Extracted from:

Pragmatic Unit Testing in Java 8 with JUnit

This PDF file contains pages extracted from *Pragmatic Unit Testingin Java 8 with JUnit*, published by the Pragmatic Bookshelf. For more information or to purchase a paperback or PDF copy, please visit http://www.pragprog.com.

Note: This extract contains some colored text (particularly in code listing). This is available only in online versions of the books. The printed versions are black and white. Pagination might vary between the online and printed versions; the content is otherwise identical.

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

Pragmatic Unit Testing in Java 8 with JUnit

Jeff Langr

with Andy Hunt
& Dave Thomas

edited by
Susannah Davidson Pfalzer

Pragmatic Unit Testing in Java 8 with JUnit

Jeff Langr

with Andy Hunt
Dave Thomas

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at *https://pragprog.com*.

The team that produced this book includes:

Susannah Davidson Pfalzer (editor)
Potomac Indexing, LLC (indexer)
Eileen Cohen (copyeditor)
Dave Thomas (typesetter)
Janet Furlow (producer)
Ellie Callahan (support)

For international rights, please contact *rights@pragprog.com*.

Printed in the United States of America.
ISBN-13: 978-1-94122-259-1
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—March 2015

Our systems are bloated! You can pick almost any system at random and spot obvious bits of rampant duplication—whether it's a hundred-line-long method that's almost a complete replication from another class or a few lines of utility code repeated megaumpteen times throughout. The cost of such duplication is significant: every piece of code duplicated increases the cost to maintain it, as well as the risk in making a change. You want to minimize the amount of duplication in your system's code.

The cost of understanding code is also significant. A change requiring ten minutes of effort in clear, well-structured code can require hours of effort in convoluted, muddy code. You want to maximize the clarity in your system's code.

You can accomplish both goals—low duplication and high clarity—at a reasonable cost and with a wonderful return on investment. The good news is that having unit tests can help you reach the goals. In this chapter you'll learn how to *refactor* your code with these ideals in mind.

A Little Bit o' Refactor

If you've recently arrived from Proxima Centauri in a slow warp drive that required fifteen years of travel time, you might not have heard the term *refactoring*. Otherwise, you at least recognize it from the menus in your IDE. You might even be aware that refactoring your code means you're transforming its underlying structure while retaining its existing functional behavior.

In other words, refactoring is moving code bits around and making sure the system still works. Willy-nilly restructuring of code sounds risky! By gosh, you really want to make sure you have appropriate protection when doing so. You know…tests.

An Opportunity for Refactoring

Let's revisit the iloveyouboss code. You wrote a couple of tests with us for it back in Chapter 2, *Getting Real with JUnit*, on page ?. As a reminder, here's the core `matches()` method from the `Profile` class:

iloveyouboss/16/src/iloveyouboss/Profile.java

```
public boolean matches(Criteria criteria) {
   score = 0;

   boolean kill = false;
   boolean anyMatches = false;
   for (Criterion criterion: criteria) {
      Answer answer = answers.get(
            criterion.getAnswer().getQuestionText());
```

```
        boolean match =
              criterion.getWeight() == Weight.DontCare ||
              answer.match(criterion.getAnswer());
        if (!match && criterion.getWeight() == Weight.MustMatch) {
           kill = true;
        }
        if (match) {
           score += criterion.getWeight().getValue();
        }
        anyMatches |= match;
     }
     if (kill)
        return false;
     return anyMatches;
  }
```

The method isn't particularly long, weighing in at around a dozen total lines of expressions and/or statements. Yet it's reasonably dense, embodying quite a bit of logic. We were able to add five more test cases behind the scenes.

Extract Method: Your Second-Best Refactoring Friend

(Okay, we'll kill the mystery before you go digging in the index…. Your *best* refactoring friend is *rename*, whether it be a class, method, or variable of any sort. Clarity is largely about declaration of intent, and good names are what impart clarity best in code.)

Our goal: reduce complexity in the matches() method so that we can readily understand what it's responsible for—its *policy*. We do that in part by *extracting* detailed bits of logic to new, separate methods.

Conditional expressions often read poorly, particularly when they are complex. An example is the assignment to match that appears in the for loop in matches():

iloveyouboss/16/src/iloveyouboss/Profile.java

```
for (Criterion criterion: criteria) {
   Answer answer = answers.get(
         criterion.getAnswer().getQuestionText());
➤  boolean match =
➤        criterion.getWeight() == Weight.DontCare ||
➤        answer.match(criterion.getAnswer());
   // ...
}
```

Isolate the complexity of the assignment by extracting it to a separate method. You're left with a simple declaration in the loop: the match variable represents whether or not the criterion matches the answer:

iloveyouboss/17/src/iloveyouboss/Profile.java

```
public boolean matches(Criteria criteria) {
   score = 0;

   boolean kill = false;
   boolean anyMatches = false;
   for (Criterion criterion: criteria) {
      Answer answer = answers.get(
            criterion.getAnswer().getQuestionText());
➤     boolean match = matches(criterion, answer);

      if (!match && criterion.getWeight() == Weight.MustMatch) {
         kill = true;
      }
      if (match) {
         score += criterion.getWeight().getValue();
      }
      anyMatches |= match;
   }
   if (kill)
      return false;
   return anyMatches;
}

➤ private boolean matches(Criterion criterion, Answer answer) {
➤    return criterion.getWeight() == Weight.DontCare ||
➤          answer.match(criterion.getAnswer());
➤ }
```

If you need to know the details of *how* a criterion matches an answer, you can navigate into the newly extracted matches() method. Extracting lower-level details removes distracting clutter if you need only understand the high-level policy for how a Profile matches against a Criteria object.

It's way too easy to break functionality when moving code about. You need the confidence to know that you can change code and not introduce sneaky little defects that aren't discovered until much later.

Fortunately, the tests written for Profile (see Chapter 2, *Getting Real with JUnit*, on page ?) begin to provide you with the confidence you need. With each small change, you run your fast set of tests—it's cheap, easy, and fun.

The ability to move code about safely is one of the most important benefits of unit testing. It allows you to add new features safely, and it also allows you to make changes that keep the design in good shape. In the absence of sufficient tests, you'll tend to make fewer changes. Or you'll make changes that are highly risky.

Finding Better Homes for Our Methods

Our loop is a bit easier to read—great! But we note that the newly extracted code in matches() doesn't have anything to do with the Profile object itself. It seems that either the Answer class or the Criterion class could be responsible for determining when one matches another.

Move the newly extracted matches() method to the Criterion class. Criterion objects already know about Answer objects, but the converse is not true—Answer is not dependent on Criterion. If you were to move matches() to Answer, you'd have a bidirectional dependency. Not cool.

Here's matches() in its new home:

iloveyouboss/18/src/iloveyouboss/Criterion.java

```
public class Criterion implements Scoreable {
   // ...
   public boolean matches(Answer answer) {
      return getWeight() == Weight.DontCare ||
             answer.match(getAnswer());
   }
}
```

And here's what the loop looks like after the move:

iloveyouboss/18/src/iloveyouboss/Profile.java

```
for (Criterion criterion: criteria) {
   Answer answer = answers.get(
         criterion.getAnswer().getQuestionText());
➤  boolean match = criterion.matches(answer);

   if (!match && criterion.getWeight() == Weight.MustMatch) {
      kill = true;
   }
   if (match) {
      score += criterion.getWeight().getValue();
   }
   anyMatches |= match;
}
```

The statement that assigns into the answer local variable is quite a mouthful:

iloveyouboss/18/src/iloveyouboss/Profile.java

```
Answer answer = answers.get(
      criterion.getAnswer().getQuestionText());
```

It suffers for violating the Law of Demeter (which roughly says to avoid chaining together method calls that ripple through other objects), and it's simply not clear.

A first step toward improving things is to extract the right-hand-side expression of the answer assignment to a new method whose name, answerMatching(), better explains what's going on:

iloveyouboss/19/src/iloveyouboss/Profile.java

```
public boolean matches(Criteria criteria) {
   score = 0;

   boolean kill = false;
   boolean anyMatches = false;
   for (Criterion criterion: criteria) {
➤     Answer answer = answerMatching(criterion);
      boolean match = criterion.matches(answer);

      if (!match && criterion.getWeight() == Weight.MustMatch) {
         kill = true;
      }
      if (match) {
         score += criterion.getWeight().getValue();
      }
      anyMatches |= match;
   }
   if (kill)
      return false;
   return anyMatches;
}

➤ private Answer answerMatching(Criterion criterion) {
➤    return answers.get(criterion.getAnswer().getQuestionText());
➤ }
```

Temporary variables have a number of uses. You might be more accustomed to temporaries that cache the value of an expensive computation or collect things that change throughout the body of a method. The answer temporary variable does neither, but another use of a temporary variable is to clarify the intent of code—a valid choice even if the temporary is used only once.