TEST DRIVEN

Practical TDD and Acceptance TDD for Java Developers

SAMPLE CHAPTER

LASSE KOSKELA

Test Driven

by Lasse Koskela

7\UdhYf˙&

Copyright 2006 Manning Publications

brief contents

PART 1 A TDD PRIMER 1

1 ■ The big picture 3
2 ■ Beginning TDD 43
3 ■ Refactoring in small steps 75
4 ■ Concepts and patterns for TDD 99

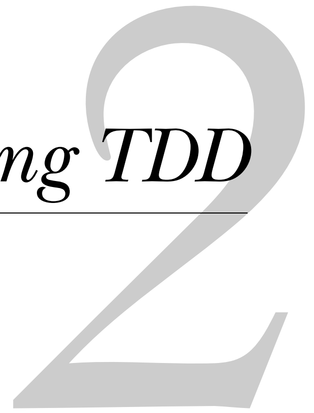
PART 2 APPLYING TDD TO SPECIFIC TECHNOLOGIES 151

5 ■ Test-driving web components 153
6 ■ Test-driving data access 195
7 ■ Test-driving the unpredictable 249
8 ■ Test-driving Swing 279

PART 3 BUILDING PRODUCTS WITH ACCEPTANCE TDD.............................. 321

9 ■ Acceptance TDD explained 323
10 ■ Creating acceptance tests with Fit 364
11 ■ Strategies for implementing acceptance tests 396
12 ■ Adopting TDD 435

appendix A ■ Brief JUnit 4 tutorial 467
appendix B ■ Brief JUnit 3.8 tutorial 470
appendix C ■ Brief EasyMock tutorial 473
appendix D ■ Running tests with Ant 475



Beginning TDD

Experience is a hard teacher because she gives the test first, the lesson afterward.

—Chinese proverb

Something big airports are generally very good at is using signs. No matter which airport I fly from, I never have to worry about not knowing which way to turn from a corner. Whenever I no longer know which way to walk, I look up and there it is: the sign that says, "Gate 42: continue straight forward."

It wasn't like that when I started writing software for a living. There was just one sign outside the terminal building saying, "Good luck." Figuring out what I should change and ending up with a correct solution was a bit like trying to find your way to the right gate at Heathrow without the hundreds of signs and displays. For software development, test-driven development with its simple rules and guidance is like those signs at the airport, making sure I end up where I need to be—and in time.

We already created a somewhat shallow idea in chapter 1 of what TDD is and talked about the forces in play, such as incremental and evolutionary design, and the "reversed" cycle of development. As we learned in chapter 1, TDD is both a design and programming technique that's essentially based on the simple rule of only writing production code to fix a failing test. TDD turns the usual design-code-test sequence of activities around into what we call the TDD cycle: test-code-refactor, illustrated in figure 2.1 and reproduced from chapter 1.

We talked about TDD being a simple and effective technique for developing higher-quality software that works and stays malleable through the continuous design we perform via frequent, merciless refactoring. Thanks to the enhanced confidence we have in our code, along with lower defect rates and a thorough test suite combined with better design, we become more productive and our work is much more enjoyable than in the times where a go-live was followed by an undetermined period of late nights and angry customers.

In this chapter, we'll develop further our understanding of what TDD is and what the secret (and not so secret) ingredients are that make it work. We'll do that by writing code test-first, learning by doing. We will be writing small tests, squeezing out the desired behavior step by step, and refactoring our code mercilessly. We're going to

Figure 2.1 The TDD cycle is a three-step process of writing a test, making it pass by writing just enough production code, and finally improving the design by disciplined refactorings.

be adding and removing code, and we're going to be changing code so that we can remove code. Along the way, we'll introduce an increasingly robust and capable implementation of a template engine, including some exception-handling functionality to accompany the "happy path" behavior we're going to begin with. We will also discuss how there are often two possible paths, *breadth first* and *depth first*, to the ultimate goal of a feature-complete system.

It'll be intensive, and there will be code listings after one another. Once we're through all that, you should have a much better understanding of how TDD works in practice and what kinds of sometimes awkward-seeming tricks we pull in order to keep moving fast.

We'll soon begin developing actual software using TDD. We're going to develop a template engine capable of rendering templates containing variables to be populated dynamically at runtime. Because the first step in TDD is writing a failing test, we need to figure out what desired behavior we'd like to test for. So, before getting our hands dirty, let's have a quick chat about how to get from abstract requirements to concrete tests.

2.1 From requirements to tests

Imagine you are implementing a subsystem for the corporate collaboration software suite[1] that's responsible for providing mail-template functionality so that the CEO's assistant can send all sorts of important, personalized emails to all personnel with a couple of mouse-clicks. How would tests drive the development of the said subsystem? You're going to know the answer to that question by the time you've finished reading this chapter; but, as with any system, it starts with decomposing the requirements into something smaller and more concrete.

2.1.1 Decomposing requirements

With any requirements you may have envisioned for the subsystem, you've probably already sliced and diced them into some kind of a set of "things we need to do"—let's call them tasks—that will, when completed, lead to satisfying the original requirement. Now erase those tasks out of your head and do it all again, this time slicing the requirements into a set of tests that, when passing, lead to the requirements being satisfied. Can you think of such tests? I'll bet you can. It might be difficult at first, but I'm confident that you've already thought of a number of little tests that would verify some tiny part of the whole.

[1] Now that's a fancy way to say "email application"!

"A mail template without any variable data is sent off as is." "The placeholder for the recipient's first name in a greeting should get replaced with each individual recipient's name, respectively." Soon, you'll have a whole bunch of those tests, which together verify more than just a tiny part of the whole. Eventually, you'll have covered most if not all of your expectations for the product's behavior and functionality.

To clarify the difference between tasks and tests, table 2.1 contrasts a few of the possible decompositions of the mail-template subsystem into tasks and tests, respectively. Decomposing into tasks (left-hand column) leads to items that do not represent progress in terms of produced software. Contrast this with the right-hand column of tests, which have a clear connection to capabilities of the produced software.

Table 2.1 Alternative decompositions of a mail-template subsystem

Mail template subsystem decomposed into a set of tasks	Mail template subsystem decomposed into a set of test(s)
Write a regular expression for identifying variables from the template.	Template without any variables renders as is.
Implement a template parser that uses the regular expression.	Template with one variable is rendered with the variable replaced with its value.
Implement a template engine that provides a public API and uses the template parser internally.	Template with multiple variables is rendered with the appropriate placeholders replaced by the associated values.
...	...

Translating requirements into tests is far superior to merely decomposing requirements into tasks because tasks such as those in table 2.1 make it is easy to lose sight of the ultimate goal—the satisfied requirement. Tasks only give us an idea of what we should *do*. Without a more concrete definition of *done*, we're left with a situation that's not that different from saying we've won the lottery simply because we purchased a lottery ticket.

We'll see more examples of decomposing requirements into tests rather than tasks later in this chapter. There are a couple of topics I'd like to discuss before that, though.

2.1.2 What are good tests made of?

So tests are generally better than tasks for guiding our work, but does it matter what kind of tests we write? Sure it does. Although there's no universal definition for what constitutes a good test, there are some guidelines and heuristics we can use to assess whether we're writing good tests. There are plenty of rules for the technical implementation of a (unit) test,[2] but from the perspective of decomposing requirements into tests, two properties of a good test can be identified as especially important:

- A good test is atomic.
- A good test is isolated.

What these properties are saying is that a good test tests a small, focused, atomic slice of the desired behavior and that the test should be isolated from other tests so that, for example, the test assumes nothing about any previously executed tests. The atomicity of a test also helps us keep our eye on the ball—a small but steady step toward the goal of a fully functional, working system.

As long as we represent our steps in the language of unambiguous tests, the mental connection between the individual steps and the ultimate destination remains much stronger and much clearer, and we're less likely to forget something.

Speaking of steps, we don't just pick tests at random until we're done. Let's talk about how we pick the next step or test to tackle.

2.1.3 Working from a test list

Armed with an initial set of tests, we can proceed to making them pass, one by one. To start with, we need to pick one of the tests—often the one we think is the easiest to make pass or one that represents the most progress with the least effort. For now, we forget that any of those other tests exist. We're completely focused on the one test we picked. We'll also come back to the question of which test to tackle next several times in the remaining chapters in part 1. After all, it's a decision we need to make every couple of minutes when test-driving with small tests.

So, what next? We'll start by writing the test code. We'll go as far as to compile and execute the test before even thinking about writing the production code our test is exercising. We can't write the test because it doesn't compile? We can't write a test for code we don't have? Oh, yes we can. Let's discuss how to do that.

---

[2] Such as "tests should be isolated and order independent," "tests should run fast," "tests shouldn't require manual setup," and so on.

2.1.4 Programming by intention

When you're writing tests before the production code they're supposed to test, you inevitably face a dilemma: how to test something that doesn't exist without breaking our test-first rule. The answer is as simple as the rule was—imagine the code you're testing exists!

How's that possible? What are we supposed to imagine? We imagine the ideal shape and form of the production code from this particular test's point of view. Isn't that cheating? Yes, it is cheating, and we love it! You see, by writing our tests with the assumption that the production code is as easy to use as we could imagine, we're making it a breeze to write the test and we're effectively making sure that our production code will become as easy to use as we were able to imagine. It has to, because our tests won't pass otherwise.

That's what I call the power of imagination! And it has a name, too. It's called *programming by intention*. Programming by intention, the concept of writing code as if another piece of code exists—even if it doesn't—is a technique that makes you focus on what we could have instead of working around what we have. Programming by intention tends to lead to code that flows better, code that's easier to understand and use—code that expresses what it does and why, instead of how it does it.

Now that we know that we should split our requirements into small, focused tests rather than tasks, and now that we understand how to progress through the list of tests by programming by intention, it's time to get busy developing a world class template engine test-first.

2.2 Choosing the first test

As promised, we're going to develop a template engine using test-driven development. In order to not take too big a bite at once, we'll restrict our focus to the business logic of the template engine and not worry about the whole system within which the template engine is being used for rendering personalized emails and what not.

TIP At this point, I highly recommend launching your favorite IDE and following along as we proceed to develop the template engine one small step at a time. Although I have done my best to write a chapter that helps you understand what we're doing and why, there's nothing more illustrative than doing it yourself!

The template engine we're talking about needs to be capable of reading in a template, which is basically static text with an arbitrary number of variable placeholders

mixed in. The variables are marked up using a specific syntax; before the template engine is asked to render the template, it needs to be given the values for the named variables.

Everything clear? Let's get going. The first thing we need to do is process the description of the template engine into an initial list of tests and then choose one of them to be implemented.

2.2.1 Creating a list of tests

Before we can come up with an initial list of tests, we need to have some requirements. For our example, let's use the following set of requirements for the mail template subsystem:

- The system replaces variable placeholders like ${firstname} and ${lastname} from a template with values provided at runtime.
- The system attempts to send a template with some variables not populated will raise an error.
- The system silently ignores values for variables that aren't found from the template.
- The system supports the full Latin-1 character set in templates.
- The system supports the full Latin-1 character set in variable values.
- And so forth...

Those are detailed requirements, aren't they? Are they tests? No. They're requirements smaller and more detailed than "implement a mail template subsystem," but they're not tests. Tests are typically much more explicit, describing the expected behavior for specific scenarios rather than articulating a generic description of that behavior.

Here's one attempt at turning the mail template subsystem's requirements into proper tests:

- Evaluating template "Hello, ${name}" with the value "Reader" for variable "name" results in the string "Hello, Reader".
- Evaluating template "${greeting}, ${name}" with values "Hi" and "Reader", respectively, results in the string "Hi, Reader".
- Evaluating template "Hello, ${name}" with no value for variable "name" raises a `MissingValueError`.

- Evaluating template “Hello, ${name}” with values “Hi” and “Reader” for variables “doesnotexist” and “name”, respectively, results in the string “Hello, Reader”.
- And so forth... (We could, for example, include some specific examples that would serve as proof that the system handles Latin-1 characters.)

See the difference? The requirements have been transformed into something that’s clearly a degree more concrete, more executable, more example-like. With these tests as our completion criteria, there’s no question of whether we’re there yet or not. With these tests, we don’t need to wonder, for example, what it means to “raise an error” and, assuming it means throwing an exception, what kind of an exception it should be and what the exception message should say. The tests tell us that we should throw a `MissingValueError` exception and that we’re not interested in the specific error message.

With this kind of test, we are finally able to produce that binary answer for the question, “am I done?” The test list itself is never done, however. The test list is a working document to which we add new tests as we go. We’re bound to be missing some tests from our initial list, and we’ll likely think of a bunch of them while we’re deep into implementing some other test. At that time, we write those tests down and continue with what we’re working on.

We’ve got a list of tests that tell us exactly when the requirements have been fulfilled. Next, we start working through the list, making them pass one by one.

2.2.2 Writing the first failing test

Let’s take another example and practice our programming by intention a bit before continuing with our journey to test-driven development. Why don’t we take the first test from the list and see how we might go about writing the test before the code?

Here’s our test again:

Evaluating template “Hello, ${name}” with the value “Reader” for variable “name” results in the string “Hello, Reader”.

NOTE At this point, we’re pulling out the IDE to write some code. We’ll be using JUnit 4, the latest version of the de facto unit-testing framework for Java today (http://www.junit.org). If you’re not already familiar with JUnit, take a peek at appendix A for a short introduction. You won’t need to do that quite yet, though, if you don’t feel like it. The JUnit stuff is not essential here—the purpose is to see programming by intention in action—so just focus on the code we’re writing inside the methods and pay no attention to the weird annotations we’re using.

Because we're going to implement the test first, let's start by giving a name for our test class, in listing 2.1.

Listing 2.1 Creating a skeleton for our tests

```
public class TestTemplate {
}
```

Next, we need a test method. Let's call it `oneVariable`, as shown in listing 2.2.

Listing 2.2 Adding a test method

```
import org.junit.Test;
public class TestTemplate {
    @Test
    public void oneVariable() throws Exception {
    }
}
```

We're making progress. Can you feel the wind? No? Well, that's all right, because we know that small steps are good for us. Besides, we're just about to get into the interesting part—programming stuff by intention. The code in listing 2.3 is what we might feel like writing for the test, *assuming* that the implementation is there (even though it isn't), exercising our freedom to design the template feature in a way that makes it easy to use.

Listing 2.3 Writing the actual test

```
import org.junit.Test;
import static org.junit.Assert.*;

public class TestTemplate {
    @Test
    public void oneVariable() throws Exception {
        Template template = new Template("Hello, ${name}");
        template.set("name", "Reader");
        assertEquals("Hello, Reader", template.evaluate());
    }
}
```

In the test method in listing 2.3, we first create a `Template` object by passing the template text as a constructor argument. We then set a value for the variable "name" and finally invoke a method named `evaluate`, asserting
the resulting output matches our expectation.

How does that feel? Is that the way you'd like the template system to work? That's the way I felt like it should work. I'm a bit undecided about the name of `set` because `put` would be consistent with the ubiquitous `java.util.Map` interface. I do think `set` sounds better to my ear than `put`, however, so let's leave it like that for now. Let's proceed and see how we can make the compiler happy with our imagination-powered snippet of code.

Making the compiler happy

Now, the compiler is eager to remind us that, regardless of our intentions, the class `Template` does not exist. We'll have to add that. Next, the compiler points out that there's no such constructor for `Template` that takes a `String` as a parameter. It's also anxious to point out that the methods `set` and `evaluate` don't exist either, so we'll have to add those as well.[3] We end up with the following skeleton of a class, in listing 2.4.

Listing 2.4 Satisfying the compiler by adding empty methods and constructors

```
public class Template {

    public Template(String templateText) {
    }

    public void set(String variable, String value) {
    }

    public String evaluate() {
        return null;
    }
}
```

Finally, the code compiles. What's next? Next, we run the test, of course.

[3] Modern IDEs provide shortcuts for generating missing methods in a breeze, making our test-driven life that much easier.

Running the test

When we run our freshly written test, it fails—not surprisingly, because none of the methods we added are doing anything. (You didn't implement the methods already, did you?)

I like to run the test at this point and see it fail before I proceed to making the test pass. Why? Because then I know that when I'm clicking that Run button in my IDE, the test I'm working on is indeed executed, and I'm not looking at the result of running some other test. That would be confusing. I know. I've done that. And not just once or twice. Plus, seeing the test fail tells me that the functionality I'm testing for doesn't exist yet—an unexpected green bar means that an assumption I have made about the code base doesn't hold!

A failing test is progress

If you're thinking we haven't even gotten started yet—after all, we haven't implemented a single line of code for the behavior we're testing for—consider the following: What we have now is a test that tells us when we're done with that particular task. Not a moment too soon, not a moment too late. It won't try to tell us something like "you're 90% done" or "just five more minutes." We'll know when the test passes; that is, when the code does what we expect it to do.

I wouldn't be surprised if you're thinking I've got a marble or two missing. I'm pretty confident that you'll figure out on your own exactly how significant a change this way of looking at things is for you, personally—we're not all alike, and certain techniques help some more than others. Oh, and we'll be writing a lot more code test-first during the first half of the book, so there will be plenty of opportunities to evaluate how well the technique works in practice.

Running the test, we get the output in figure 2.2, complaining about getting a `null` when it expected the string "Hello, Reader".

We're in the red phase of the TDD cycle, which means that we have written and executed a test that is failing—and our IDE indicates that by showing a red progress bar. We've now written a test, programming by intention, and we have a skeleton of the production code that our test is testing. At this point, all that's left is to implement the `Template` class so that our test passes and we can rest our eyes on a green progress bar.

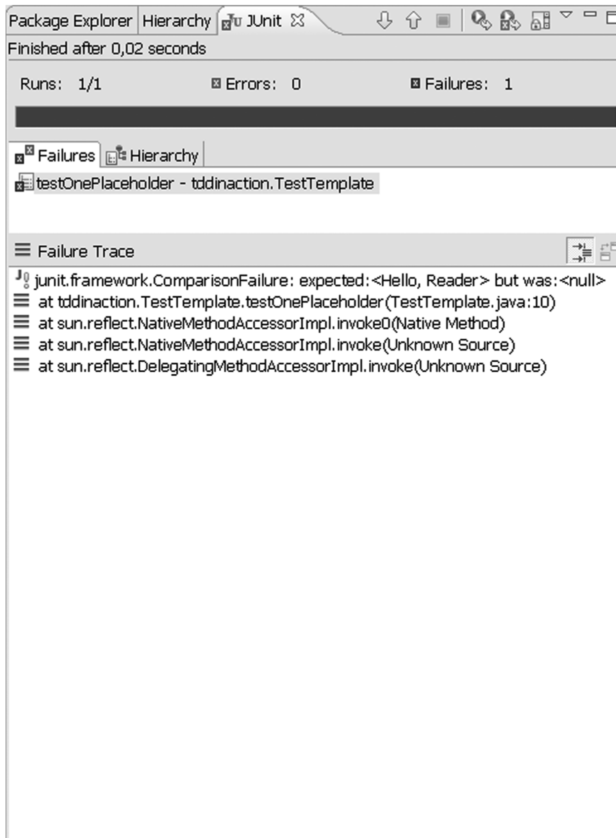


Figure 2.2
Our first failing test. What a precious moment. What we see here is the Eclipse IDE's JUnit runner. But all major IDEs have something similar to a red bar, some kind of hierarchical list of tests run, and access to the detailed error—the stack trace for failed tests.

2.2.3 Making the first test pass

We don't have much code yet, but we've already made a number of significant design decisions. We've decided that there should be a class `Template` that loads a template text given as an argument to the constructor, lets us set a value for a named placeholder, and can evaluate itself, producing the wanted output. What's next?

We had already written the skeleton for the `Template` class, shown in listing 2.5, so that our test compiles.

Listing 2.5 Stubbing out methods of the `Template` class to make compilation succeed

```
public class Template {

    public Template(String templateText) {
    }

    public void set(String variable, String value) {
    }

    public String evaluate() {
        return null;
    }
}
```

All the constructors and methods are in place to make the compiler happy, but none of those constructors and methods is doing anything so far. We've written a single failing test to show us a path forward, we've added just enough production code to make the test compile, and the next thing to do is the often most brain-tickling part of creating the functionality our test is asking of us.

There's more to making the test pass than just that, though. We want to make the test pass in the easiest, quickest way possible. To put it another way, we're now facing a red bar, meaning the code in our workspace is currently in an unstable state. We want to get out of that state and into stable ground *as quickly as possible.*

Here's an interesting decision. How do we make the test pass as quickly and easily as possible? Evaluating a template that simple, "Hello, ${name}", with a string-replace operation would be a matter of a few minutes of work, probably. There's another implementation, however, of the functionality implied by our failing test that fits our goal of "as quickly as possible" a bit better. Listing 2.6 shows us what that could look like.

Listing 2.6 Passing as quickly as possible with a hard-coded return statement

```
public class Template {

    public Template(String templateText) {
    }

    public void set(String variable, String value) {
    }

    public String evaluate() {
        return "Hello, Reader";    <-- It couldn't get much simpler than this
    }
}
```

To save space, from now on we'll omit the two `import` statements from the test listings, but you'll still need them to access the JUnit library.

Yes, I am serious. Hard-coding the `evaluate` method to return "Hello, Reader" is most certainly the quickest and easiest way to make the test pass. Although it may not make much sense at first, it's good to push ourselves a bit to squeeze out the wanted behavior with our tests.

In this case, we're making use of neither the actual variable nor the template. And that means we know at least two dimensions on which to push our code toward a proper implementation. Let's extend our test to squeeze out the implementation we're looking for.

2.2.4 *Writing another test*

Let's say we want to first drive out hard-coding on the part of the variable's value. Listing 2.7 presents one way to do this by extending the existing test through another assertion.

Listing 2.7 Forcing out the hard-coded return statement with another test

```
public class TestTemplate {
    @Test
    public void oneVariable() throws Exception {
        Template template = new Template("Hello, ${name}");
        template.set("name", "Reader");
        assertEquals("Hello, Reader", template.evaluate());
    }

    @Test
    public void differentValue() throws Exception {
        Template template = new Template("Hello, ${name}");
        template.set("name", "someone else");
        assertEquals("Hello, someone else", template.evaluate());
    }
}
```

Triangulate with a different value.

We've added a second call to `set`—with different input—and a second assertion to verify that the `Template` object will re-evaluate the template with the latest value for the "name" variable. Our hard-coded `evaluate` method in the `Template` class will surely no longer pass this test, which was our goal.

This technique is aptly named *triangulation*, referring to how we're using multiple bearings to pinpoint the implementation toward the proper implementation.

We could call it *playing difficult*, but it really can help in avoiding premature optimization, feature creep, and over-engineering in general.

Now, how could we make this enhanced test pass? Do you see a way around having to parse the actual template? I guess we could hard-code some more and push the need for the real implementation a bit farther. Let's see where that would take us, as illustrated by listing 2.8.

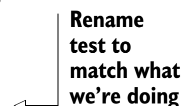
Listing 2.8 Making the second test pass by storing and returning the set value

```
public class Template {

    private String variableValue;

    public Template(String templateText) {
    }

    public void set(String variable, String value) {
        this.variableValue = value;
    }

    public String evaluate() {
        return "Hello, " + variableValue;
    }
}
```

Our test passes again with minimal effort. Obviously our test isn't good enough yet, because we've got that hard-coded part in there, so let's continue triangulating to push out the last bits of literal strings from our code. Listing 2.9 shows how we alter our test to drive out not just the hard-coding of the variable value but also the template text around the variable.

Listing 2.9 Applying triangulation for the static template text

```
public class TestTemplate {
    @Test
    public void oneVariable() throws Exception {
        Template template = new Template("Hello, ${name}");
        template.set("name", "Reader");
        assertEquals("Hello, Reader", template.evaluate());
    }

    @Test
    public void differentTemplate() throws Exception {
        Template template = new Template("Hi, ${name}");
        template.set("name", "someone else");
```



```
        assertEquals("Hi, someone else", template.evaluate());
    }
}
```

Squeeze out more hard coding

Red bar. Obviously our hard-coded return statement doesn't cut it anymore. At this point, we're facing the task of parsing the template text somehow. Perhaps we should first implement the parsing logic and then come back to this test. What do you think?

I'm thinking it's time to discuss breadth and depth.

2.3 Breadth-first, depth-first

The software we build today is not something trivial that can be built in a matter of a couple of hours. I've worked with systems with tens of thousands of lines of code. One system had over a million lines of code. And I know there are plenty of people out there who have developed and maintained systems that have had many more millions of lines of code than the one I had the privilege of being involved with. Just like it'll get messy if we try to swallow a muffin as a whole, taking too big a bite talking about code will surely backfire as well.

In our template engine example, we just reached a point where we uncovered a muffin-size bit of complexity that we haven't yet taken care of. At the same time, the test we were working on—evaluating a template—cannot work without that particular bit of complexity being tackled first. Or can it?

As you may remember from the algorithms course in school, there is more than one way to traverse a tree. We can either traverse breadth-first or we can traverse depth-first. The same holds true for test-driven development. Well, sort of. Figures 2.3 and 2.4 compare these two alternatives side by side.

We could decide to go breadth-first, writing tests against the public interface of the `Template` class; making them pass by faking the internals, the lower-level functionality; and only then proceed to writing tests for (and implementing) the layer beneath the original one. Figure 2.3 illustrates this approach.

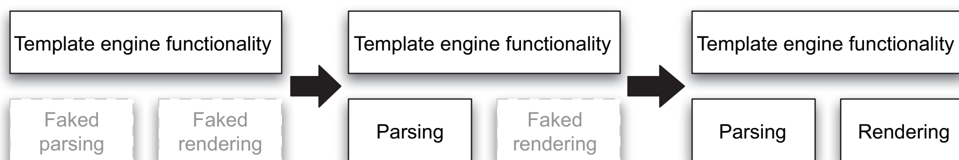


Figure 2.3 With a breadth-first strategy, we implement the higher-level functionality first by faking the required lower-level functionality.

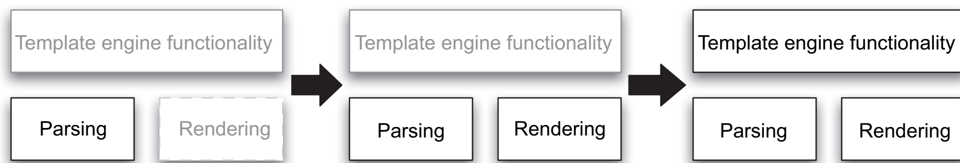


Figure 2.4 With a depth-first strategy, we implement the lower-level functionality first and only compose the higher-level functionality once all the ingredients are present.

The other option would be to back out to a working state and start writing tests for the template-parsing logic instead—and come back once we've got the template-parsing stuff in check. That is, traverse functionality depth-first and implement the details first for one small vertical slice before moving on with the next slice. Figure 2.4 shows what this approach might look like.

In the case of our template engine having to deal with "Hello, ${name}", we can fake the lower-level functionality—parsing the template—a little longer before we add more functionality (more tests, that is) and see where that'll lead us.

2.3.1 Faking details a little longer

First, we'll need to start storing the variable value and the template text somewhere. We'll also need to make `evaluate` replace the placeholder with the value. Listing 2.10 shows one way to get our test passing.

Listing 2.10 Our first attempt at handling the variable for real

```
public class Template {

    private String variableValue;

    private String templateText;

    public Template(String templateText) {
        this.templateText = templateText;
    }

    public void set(String variable, String value) {
        this.variableValue = value;
    }

    public String evaluate() {
        return templateText.replaceAll("\\$\\{name\\}", variableValue);
    }
}
```

Some might say what we're doing here is cheating—we're still hard-coding the regular expression to look for "${name}". It's not really cheating, however. We're being disciplined and not giving in to the temptation of "just write the code and be done with it." We'll get there eventually. Baby steps, remember? All tests are passing, we're on stable ground—with a green bar as evidence—and we know exactly where that ground is. What's next?

Refactoring is what's next. Do you see anything to refactor? Spot any duplication that's bothering you? Any less-than-ideal code constructs? Nothing strikes my eye, so I'll go on and proceed to enhancing our test. Specifically, I'd like to drive out that hard-coded variable name. What better way to do that than by introducing multiple variables into the template?

2.3.2 Squeezing out the fake stuff

It's time to write a test for multiple variables on a template. This, by the way, is test number 2 on our test list. We're proceeding almost at light-speed, aren't we? Adding the following snippet to our test class should give us more to think about:

```
@Test
public void multipleVariables() throws Exception {
    Template template = new Template("${one}, ${two}, ${three}");
    template.set("one", "1");
    template.set("two", "2");
    template.set("three", "3");
    assertEquals("1, 2, 3", template.evaluate());
}
```

This test fails right now, telling us that `evaluate` returns the template text as is instead of "1, 2, 3" (which is hardly a surprise because our regular expression is looking for just "${name}"). The big question now is how to deal with this added level of variability in the template.

One way that comes to mind—to get the test passing as quickly as possible—would be to do the search-and-replace operation for each of the variable name-value pairs the `Template` knows about. That could be slow, and there could be some issues with variable values containing substrings that look like a "${variable}", but that should be good enough for us right now. Usually, when I think of worries or deficiencies like the one with the regular expression approach we're talking about here, I write them down as new tests on the list. Let's do that right now:

Evaluate template "${one}, ${two}, ${three}" with values "1", "${foo}", and "3", respectively, and verify that the template engine renders the result as "1, ${foo}, 3".

When we find that, for example, something is wrongly implemented or missing altogether while we're working on one test, it's usually a good idea to write down a short reminder and continue with what we're working on. This let's us focus on one task at a time instead of overloading our brains by jumping back and forth between multiple things (and likely making a mess of our code base in the process).

Now, let's see how we could apply the search-and-replace approach to pass our current failing test.

New and improved details

Back to business. The search-and-replace-based implementation shown in listing 2.11 does indeed get us back to green.

Listing 2.11 Search-and-replace for handling multiple variables

```
import java.util.Map;
import java.util.HashMap;
import static java.util.Map.Entry;

public class Template {

    private Map<String, String> variables;
    private String templateText;

    public Template(String templateText) {
        this.variables = new HashMap<String, String>();
        this.templateText = templateText;
    }

    public void set(String name, String value) {
        this.variables.put(name, value);
    }

    public String evaluate() {
        String result = templateText;
        for (Entry<String, String> entry : variables.entrySet()) {
            String regex = "\\$\\{" + entry.getKey() + "\\}";
            result = result.replaceAll(regex, entry.getValue());
        }
        return result;
    }
}
```

Store variable values in HashMap (private Map<String, String> variables;)

Store variable values in HashMap (this.variables = new HashMap<String, String>();)

Store variable values in HashMap (set method)

Loop through variables

Replace each variable with its value

The tests are running green again, which is cool. What's next? Refactoring! I still don't see much to refactor, however, so let's skip that step again and move on to the next test.

We've got the following tests still untouched on our list:

- Evaluating template "Hello, ${name}" with no value for variable "name" raises a `MissingValueError`.
- Evaluating template "Hello, ${name}" with values "Hi" and "Reader" for variables "doesnotexist" and "name", respectively, results in the string "Hello, Reader".
- Evaluate template "${one}, ${two}, ${three}" with values "1", "${foo}", and "3", respectively, and verify that the template engine renders the result as "1, ${foo}, 3".

I'm thinking the second of these seems like it should pass already. Let's see whether that's the case.

Testing for a special case

The following snippet introduces a new test with which we assert that if we set variables that don't exist in the template text, the variables are ignored by the `Template` class. I have a theory that our current implementation should pass this test with flying colors without any changes:

```
@Test
public void unknownVariablesAreIgnored() throws Exception {
    Template template = new Template("Hello, ${name}");
    template.set("name", "Reader");
    template.set("doesnotexist", "Hi");
    assertEquals("Hello, Reader", template.evaluate());
}
```

Yep. It passes right away. Or does it?

My IDE runs unit tests so fast that every now and then I'm not 100% sure whether I actually ran the tests or not. This is why you might want to consider insisting on seeing red before green. In other words, fail the test intentionally at first, just to see that our test execution catches the failure—that we're really executing the newly added test—and only then proceed to implement the test and see the bar turn green again.

TIP Most modern IDEs let us customize the code templates used for code generation. I have configured the "test" template in my Eclipse, for example, to include a statement to throw a `RuntimeException` with the message *Not implemented*. This statement gets added into every test method I generate by typing *test* and pressing Ctrl+Space, and it serves not only as a time-saver but also as a reminder for me to run the failing test first.

In this case, the code really does pass the new test with no changes. Wouldn't it be wonderful if all development were this easy? Perhaps not, but it's nice to have a freebie every now and then, isn't it?

Moving on, it's time for the refactoring step again.

2.4 Let's not forget to refactor

We haven't yet refactored anything, even though we've written a lot of code. The reason is that we haven't spotted anything that needs refactoring. Is there still nothing to refactor? Probably not, because we didn't add any code, right? Wrong. Even though we didn't add any production code, we added test code, and that's code just like any other—and we know better than to let our test code rot and get us into serious trouble later on.

Let's see what we could do about our test code by first identifying any potential refactorings we might want to do and then deciding which of them we'll carry out. Listing 2.12 presents our test class so far.

Listing 2.12 Our test code up to this point—can you spot anything to refactor?

```
public class TestTemplate {

    @Test
    public void oneVariable() throws Exception {
        Template template = new Template("Hello, ${name}");
        template.set("name", "Reader");
        assertEquals("Hello, Reader", template.evaluate());
    }

    @Test
    public void differentTemplate() throws Exception {
        template = new Template("Hi, ${name}");
        template.set("name", "someone else");
        assertEquals("Hi, someone else", template.evaluate());
    }

    @Test
    public void multipleVariables() throws Exception {
        Template template = new Template("${one}, ${two}, ${three}");
        template.set("one", "1");
```

```
        template.set("two", "2");
        template.set("three", "3");
        assertEquals("1, 2, 3", template.evaluate());
    }

    @Test
    public void unknownVariablesAreIgnored() throws Exception {
        Template template = new Template("Hello, ${name}");
        template.set("name", "Reader");
        template.set("doesnotexist", "Hi");
        assertEquals("Hello, Reader", template.evaluate());
    }
}
```

Take a moment to think about what we could improve in the test code. Duplication. Semantic redundancy. Anything that jumps out. Anything we perhaps should clean up. We're next going to take a look at a couple of potential refactorings that I've spotted, but I urge you to see what kind of smells you find from the code we have so far. We'll continue when you're ready to compare notes.

2.4.1 *Potential refactorings in test code*

There are at least a couple of things that come to mind when scanning through the code. First, all of our tests are using a `Template` object, so we probably should extract that into an instance variable rather than declare it over and over again. Second, we're calling the `evaluate` method several times as an argument to `assertEquals`. Third, we're instantiating the `Template` class with the same template text in two places. That's duplication and probably should be removed somehow.

However, one of the tests is using a different template text. Should we use two instance variables, one for each template text? If so, we probably should split the class into two to deal with the two fixtures we clearly have.

> **NOTE** If you're not familiar with the concept of fixtures, they're basically the set of objects—the state—we've initialized for our test to use. With JUnit, the fixture is effectively the set of instance variables and the other configured state of a test class instance. We'll talk more about fixtures in chapter 4, but for now just think of the shared starting state between different test methods of a test class.

There is an alternative, however, to splitting the class with `TestTemplate`. Let's see what that alternative refactoring is.

2.4.2 Removing a redundant test

There's a more fundamental type of duplication present in our test code between `oneVariable`, `differentTemplate`, and `multipleVariables`. Thinking about it, the latter test basically covers the first one as well, so we can safely get rid of the single-variable version if we also add a new test to verify that we can set a new value for a variable and re-evaluate the template. Furthermore, we can make `unknownVariablesAreIgnored` use the same template text as `multipleVariables`. And I'm not so sure we need `differentTemplate` either, so maybe that should go, too.

Let's see what the refactored test code shown in listing 2.13 looks like.

Listing 2.13 Test code after removing a redundant test and unifying the fixture

```
import org.junit.Test;
import org.junit.Before;
import static org.junit.Assert.*;

public class TestTemplate {

    private Template template;

    @Before
    public void setUp() throws Exception {
        template = new Template("${one}, ${two}, ${three}");
        template.set("one", "1");
        template.set("two", "2");
        template.set("three", "3");
    }

    @Test
    public void multipleVariables() throws Exception {
        assertTemplateEvaluatesTo("1, 2, 3");
    }

    @Test
    public void unknownVariablesAreIgnored() throws Exception {
        template.set("doesnotexist", "whatever");
        assertTemplateEvaluatesTo("1, 2, 3");
    }

    private void assertTemplateEvaluatesTo(String expected) {
        assertEquals(expected, template.evaluate());
    }
}
```

A common fixture for all tests

Simple, focused test

Simple, focused test

As you can see, we were able to mold our tests toward using a single template text and common setup,[4] leaving the test methods themselves delightfully trivial and focusing only on the essential—the specific aspect of functionality they're testing.

But now, time for more functionality—that is, another test. I'm thinking it might be a good time to look into error handling now that we have the basic functionality for a template engine in place.

2.5 Adding a bit of error handling

It's again time to add a new test. The only one remaining on our list is the one about raising an error when evaluating a template with some variable left without a value. You guessed right, we're going to proceed by first writing a test, then making it pass, and refactoring it to the best design we can think of.

Now, let's see how we'd like the template engine to work in the case of missing values, shall we?

2.5.1 Expecting an exception

How do we write a JUnit test that verifies that an exception is thrown? With the `try-catch` construct, of course, except that this time, the code throwing an exception is a good thing—the expected behavior. The approach shown in listing 2.14 is a common pattern[5] for testing exception-throwing behavior with JUnit.

Listing 2.14 Testing for an exception

```
@Test
public void missingValueRaisesException() throws Exception {
    try {
        new Template("${foo}").evaluate();
        fail("evaluate() should throw an exception if "
                + "a variable was left without a value!");
    } catch (MissingValueException expected) {
    }
}
```

Note the call to the `fail` method right after `evaluate`. With that call to `org.junit.Assert#fail`, we're basically saying, "if we got this far, something went wrong" and the `fail` method fails the test. If, however, the call to `evaluate` throws

[4] The `@Before` annotated method is invoked by JUnit before each `@Test` method.

[5] See Recipe 2.8 in *JUnit Recipes* by J. B. Rainsberger (Manning Publications, 2005).

Testing for exceptions with an annotation

JUnit 4 brought us a handy annotation-based syntax for expressing exception tests such as our `missingValueRaisesException` in listing 2.14. Using the annotation syntax, the same test could be written as follows:

```
@Test(expected=MissingValueException.class)
public void testMissingValueRaisesException() throws Exception {
    new Template("${foo}").evaluate();
}
```

Although this annotation-based version of our test is less verbose than our `try-catch`, with the `try-catch` we can also make further assertions about the exception thrown (for example, that the error message contains a key piece of information). Some folks like to use the annotation syntax where they can, while others prefer always using the same `try-catch` pattern. Personally, I use the annotation shorthand when I'm only interested in the type and use the `try-catch` when I feel like digging deeper.

an exception, we either catch it and ignore it—if it's of the expected type—or let it bubble up to JUnit, which will then mark the test as having had an error.

OK. We've got a test that's failing. Well, at first it's not even compiling, but adding an empty `MissingValueException` class makes that go away:

```
public class MissingValueException extends RuntimeException {
    // this is all we need for now
}
```

We have the red bar again, and we're ready to make the test pass. And that means we have to somehow check for missing variables.

Once again, we're faced with the question of how to get to the green as quickly as possible. How do we know, inside `evaluate`, whether some of the variables specified in the template text are without a value? The simplest solution that springs to my mind right now is to look at the output and figure out whether it still contains pieces that look like variables. This is by no means a robust solution, but it'll do for now and it should be pretty easy to implement.

Sure enough, it was easy—as you can see from listing 2.15.

Listing 2.15 Checking for remaining variables after the search-and-replace

```
public String evaluate() {
    String result = templateText;
    for (Entry<String, String> entry : variables.entrySet()) {
        String regex = "\\$\\{" + entry.getKey() + "\\}";
```

```
        result = result.replaceAll(regex, entry.getValue());
    }
    if (result.matches(".*\\$\\{.+\\}.*")) {
        throw new MissingValueException();
    }
    return result;
}
```

Does it look like we left a variable in there?

We're back from red to green again. See anything to refactor this time? Nothing critical, I think, but the `evaluate` method is starting to become rather big. "Big?" you say. "That tiny piece of code?" I think so. I think it's time to refactor.

2.5.2 Refactoring toward smaller methods

There are differences in what size people prefer for their methods, classes, and so forth. Myself? I'm most comfortable with fewer than 10 lines of Java code per method.[6] Anything beyond that, and I start to wonder if there's something I could throw away or move somewhere else. Also, `evaluate` is on the verge of doing too many different things—replacing variables with values and checking for missing values. I'm starting to think it's already over the edge.

Let's apply some refactoring magic on `evaluate` to make it cleaner. I'm thinking we could at least extract the check for missing values into its own method. Listing 2.16 shows the refactored version of our `evaluate` method.

Listing 2.16 Extracting the check for missing variables into a method

```
public String evaluate() {
    String result = templateText;
    for (Entry<String, String> entry : variables.entrySet()) {
        String regex = "\\$\\{" + entry.getKey() + "\\}";
        result = result.replaceAll(regex, entry.getValue());
    }
    checkForMissingValues(result);
    return result;
}

private void checkForMissingValues(String result) {
    if (result.matches(".*\\$\\{.+\\}.*")) {
        throw new MissingValueException();
    }
}
```

Hooray! We got rid of a whole if-block from evaluate()

[6] A perhaps better indicator of too big a method would be a complexity metric such as McCabe. Long methods are pretty good indicators of too high complexity as well, however.

Much better already, but there's still more to do. As we extracted the check for missing values from the `evaluate` method, we introduced a mismatch in the level of abstraction present in the `evaluate` method. We made `evaluate` less balanced than we'd like our methods to be. Let's talk about balance for a minute before moving on with our implementation.

2.5.3 Keeping methods in balance

One property of a method that has quite an effect on the readability of our code is the consistency of the abstraction level of the code within that method. Let's take another look at `evaluate` as an example of what we're talking about:

```
public String evaluate() {
    String result = templateText;
    for (Entry<String, String> entry : variables.entrySet()) {
        String regex = "\\$\\{" + entry.getKey() + "\\}";
        result = result.replaceAll(regex, entry.getValue());
    }
    checkForMissingValues(result);
    return result;
}
```

The `evaluate` method is doing two things: replacing variables with values and checking for missing values, which are at a completely different level of abstraction. The `for` loop is clearly more involved than the method call to `checkfor-MissingValues`. It's often easy to add little pieces of functionality by gluing a one-liner to an existing method, but without keeping an eye on inconsistencies in abstraction levels, the code is soon an unintelligible mess.

Fortunately, these kinds of issues usually don't require anything more than a simple application of the *extract method* refactoring, illustrated by listing 2.17.

Listing 2.17 Extracting another method from `evaluate()`

```
public String evaluate() {
    String result = replaceVariables();
    checkForMissingValues(result);
    return result;
}

private String replaceVariables() {
    String result = templateText;
    for (Entry<String, String> entry : variables.entrySet()) {
        String regex = "\\$\\{" + entry.getKey() + "\\}";
        result = result.replaceAll(regex, entry.getValue());
    }
    return result;
}
```

evaluate() method's internals much better balanced

New method is simple and has single, clear purpose

```
private void checkForMissingValues(String result) {
    if (result.matches(".*\\$\\{.+\\}.*")) {
        throw new MissingValueException();
    }
}
```

Running our tests, we realize that we didn't break anything with the refactoring, so we're good to go. Doing these kinds of edits on working code would be a much bigger effort if we didn't have the test battery watching our back. Let's face it: we might not have even carried out the refactoring we just did if we hadn't had those tests.

Our tests are running green again, which is a good thing. We're not done with the exception case yet, however. There's still one piece of functionality that we want to have for handling missing values: a meaningful exception message.

2.5.4 *Expecting details from an exception*

While writing the test for the missing value, we just caught a `MissingValueException` and called it a day. You can't possibly know how many times I've stared at a meaningless exception message[7] and cursed the developer who didn't bother giving even the least bit of information to help in problem solving. In this case, we probably should embed the name of the variable missing a value into the exception message. And, since we're all set, why not do just that (see listing 2.18)?

Listing 2.18 Testing for an expected exception

```
@Test
public void missingValueRaisesException() throws Exception {
    try {
        new Template("${foo}").evaluate();
        fail("evaluate() should throw an exception if "
                + "a variable was left without a value!");
    } catch (MissingValueException expected) {
        assertEquals("No value for ${foo}",
                     expected.getMessage());
    }
}
```

Exception should name missing variable

As usual, the edit needed for making a test pass is a matter of a couple of minutes. This time, we need to use the `java.util.regex` API a bit differently in order to

[7] Does "java.lang.NullPointerException: null" ring a bell...?

dig out the part of the rendered result that matches a variable pattern. Perhaps a snippet of code would explain that better:

```
import java.util.regex.Pattern;
import java.util.regex.Matcher;
private void checkForMissingValues(String result) {
    Matcher m = Pattern.compile("\\$\\{.+\\}").matcher(result);
    if (m.find()) {
        throw new MissingValueException("No value for " + m.group());
    }
}
```

Of course, we'll also need to add a constructor to `MissingValueException`:

```
public class MissingValueException extends RuntimeException  {
    public MissingValueException(String message) {
        super(message);
    }
}
```

That's it. A couple of minutes ago, we thought of a new test and decided to implement it right away. If it would've seemed like a bigger task, we might have just added it as a new test to our test list and come back to it later. Speaking of the test list, I think it's time for an update.

2.6 Loose ends on the test list

We've now implemented all the test cases that we thought of in the beginning. We did, however, spot some issues with our current implementation along the way. For one, it doesn't handle situations where the variable values contain delimiters such as "${" and "}". I've also got some worries with regard to how well our template engine performs. Adding performance to our task list, these are the remaining tests:

- Evaluate template "${one}, ${two}, ${three}" with values "1", "${foo}", and "3", respectively, and verify that the template engine renders the result as "1, ${foo}, 3".
- Verify that a template of 100 words and 20 variables with values of approximately 15 characters each is evaluated in 200 milliseconds or less.

We should add tests to the list as soon as we think of them, just to be sure we don't forget. That doesn't mean, however, that we should immediately drop on all fours and start chasing cars—we don't want to let these new tests interrupt our flow. We write down a simple reminder and continue with what we're currently working with, knowing that we'll come back to that new test later.

In our case, "later" is actually "as soon as I stop babbling," so let's get to it right away. How about going after that performance check first and then returning to the double-rendering issue?

2.6.1 Testing for performance

Wanting to get an idea of whether our template engine is any good performance-wise, let's quickly add a test for the performance of `evaluate` to see whether we're close to a reasonable execution time. Listing 2.19 presents our new test class for such a micro-performance check.

Listing 2.19 Writing a simple performance check to act as an early warning system

```
import org.junit.Test;
import static org.junit.Assert.*;

public class TestTemplatePerformance {

    // Omitted the setUp() for creating a 100-word template
    // with 20 variables and populating it with approximately
    // 15-character values

    @Test
    public void templateWith100WordsAnd20Variables() throws Exception {
        long expected = 200L;
        long time = System.currentTimeMillis();
        template.evaluate();
        time = System.currentTimeMillis() - time;
        assertTrue("Rendering the template took " + time
                + "ms while the target was " + expected + "ms",
                time <= expected);
    }
}
```

It seems that our `Template` implementation does pass this performance check for now, taking approximately 100 milliseconds to render the 100-word template with some 20 variables. Nice to know that we're not already spending over our budget! Besides, now that we've got the test in place, we'll know immediately when we do something that makes the template engine slower than we're expecting it to be.

This is not an approach without any issues, of course. Because our code is not executing in the computer equivalent of a vacuum, the test—which depends on the elapsed system time—is non-deterministic. In other words, the test might pass on one machine but fail on another. It might even alternate between passing and failing on a single machine based on what other software happens to be running at the same time. This is something we'll have to deal with, unless we're willing to

accept the fact that some of our test runs might fail seemingly sporadically. Most of the time, it's a good enough solution to tune the test such that it won't fail too often while still alerting us to a performance problem.

With that worry largely behind us, let's see about that other test we added for the double-rendering problem—the one about rendering variables that have values that look like variable placeholders.

2.6.2 A looming design dead-end

Regarding the remaining test for variable values that contain "${" and "}", things are starting to look more difficult. For one thing, we can't just do a search-and-replace over and over again until we've covered all variable values set to the template, because some of the variable values rendered first could be re-rendered with something completely different during later rounds of search-and-replace. In addition, we can't rely on our method of detecting unset variables by looking for "${…}" in the result.

Before we go further, let's stop all this speculation and write a test that proves whether our assumptions about the code's current behavior are correct! Adding the following test into the `TestTemplate` class does just that:

```
@Test
public void variablesGetProcessedJustOnce() throws Exception {
    template.set("one", "${one}");
    template.set("two", "${three}");
    template.set("three", "${two}");
    assertTemplateEvaluatesTo("${one}, ${three}, ${two}");
}
```

Running the test tells us that we certainly have a problem. This test is causing an `IllegalArgumentException` to be thrown from the regular expression code invoked from `evaluate`, saying something about an "illegal group reference," so our code is definitely not handling the scenario well. I'm thinking it's time to back out of that test, pick up the notepad, and sketch a bit. Instead, let's sum up the chapter so far, take a break, and continue with this double-rendering issue in the next chapter.

2.7 Summary

Test-driven development is a powerful technique that helps us write better software faster. It does so by focusing on what is absolutely needed right now, then making that tiny piece work, and finally cleaning up any mess we might've made while making it work, effectively keeping the code base healthy. This cycle of

first writing a test, then writing the code to make it pass, and finally refactoring the design makes heavy use of programming by intention—writing the test as if the ideal implementation exists already—as a tool for creating usable and testable designs.

In this chapter, we have seen TDD in action, we've lived through the TDD cycle, and we've realized that our current design for the template engine doesn't quite cut it. We set out to write a template engine based on a short list of tests that specify the expected behavior of the engine, and we followed the test-code-refactor (or red-green-green) cycle all the way.[8] The code already satisfies most of our requirements (we've got tests to prove it!) and would certainly be useful in many contexts as it is. We are able to make progress fast with the tests watching our backs, and we're not afraid to refactor knowing that the safety net will catch us should we fail to retain functionality as it is.

Now, let's flip the page to the next chapter and see how we can overcome the remaining issues and wrap up that template engine into a fully functional, beautifully constructed piece of product!

---

[8] I'll forgive you if you slipped. I do that, too. And it tends to come back to bite me, reminding of the various benefits of TDD. I'm sure you'll also get better and better at doing TDD as time goes by.

JAVA

TEST DRIVEN

Lasse Koskela

In test-driven development, you first write an executable test of what your application code must do. Only then do you write the code itself and, with the test spurring you on, improve your design. In acceptance test-driven development (ATDD), you use the same technique to implement product features, benefiting from iterative development, rapid feedback cycles, and better-defined requirements. TDD and its supporting tools and techniques lead to better software faster.

Test Driven brings under one cover practical TDD techniques distilled from several years of community experience. With examples in Java and the Java EE environment, it explores both the techniques and the mindset of TDD and ATDD. It uses carefully chosen examples to illustrate TDD tools and design patterns, not in the abstract but concretely in the context of the technologies you face at work. It is accessible to TDD beginners, and it offers effective and less-well-known techniques to older TDD hands.

What's Inside

- Hands-on examples of how to test-drive Java code
- How to avoid common TDD adoption pitfalls
- ATDD development and the Fit framework
- How to test Java EE components—Servlets, JSPs, and Spring Controllers
- How to handle tough issues like multithreaded programs and data-access code

Lasse Koskela, a methodology specialist at Reaktor Innovations in Finland, has coached dozens of teams in agile methods and practices such as test-driven development.

For more information, code samples, and to purchase an ebook visit www.manning.com/TestDriven

"... very engaging writing style. I was blown away!"

—Michael Feathers
Consultant, Object Mentor

"Full of hard-won lessons that take years to learn on your own."

—Laurent Bossavit
Consultant, 2006 Gordon Pask Award Winner

"Strengthen your quality safety-net with the TDD ideas in this book!"

—Christopher Haupt
Principal Consultant
Mobirobo LLC

"... a well-spring of up-to-date information and practices."

—Jason Rogers
Software Engineer
RiskMetrics Group, Inc.

"... it's much better on TDD than other books I've read ... including a wonderful job in providing TDD philosophy."

—Dave Corun
Architect, Social Solutions

ISBN-13: 978-1932394856
ISBN-10: 1932394850

manning

$44.99 / Can $53.99