to cover every possible input and expected outcome, only the ones of significant business value. Often you'll have more technical edge cases that you'd like to document that aren't included in the acceptance criteria.

Once you've implemented enough code to make the code work for zero points,[3] you might proceed to another edge case:

```
@Test
public void should_stay_on_bronze_up_to_299_points() {
    assertThat(Status.statusLevelFor(299), is(Status.Bronze));
}
```

What is the status level for 299 points?

Then you might proceed to one of the acceptance criteria examples:

```
@Test
public void should_earn_silver_for_300_points() {
    assertThat(Status.statusLevelFor(300), is(Status.Silver));
}
```

What is the status level for 300 points?

[3] We won't dwell on the implementation details here, but you can find a sample solution in the sample code for this chapter.

You'd continue with this process until you're confident that your class does everything it needs to do to satisfy the acceptance criteria you're working on.

The exact number of unit tests you need to write will vary from case to case. You could repeat this process for each status level, including edge cases, boundary conditions, and so on. This would make troubleshooting easier, but it could introduce some duplication between the acceptance tests and the unit test because the status levels are also described in the acceptance criteria; if the requirements change for the acceptance tests, the unit tests would need updating as well. Alternatively, if the `StatusService` used a database table, and the logic was identical for every status level, you might simply illustrate one status level and let the acceptance test do the work for the others.

Wording your requirements well: low-level BDD vocabulary

As you may have noticed in the examples so far, BDD practitioners often like to express requirements using the word "should." For example,

```
"a new frequent flyer should have Bronze status"
```

or

```
"should be able to upgrade a frequent flyer status"
```

This vocabulary isn't an accident. Although some developers prefer more definitive wording like "must" or "will," and produce fine executable specifications and living documentation doing so, the reasoning behind the BDD "should" are worth considering.

Language influences thought patterns. The words you use influence the way your message is perceived. Consider the following subconscious dialogue that our brains go through when reading a requirement:

> Requirement: A new Frequent Flyer member status must be Bronze.
>
> Brain: Yes Sir, very good Sir, right away Sir!

or

> Requirement: A new Frequent Flyer member status should be Bronze.
>
> Brain: Should it? Are there times when it might not be Bronze? What about special deals from marketing where they start out as Silver? What about if they're transferring points from a partner airline?

Only one word has changed, but the way we receive the information is quite different. The word "should" invites a question: "should it?" "Should" helps us justify why we want a class to provide a particular service, or to perform a particular task. Should also implies that any requirement, at any level, can be questioned, leaving communication channels much more open.

This use of the word "should" is, incidentally, quite different from the way the word is used in more traditional requirements documentation. In old-style formal specifications, and in legal documents, words like "must," "should," and "may" often have very precise meanings. For example, in the IETF recommendations, "must" or "shall"

(continued)

refers to a mandatory feature, "should" refers to a recommended one, and "may" refers to something that's optional.[4] The goal of the IETF recommendations is essentially to distinguish the really important requirements from the "nice-to-haves."

BDD doesn't use the word "should" in this way. By the time a feature or requirement gets to the stage of writing BDD-style acceptance criteria, there isn't much room for optional requirements. If it's in the acceptance criteria, it needs doing. If you discover that one of the acceptance criteria isn't essential, you can safely remove it.

You've now seen what a typical BDD/TDD development process looks like. BDD unit testing is more an approach than a particular toolset, but there are tools and techniques that can make practicing BDD unit testing easier. We'll look at a few of these in the next few sections.

10.4 Tools that make BDD unit testing easier

Many tools and techniques can smooth the transition to BDD-style unit-testing practices and make the unit tests you write more descriptive and easier to maintain:

- By adopting a descriptive coding style, you can practice BDD unit tests with traditional unit-testing tools such as JUnit and NUnit.
- Other unit-testing tools, such as RSpec, NSpec, and Jasmine, put the emphasis on writing low-level executable specifications in the form of unit tests, which encourages a more rigorous, test-first approach.
- More recent BDD unit-testing tools such as Spock and Spec2 allow for more expressive and powerful low-level specifications, including example-driven specifications.

Let's start by looking at what you can do with traditional unit-testing tools.

10.4.1 Practicing BDD with traditional unit-testing tools

It can sometimes be less intimidating for teams starting out with BDD to keep their existing unit-testing tools. In addition, teams often have a large investment in existing unit tests using a traditional unit-testing library, and they're understandably reluctant to migrate to a new toolset. In this section, we'll look at a few simple steps you can take to start your journey down the road of BDD unit testing without changing your current unit-testing toolset.

WRITE TEST METHOD NAMES THAT DESCRIBE THE BEHAVIOR

The first step toward a more BDD-style approach to unit testing can be as simple as giving your tests more meaningful names. Traditionally, many unit tests use the convention

[4] See Network Working Group, RFC 2119, "Key words for use in RFCs to Indicate Requirement Levels," www.ietf.org/rfc/rfc2119.txt.

of starting test method names with the word "test." For example, the following JUnit tests follow this convention:

```
@Test
public void testAddStatusPoints() {...}

@Test
public void testTransferPoints() {...}
```

Some earlier unit-testing libraries required you to respect this convention, and many development tools still encourage it by letting you generate a unit test for each method in your class, prefixed with the word "test." And this convention does make it easy to see where a particular method has been tested.

But from a BDD perspective, this approach has some limitations. First, it assumes that the method under test already exists, or that you know what it should be called from the outset. It also binds a test to a particular method, which limits the ways you might want to test a given method and makes for more work when refactoring.

In BDD you focus on specifying the *behavior* of a class, rather than just testing its methods. After all, the methods are simply a way of getting the class to perform some particular task or behavior. To reflect this, instead of naming a test after the method it's testing, you name the test based on what you expect the class to do. For example, in .NET you might write tests like the following:

```
[TestFixture]
public class WhenUpdatingStatusPoints
{
    [Test]
    public void ShouldBeAbleToAddStatusPointsEarnedFromAFlight()
    {...}

    [Test]
    public void ShouldUpdateStatusWhenEnoughStatusPointsAreEarned()
    {...}
}
```

These test names are designed to be part of the living technical documentation, so clarity and readability are key. Some Java and .NET practitioners prefer to use a more readable notation for test names based on underscores instead of CamelCase. For example, in JUnit you could write the following:

```
@Test
public void should_be_able_to_add_status_points_earned_from_a_flight() {
    ...
}

@Test
public void should_update_status_when_enough_status_points_are_earned() {
    ...
}
```

Other practitioners use a slight variation on this approach that respects the use of the "test" prefix. Instead of using the word "should," they use "test that." This works well

for unit-testing libraries that still require the "test" prefix on unit-test method names, as illustrated in the following Python examples:

```
class WhenUpdatingStatusPoints(unittest.TestCase):

    def test_that_status_points_can_be_added(self):
        ...

    def test_that_status_is_updated_when_enough_points_are_earned(self):
        ...
```

In all of these cases, you're describing what a class should do, rather than testing individual methods. This makes the tests read more like technical documentation for the class that describes and illustrates the intended usage of the class, rather than just listing method names prefixed with the word "test." It also makes the test cases more robust: method names, and how you use them, are much more likely to change during refactoring than the expected behavior of the class.

USE TEST CLASS NAMES THAT PROVIDE CONTEXT

In a similar vein, BDD practitioners like to name the test classes in a more meaningful way. The name of a BDD-style unit-test class isn't related to the class being tested (which, when you create the test class, may not even exist). Rather, it gives the context of the behavior being described.

A useful trick for doing this is to prefix the test class name with the word "When." This approach, combined with the more expressive test names discussed earlier, makes the tests read much more like technical specifications:

```
public class WhenUpdatingMemberStatusPoints {

  @Test
  public void should_be_able_to_add_status_points_earned_in_a_flight(){
    ...
  }

  @Test
  public void should_update_status_when_enough_status_points_are_earned(){
    ...
  }
  ...
}
```

The test class name describes the feature or context.

The test names describe the detailed requirements.

These techniques let you obtain many of the benefits of BDD unit testing without needing to change your unit-testing tools. But BDD unit-testing tools that are more focused, such as RSpec for Ruby, NSpec for .NET, and Spock for Java and Groovy, make things a little easier. They make it simpler and more natural to write unit tests in the form of clear, readable, low-level executable specifications. In the next section, we'll look at how they can help.

10.4.2 Writing specifications, not tests: the RSpec family

Inspired by the concepts of BDD, in the mid-2000s a new generation of low-level BDD tools emerged that put a greater distance between "executable specifications" and

"tests." When you write a test in JUnit or NUnit, it's difficult to avoid mentioning it somewhere. You use the `@Test` annotation, for example, or you make assertions about the result value. These conventions come from the mindset of writing unit tests once the code is complete, which, as you've seen, doesn't reflect how BDD and TDD work. Even if you try to adopt more BDD-style naming conventions, xUnit tests are still focused on *verification* rather than *specification*. In BDD, the idea is to use unit tests to implement low-level executable specifications, not to test code that has already been written.

Members of the Ruby community have traditionally been very early adopters of BDD practices and tools, and in 2005 RSpec, the first BDD unit-testing tool, was released (http://rspec.info). RSpec is fairly typical of a whole family of low-level BDD tools, so we'll look at RSpec first, and then see what other similar libraries exist for other languages.

WRITING LOW-LEVEL EXECUTABLE SPECIFICATIONS IN RUBY WITH RSPEC

The main idea behind RSpec is, in true BDD fashion, to think in terms of describing application behavior rather than verifying code. In RSpec, rather than speaking in terms of tests, you talk about *specifications*, expressed in terms of executable examples.

For example, to specify the requirement that a Frequent Flyer should initially have a Bronze Frequent Flyer status, you could write the following specification:

```
describe FrequentFlyer do
  it 'should initially have Bronze status' do
    frequentFlyer = FrequentFlyer.new
    expect(frequentFlyer.status).to eq('BRONZE')
  end
end
```

What should it do or have?

You're talking about the Frequent Flyer domain object.

Create a new Frequent Flyer object.

You expect the status to be Bronze.

This descriptive, example-based approach steers away from mentioning the word "test" at all; it's more in line with the BDD philosophy of living documentation and executable specifications. This can be a great help in getting developers into the BDD mindset of writing specifications rather than tests.

RSpec provides many features that make it easier to structure these executable specifications in a sensible manner. For example, you can group requirements into different contexts to give more background to each set of granular specifications:

```
describe FrequentFlyer do
  context 'when the frequent flyer account is first created' do
    it 'should initially have Bronze status' do
      frequentFlyer = FrequentFlyer.new
      expect(frequentFlyer.status).to eq('BRONZE')
    end
  end

  context 'when a new member starts to fly with Flying High' do
    it 'should earn points for each flight' do
      frequentFlyer = FrequentFlyer.new

      frequentFlyer.earn_status_points(100)
```

1 Given...

Group requirements by context.

Contexts can contain several related requirements.

```
      expect(frequentFlyer.status_points).to eq(100)
    end

    it 'should upgrade member status when enough points are earned' do
      frequentFlyer = FrequentFlyer.new

      frequentFlyer.earn_status_points(100).earn_status_points(200)

      expect(frequentFlyer.status).to eq('SILVER')
    end
  end
end
```

Contexts can contain several related requirements.

❶ **Given...**

❷ **When...**

❸ **Then...**

Specifications written this way should stay simple and concise, respecting the BDD approach of favoring simplicity and communication. If you study the structure of these specifications, you may notice that they actually respect a "Given ... When ... Then" style. For example, in the last specification above, the *Given* creates a new Frequent Flyer object ❶, the *When* is the action you're demonstrating ❷, and the *Then* is the expected outcome ❸. Even in the more concise RSpec style, respecting this structure will make your specifications much cleaner and easier to understand.

These specifications are not only quite readable, they'll also produce reports that describe the low-level specifications in a more structured way than a list of unit-test results. For example, running `rspec` from the command line would produce a summary of the requirements in text form:

```
$rspec --format documentation

FrequentFlyer
  when the frequent flyer account is first created
    should initially have Bronze status
  when a new member starts to fly with Flying High
    should earn points for each flight
    should upgrade member status when enough points are earned
```

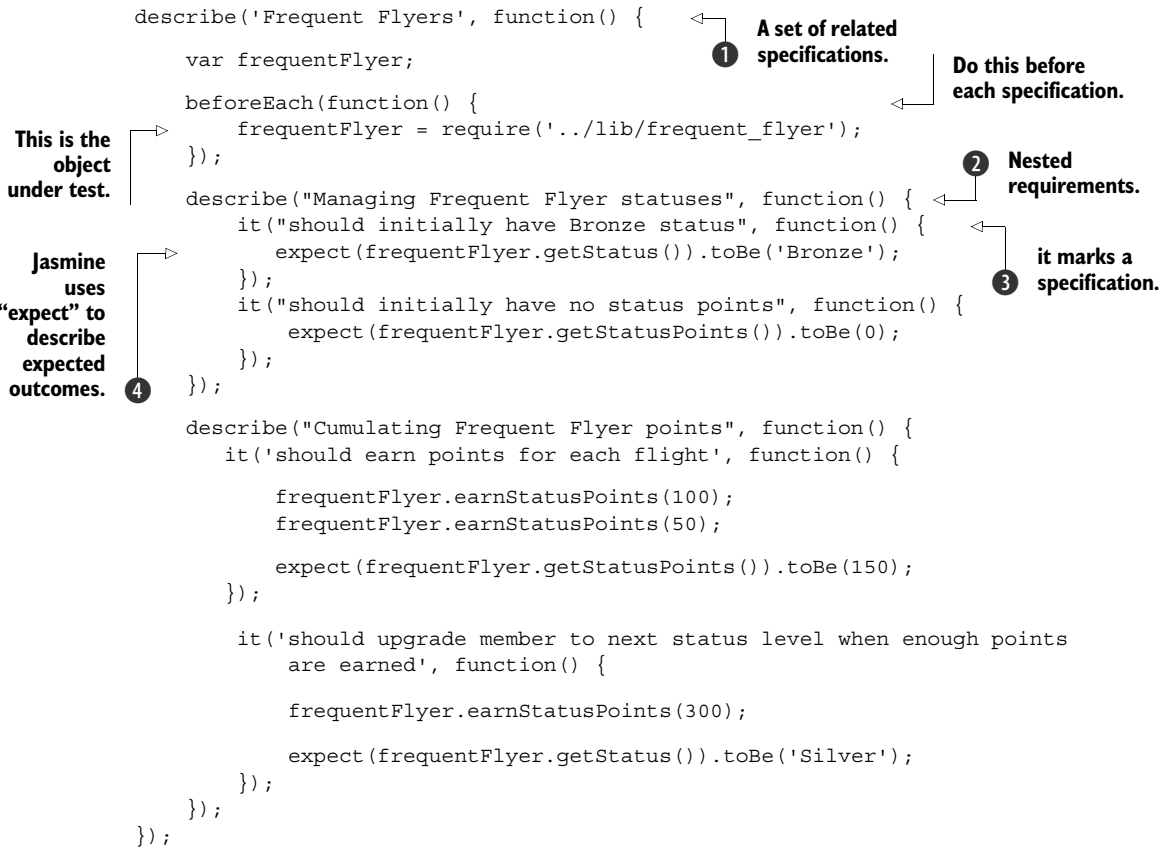
RSpec has many other interesting features that we don't have time to discuss here. Since RSpec was released, other languages have also adopted RSpec-style testing libraries for BDD unit testing, including NSpec for .NET and a number of libraries for JavaScript. As an example, let's look at one of these.

Executable specifications in JavaScript with Jasmine

JavaScript is playing an increasingly important role in modern application development, both on the client side, and, with the rise of JavaScript application platforms like Node.js, on the server side.

Although more traditional unit-testing tools such as QUnit are widely used in the JavaScript world, there are several BDD-style JavaScript unit-testing libraries around. The best known of these are Jasmine (http://jasmine.github.io) and Mocha (http://visionmedia.github.io/mocha/). As far as BDD goes, both these libraries have a very similar style. You've already seen Jasmine in action in chapter 9, where you used it to test an AngularJS web application. Let's take a closer look at how it works.

Jasmine lets you describe granular requirements in a format similar to RSpec. It supports declarative, example-based specifications, nested contexts, and its own mocking library. Here's a simple example of some requirements expressed using Jasmine:




```
describe('Frequent Flyers', function() {

    var frequentFlyer;

    beforeEach(function() {
        frequentFlyer = require('../lib/frequent_flyer');
    });

    describe("Managing Frequent Flyer statuses", function() {
        it("should initially have Bronze status", function() {
            expect(frequentFlyer.getStatus()).toBe('Bronze');
        });
        it("should initially have no status points", function() {
            expect(frequentFlyer.getStatusPoints()).toBe(0);
        });
    });

    describe("Cumulating Frequent Flyer points", function() {
       it('should earn points for each flight', function() {

           frequentFlyer.earnStatusPoints(100);
           frequentFlyer.earnStatusPoints(50);

           expect(frequentFlyer.getStatusPoints()).toBe(150);
       });

        it('should upgrade member to next status level when enough points
            are earned', function() {

            frequentFlyer.earnStatusPoints(300);

            expect(frequentFlyer.getStatus()).toBe('Silver');
        });
    });
});
```

As you can see, Jasmine specifications have a layout that's very similar to the layout of RSpec specifications. Groups of related specifications are marked by the `describe` function ❶, which can contain specifications or more nested `describe` functions ❷. Each specification is marked by the `it` method ❸, which is made up of a title and a function containing the specification test code. Jasmine uses the `expect()` function ❹ to describe expected outcomes, though as you'll see in section 10.5.2, this isn't your only option.

EXECUTABLE SPECIFICATIONS IN .NET WITH NSPEC

You've seen how you can write NUnit tests in a more BDD style. But for teams who prefer a purer BDD approach, NSpec is an elegant BDD library that lets you write RSpec-style specifications directly in .NET (http://nspec.org/).

A sample set of specifications can be seen here:

```
public class WhenUpdatingStatusPoints : nspec
{
    FrequentFlyer member;
```

```
        void before_each()
        {
            member = new FrequentFlyer();
        }

        void earning_status_points()
        {
            context["When the frequent flyer account is created"] = () =>
            {
                it["should have BRONZE status"] = () =>
                    member.getStatus().should_be(Status.Bronze);

                it["should have 0 status points"] = () =>
                    member.getStatusPoints().should_be(0);
            };

            context["When cumulating Frequent Flyer points"] = () =>
            {
                it["should earn points for each flight"] = () =>
                {
                    member.earnStatusPoints(100);
                    member.earnStatusPoints(50);
                    member.getStatusPoints().should_be(150);
                };

                it["should get upgrade when enough points are earned"] = () =>
                {
                    member.earnStatusPoints(00);

                    member.getStatus().should_be(Status.Silver);
                };
            };
        }
    }
```

A new specification context

Specifications within the context

"should"-style assertions

This is very close, at least in spirit, to the RSpec and Jasmine examples you saw earlier. As with those other tools, you describe your specifications in natural language and group them within contexts, using very clean "should"-style assertions.

10.4.3 Writing more expressive specifications using Spock or Spec2

Tools like RSpec, Jasmine, and NSpec are great ways of expressing BDD-style low-level specifications, but they do have some limitations. For example, there are times when it would be nice to have a more explicit "Given ... When ... Then" structure. Although good practitioners do this with whatever tool they're using, it can often help teams to work more consistently if the structures are more visible.

In addition, example-driven specifications can often be expressed more concisely using examples. You saw the use of table-based examples with tools like JBehave and Cucumber, but they're equally useful at the unit-testing level.

RSpec was released over eight years ago, and more recently a new generation of BDD unit-testing tools has emerged. These newer tools provide the expressiveness of Gherkin, including support for example tables and clear "Given ... When ... Then" structures, without the overhead of maintaining separate feature files and step-definition classes.

They do this by mixing the code with the specification definitions, as is done in RSpec, but by also providing support for much richer BDD language structures.

There aren't many unit-testing BDD tools that support these features yet, but there are a few. In Scala, for example, Specs2 is one such tool (http://etorreborre.github.io/specs2/). And if you're working with Java or Groovy, you can use Spock (https://code.google.com/p/spock/). Both tools can be used very effectively to test Java code. Let's take a closer look at Spock.

Spock is a powerful and very expressive BDD unit-testing tool that lets you write clean and readable executable specifications in Groovy. Here's a simple example:

All Spock specifications extend the Specification class.

```
class WhenManagingFrequentFlyerMembers extends Specification {

    def "a new frequent flyer should have Bronze status"() {
        given:
          def member = FrequentFlyer.withFrequentFlyerNumber("12345678").
                                     named("Joe","Bloggs")
        when:
            def status = member.status
        then:
            status == FrequentFlyerStatus.BRONZE
    }
```

❶ **Given**

❷ **When**

❸ **Then**

Notice how this example is structured. The specification is organized into clear *Given* ❶, *When* ❷, and *Then* ❸ sections. Of course, Spock, like Gherkin, gives you a fair bit of flexibility in how you organize your "Given ... When ... Then" blocks, but they're nevertheless present and clearly marked.

Spock also makes example-driven testing very easy.

```
def "should upgrade status when enough status points are acquired"() {
    given: "a frequent flyer member with some points"
        def member = FrequentFlyer.withFrequentFlyerNumber("12345678").
                                   named("Joe","Bloggs").
                                   withStatusPoints(initialPoints).
                                   withStatus(initialStatus)
    when: "he earns some extra points on a flight"
        member.earns(extraPoints).statusPoints()
    then: "he may or may not be upgraded to a new status"
        member.getStatus() == expectedStatus
    where:
        initialStatus | initialPoints | extraPoints | expectedStatus
        BRONZE        | 0             | 299         | BRONZE
        BRONZE        | 0             | 300         | SILVER
        SILVER        | 0             | 699         | SILVER
        SILVER        | 0             | 700         | GOLD
        GOLD          | 0             | 1499        | GOLD
        GOLD          | 0             | 1500        | PLATINUM
}
```

❶ **Precondition**

Action ❷

❸ **Expected outcome**

❹ **Examples**

This specification, using a format that's quite similar to the Gherkin equivalent, feeds test data in from the examples table ❹. The column headers are injected into the *Given* ❶, *When* ❷, and *Then* ❸ steps, where they're used to do the actual test.

This example also adds some extra information about each step. The texts following the `given`, `when`, and `then` labels are optional, but they're often used to make the intent of the specification clearer. Sometimes a bit of extra explanation goes a long way. Other times the code itself is sufficient.

Spock also supports a more lightweight syntax that's useful for example-based testing. For example, you could describe the minimum points required to upgrade to the next status level like this:

```
class WhenCheckingMinimumStatusPoints extends Specification {

    def "should know the minimum points required for each status level"() {
        expect:
            FrequentFlyerStatus.statusLevelFor(points) == expectedStatus
        where:
            points   | expectedStatus
            0        | BRONZE
            299      | BRONZE
            300      | SILVER
            699      | SILVER
            700      | GOLD
            1499     | GOLD
            1500     | PLATINUM
    }
}
```

The status level obtained with a given number of points

The expected statuses for a range of point values

The approach used by tools like Spock and Specs2 hits a sweet spot between expressive executable requirements and technical documentation. When using tools like Cucumber or JBehave, you keep the feature files separate from the step definitions, since the scenarios are discussed, defined, and owned by the team as a whole. Adding implementation code to the mix would make these feature files harder for non-developers to understand. But the more granular, low-level executable requirements we've looked at in this chapter need to combine technical documentation, business justification, and sample code.

10.5 Using executable specifications as living documentation

From a BDD perspective, writing a good unit test is an exercise in good communication. When you practice BDD, you think of every unit test as a low-level specification that illustrates some aspect of how a class or component behaves. You've seen ways to help ensure that the intent of these low-level specifications is clearer. But the implementation of your test is also sample code that illustrates how a particular requirement is satisfied, or how a particular goal is achieved. The code inside your tests doesn't just exercise the application; it documents *how to exercise* the application.

Not keeping your test code clean and easy to understand has very practical consequences. If an old test fails when you make a change elsewhere in the code base, it's essential to understand both what the test was trying to demonstrate and how it was doing so. Only by understanding this will you be in a position to decide what to do with the failing test.

An important part of this is keeping your test code simple. Complex, convoluted test code is hard to understand and maintain, and is more difficult for others (and yourself, later on!) to understand.

One very effective way to make your code easier to read and to understand is to use fluent coding practices.

10.5.1 Using fluent coding to improve readability

Fluent coding can help you write code that communicates your intent more effectively. We say that code is *fluent* when it reads more like a natural-language sentence than like something written for a compiler.

For example, suppose you need to create a flight for your tests. Using conventional code in Java or .NET, you might write something like this:

```
Flight lastPlaneOut = new Flight("FH-525","Hong Kong","Sydney")
```

Although it's globally clear what this code does, there's still room for doubt. For example, you could probably guess that the first parameter represents the flight number, but does the flight leave from Hong Kong or Sydney?

With a little more effort, you could design the testing API so that the tests might read more like this:

```
Flight lastPlaneOut = Flight.number("FH-525").from("Sydney")
                                             .to("Hong Kong");
```

The idea of fluent code isn't language-specific. Dynamic languages like Ruby and Groovy, for example, support relatively fluent constructions natively:

```
def lastPlaneOut = new Flight(from: "Sydney",
                              to:"Hong Kong",
                              number:"FH-525")
```

The benefits of fluent code also apply to assertions. A well-written assertion tells you at a glance what a test is trying to demonstrate. It should be simple and obvious. You shouldn't have to decipher conditional logic or sift through `for` loops to understand what the code is doing. In addition, any nontrivial logic in a test case increases the risk of the test itself being wrong.

In recent years there has been a rise in the popularity of tools and techniques that make it easier to write more fluent code, both for production code and for tests. In the testing space, in particular, many libraries support fluent assertions in different languages. There are two main flavors to fluent assertions. The first typically uses the word "assert," whereas the second uses terms like "should" or "expect."

The first approach comes from a more traditional unit-testing background and focuses on testing and verification. Indeed, you typically make an assertion about something that has already happened, or a result that has already been calculated. The second is more BDD-centric: the words "should" and "expect" describe what you think the application should do, regardless of what it does currently, or if it even exists.

Let's look at some examples of fluent assertion libraries in different languages.

10.5.2 Fluent assertions in JavaScript

JavaScript has a number of libraries that can help make your assertions more expressive. You've seen, for example, the built-in `expect()` method that comes with Jasmine:

```
expect(frequentFlyer.getStatus()).toBe('Silver');
```

But this sort of expressiveness isn't limited to Jasmine. Should.js (https://github.com/visionmedia/should.js/) and Chai (http://chaijs.com) are other well-known libraries that support similar features.

Chai is probably the most flexible of these, supporting both the "expect" and "should" formats, as well as the old-school assert. Chai focuses on using method chaining to make the assertions fluid and readable. For example, the Chai equivalent of the Jasmine `expect` statement would be the following:

```
var expect = require('chai').expect
...
expect(frequentFlyer.getStatus()).to.equal('Silver');
```

Import the Chai expect function.

Expect assertions are similar to those in Jasmine.

As you might expect, Chai supports a rich collection of assertions and can chain multiple assertions together. For example,

```
var obtainableStatuses = ['Silver','Gold','Platinum']
...
expect(obtainableStatuses).to.have.length(3).and.to.include('Gold')
```

Chai also supports the more BDD-style `should` assertion, as illustrated here:

```
var expect = require('chai').should();

frequentFlyer.getStatus().should.equal('Bronze');

obtainableStatuses.should.have.length(3).and.include('Silver');
```

Note that you're calling the should() function, not importing it.

Should can now be called on any object.

All the assertion methods can be called using either expect or should.

Both styles are equally expressive, so the choice is largely a question of style and personal preference.

10.5.3 Fluent assertions in static languages

Fluent assertion libraries also exist for static languages such as Java and .NET, although they're generally a bit less expressive than their dynamic equivalents.

Java, for example, has several fluent assertion libraries. The two best known are Hamcrest (https://github.com/hamcrest/JavaHamcrest) and FEST-Assert (https://github.com/alexruiz/fest-assert-2.x/wiki). More recent versions of NUnit come with a similar constraint-based assert model, and .NET developers can also use the very rich Fluent Assertions library (https://github.com/dennisdoomen/FluentAssertions).

All of these libraries move away from the older-style assert methods and let you express your expectations in a more fluent and concise manner. In particular, they propose a number of higher-level assertions on collections and can be extended to work with domain objects, which helps you to avoid having to put too much logic within your unit tests.

Imagine you want to say that the member status should initially be Bronze. In traditional JUnit, you'd write something like this:

```
assertEquals(BRONZE, member.getStatus());
```

The parameter order and the somewhat clumsy wording make this sort of assertion less than ideal. It doesn't read fluently or naturally, which limits your ability to express your expectations easily and quickly.

An equivalent Hamcrest assertion, on the other hand, would look like this:

```
FrequentFlyer member = FrequentFlyer.withFrequentFlyerNumber("12345678")
                                    .named("Joe", "Bloggs");
assertThat(member.getStatus(), is(FrequentFlyerStatus.BRONZE));
```

FEST-Assert does something similar, but using a different syntactic structure:

```
assertThat(member.getStatus()).isEqualTo(FrequentFlyerStatus.BRONZE);
```

In NUnit, you could write something like this:

```
Assert.That(member.getStatus(), Is.EqualTo(FrequentFlyerStatus.BRONZE));
```

The .NET Fluent Assertions library uses a more natural style built around the word "should:"

```
Member.getStatus().Should().Be(FrequentFlyerStatus.BRONZE);
```

All of these libraries propose a rich set of matchers, including a number of convenient operations on lists. For example, to check that the list of unachieved statuses contains both Gold and Platinum, you could write the following Hamcrest assertion:

```
assertThat(member.getUnachievedStatuses(), hasItems(GOLD,PLATINUM));
```

In FEST-Assert, the equivalent would be similar:

```
assertThat(member.getUnachievedStatuses()).contains(GOLD,PLATINUM);
```

With Fluent Assertions, the assertion might look like this:

```
member.getUnachievedStatuses().Should().Contain(GOLD).And.Contain(PLATINUM);
```

All of these libraries also allow more complex expressions. For example, suppose you wanted to verify that all of the Frequent Flyers in a particular group were under the age of 18. You could express this quite elegantly in Hamcrest like this:

```
List<Integer> memberAges = ...;
assertThat(memberAges, everyItem(lessThan(18)));
```

You could also do something similar in NUnit:

```
Assert.That(memberAges, Has.All.LessThan(18));
```

Alternatively, using the .NET Fluent Assertions library, you could write something like this:

```
memberAges.Should().Contain(item => item < 18);
```

Traditionally, higher-level assertions like this would often require loops and nontrivial logic in the unit tests. This isn't only risky; it's also often enough to discourage developers from testing nontrivial outcomes. By making it easier for developers to express their expectations effectively, fluent assertion libraries contribute to more meaningful and higher quality executable specifications.

Hamcrest and FEST-Assert play similar roles in Java-based BDD. Hamcrest is more flexible and easier to extend, but FEST-Assert has a simpler syntax and is a little easier to use. The constraint-based assert model in NUnit is similar, and it has a particularly rich library of assertions. And the Fluent Assertion library proposes an expressive style of assertions with a very BDD feel to it. All of these are a vast improvement on the traditional assert statements.

Fluent assertion libraries are in no way specific to BDD, and they can be used to make any unit tests easier to understand. But their emphasis on readability, expressiveness, and communication makes them well aligned with the BDD philosophy.

10.6 Summary

In this chapter you learned about practicing TDD techniques using a BDD style:

- Low-level BDD can be considered as an extension of classic TDD, or even "TDD practiced very well."
- BDD acceptance criteria lead to lower-level BDD unit tests, and help you discover the detailed application design.
- BDD unit-testing tools exist for virtually all modern languages and platforms, with the RSpec family of tools being very widespread.
- More recent BDD tools like Spock and Specs2 are more powerful and expressive.
- Fluent assertion libraries make it easier to express your expectations clearly.

In the next chapter, we'll look at how BDD fits into the broader picture of project management and reporting, and how to get the most out of your living documentation.

AGILE DEVELOPMENT/TESTING

BDD IN ACTION

John Ferguson Smart

You can't write good software if you don't understand what it's supposed to do. Behavior Driven Development (BDD) encourages teams to use conversation and concrete examples to build up a shared understanding of how an application should work and which features really matter. With an emerging body of best practices and sophisticated new tools that assist in requirement analysis and test automation, BDD has become a hot, mainstream practice.

BDD in Action teaches you BDD principles and practices and shows you how to integrate them into your existing development process, no matter what language you use. First, you'll apply BDD to requirements analysis so you can focus your development efforts on underlying business goals. Then, you'll discover how to automate acceptance criteria and use tests to guide and report on the development process. Along the way, you'll apply BDD principles at the coding level to write more maintainable and better documented code.

What's Inside

- BDD theory and practice
- How BDD will affect your team
- BDD for acceptance, integration, and unit testing
- Examples in Java, .NET, JavaScript, and more
- Reporting and living documentation

No prior experience with BDD is required.

John Ferguson Smart is a specialist in BDD, automated testing, and software lifecycle development optimization.

"Delivers a thorough treatment of the current state of BDD tools."
—From the Foreword by Dan North, Creator of BDD

"Learn BDD from top to bottom."
—Dror Helper, CodeValue

"The first complete step-by-step guide to BDD."
—Marc Bluemner
liquidlabs GmbH

"Many useful techniques, tools, and concepts to make you more productive."
—Karl Métivier
Facilité Informatique

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/BDDinAction

ISBN 13: 978-1-617291-65-4
ISBN 10: 1-617291-65-X
54999
9 781617 291654

MANNING $49.99 / Can $52.99 [INCLUDING eBOOK]