The
Pragmatic
Programmers

Pragmatic Unit Testing
in Java with JUnit
Third Edition

Jeff Langr
edited by Kelly Talbot

This extract shows the online version of this title, and may contain features (such as hyperlinks and colors) that are not available in the print version.

For more information, or to purchase a paperback or ebook copy, please visit https://www.pragprog.com.

Amplifying the Core Intent of Code

Let's re-examine the slightly-improved matches method:

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

    var kill = false;
    var anyMatches = false;
    for (var criterion: criteria) {
        var match = criterion.isMatch(profileAnswerMatching(criterion));
        if (!match && criterion.weight() == REQUIRED) {
            kill = true;
        }
        if (match) {
            score += criterion.weight().value();
        }
        anyMatches |= match;
    }
    if (kill)
        return false;

    return anyMatches;
}
```

Careful reading reveals the following outcomes:

- Return true if any criterion matches, false if none do.
- Calculate the score by summing the weights of matching criteria.
- Return false when any required criterion does not match the corresponding profile answer.

Let's restructure matches to directly emphasize these three core concepts.

Extract Concept: Any Matches Exist?

The determination of whether any matches exist is scattered through the method. It involves both the anyMatches and matches local variables:

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

    var kill = false;
➤   var anyMatches = false;
➤   for (var criterion: criteria) {
➤       var match = criterion.isMatch(profileAnswerMatching(criterion));
        if (!match && criterion.weight() == REQUIRED) {
            kill = true;
        }
        if (match) {
```

```
            score += criterion.weight().value();
        }
➤       anyMatches |= match;
    }
    if (kill)
        return false;
➤   return anyMatches;
}
```

Your goal: Move all the logic related to making that determination to its own method. Steps:

1. Change the return statement to return the result of calling a new method, anyMatches().
2. Create a new method, anyMatches, that returns a boolean value.
3. *Copy* (don't cut) the relevant logic into the new method.

The result:

```
utj3-refactor/09/src/main/java/iloveyouboss/Profile.java
public boolean matches(Criteria criteria) {
    score = 0;

    var kill = false;
    var anyMatches = false;
    for (var criterion: criteria) {
        var match = criterion.isMatch(profileAnswerMatching(criterion));
        if (!match && criterion.weight() == REQUIRED) {
            kill = true;
        }
        if (match) {
            score += criterion.weight().value();
        }
        anyMatches |= match;
    }
    if (kill)
        return false;
➤   return anyMatches(criteria);
}
➤ private boolean anyMatches(Criteria criteria) {
    var anyMatches = false;
    for (var criterion: criteria) {
        var match = criterion.isMatch(profileAnswerMatching(criterion));
        anyMatches |= match;
    }
    return anyMatches;
}
```

There's no automated refactoring for this change. You're making riskier manual changes, so run your tests! Once they pass, remove the two lines of code in matches that reference the anyMatches variable:

utj3-refactor/10/src/main/java/iloveyouboss/Profile.java
```java
public boolean matches(Criteria criteria) {
    score = 0;

    var kill = false;
    for (var criterion: criteria) {
        var match = criterion.isMatch(profileAnswerMatching(criterion));
        if (!match && criterion.weight() == REQUIRED) {
            kill = true;
        }
        if (match) {
            score += criterion.weight().value();
        }
    }
    if (kill)
        return false;

    return anyMatches(criteria);
}
```

The loop of course must remain, and so must the line of code that assigns to the match variable.

You might be concerned about that method extraction and its performance implications. We'll discuss.

Extract Concept: Calculate Score for Matches

Now that you've isolated the anyMatches logic by extracting it to a new method, you can do the same for the code that calculates the score. If you put the call to calculateScore below if (kill) return false, however, the tests break. (The score needs to be calculated before any unmet required criterion results in an aborted method.)

utj3-refactor/11/src/main/java/iloveyouboss/Profile.java
```java
public boolean matches(Criteria criteria) {
➤   calculateScore(criteria);

    var kill = false;
    for (var criterion: criteria) {
        var match = criterion.isMatch(profileAnswerMatching(criterion));
        if (!match && criterion.weight() == REQUIRED) {
            kill = true;
        }
    }
    if (kill)
        return false;
```

```
        return anyMatches(criteria);
    }
➤   private void calculateScore(Criteria criteria) {
        score = 0;
        for (var criterion: criteria) {
            var match = criterion.isMatch(profileAnswerMatching(criterion));
            if (match) {
                score += criterion.weight().value();
            }
        }
    }
```

Hmmm. You might be wondering if you're creating performance problems.

Extract Concept: Return False When Required Criterion Not Met

The code remaining in the loop aborts method execution if the profile doesn't match a required criterion, returning false. Similarly extract this logic to a new method, anyRequiredCriteriaNotMet:

utj3-refactor/12/src/main/java/iloveyouboss/Profile.java
```
public boolean matches(Criteria criteria) {
    calculateScore(criteria);

➤   var kill = anyRequiredCriteriaNotMet(criteria);
    if (kill)
        return false;

    return anyMatches(criteria);
}
➤ private boolean anyRequiredCriteriaNotMet(Criteria criteria) {
    var kill = false;
    for (var criterion: criteria) {
        var match = criterion.isMatch(profileAnswerMatching(criterion));
        if (!match && criterion.weight() == REQUIRED) {
            kill = true;
        }
    }
    return kill;
}
```

Matches is now five lines of code and fairly easy to follow! But some cleanup work remains, particularly in the three newly-extracted methods. For one, the loops are all old-school for-each loops. You'll clean up these problems after we address the performance elephant in the room.

The implementation for matches now involves three loops, spread across three methods, instead of a single loop through the criteria. That might seem hor-

rifying to you. We'll come back to discuss the performance implications; for now, let's talk about the benefits you gain with the new design.

Earlier, you invested some time in carefully reading the original code in order to glean its three intents. The boolean logic throughout created opportunities for confusion along the way. Now, matches (almost) directly states the method's high-level goals.

The implementation details for each of the three steps in the algorithm are hidden in the corresponding helper methods calculateScore, anyRequiredCriteriaNotMet, and anyMatches. Each helper method allows the necessary behavior to be expressed in a concise, isolated fashion, not cluttered with other concerns.