we recognize that it has meaty parts that concern us most.

Covering One Path

The bulk of the "interesting" logic in matches() resides in the body of the for loop. Let's write a simple test that covers one path through the loop.

Two points that glancing at the code should make obvious: we need a Profile instance, and we need a Criteria object to pass as an argument to matches().

By analyzing the code in matches() and looking at the constructors for Criteria, Criterion, and Question, we figure out how to piece together a useful Criteria object.

The analysis lets you write this part of the *arrange* portion of the test:

iloveyouboss/6/test/iloveyouboss/ProfileTest.java

```java
@Test
public void test() {
   Profile profile = new Profile("Bull Hockey, Inc.");
   Question question = new BooleanQuestion(1, "Got bonuses?");
   Criteria criteria = new Criteria();
   Answer criteriaAnswer = new Answer(question, Bool.TRUE);
   Criterion criterion = new Criterion(criteriaAnswer, Weight.MustMatch);
   criteria.add(criterion);
}
```

(From here on out, we're expecting you to code along with us. We won't be as explicit. If you see a new code snippet, figure that you'll need to make some changes on your end.)

Paraphrased in brief: after creating a profile, create a question (*Got bonuses?* They'd better!). The next three lines are responsible for putting together a Criterion, which is an answer plus a weighting of the significance of that answer. The answer, in turn, is a question and the desired value (Bool.TRUE) for the answer to that question. Finally, the criterion is added to a Criteria object.

(In case you're wondering, the Bool class is a wrapper around an enum that has values 0 and 1. We don't claim that the code we're testing is good code.)

In matches(), for each Criterion object iterated over in the for loop, the code retrieves the corresponding Answer object in the answers HashMap (line 29). That means you must add an appropriate Answer to the Profile object:

iloveyouboss/7/test/iloveyouboss/ProfileTest.java

```java
@Test
public void test() {
   Profile profile = new Profile("Bull Hockey, Inc.");
   Question question = new BooleanQuestion(1, "Got bonuses?");
   Answer profileAnswer = new Answer(question, Bool.FALSE);
   profile.add(profileAnswer);
   Criteria criteria = new Criteria();
   Answer criteriaAnswer = new Answer(question, Bool.TRUE);
   Criterion criterion = new Criterion(criteriaAnswer, Weight.MustMatch);
   criteria.add(criterion);
}
```

We finish up the test by acting and asserting. We also change its name to aptly describe the scenario it demonstrates:

iloveyouboss/8/test/iloveyouboss/ProfileTest.java

```java
@Test
public void matchAnswersFalseWhenMustMatchCriteriaNotMet() {
   Profile profile = new Profile("Bull Hockey, Inc.");
   Question question = new BooleanQuestion(1, "Got bonuses?");
   Answer profileAnswer = new Answer(question, Bool.FALSE);
   profile.add(profileAnswer);
   Criteria criteria = new Criteria();
   Answer criteriaAnswer = new Answer(question, Bool.TRUE);
   Criterion criterion = new Criterion(criteriaAnswer, Weight.MustMatch);
   criteria.add(criterion);

   boolean matches = profile.matches(criteria);
   assertFalse(matches);
}
```

We were able to piece together a test based on our knowledge of the matches method, verifying that one pathway through the code works as (apparently) intended. If neither of us knew much about the code, we would have had to spend a bit more time carefully reading through the code to understand what it does, building up more and more of a real test as we went.

Think about maintaining our test. It's ten lines of code, which doesn't seem like much. But if we write tests to cover all fifteen conditions described previously, it seems like it could get out of hand. Fifteen tests times ten lines each sounds like a lot to maintain for a target method of fewer than twenty lines.

From a cognitive standpoint, the ten lines require careful reading, particularly for someone else who knows nothing about the code we wrote.

Tackling a Second Test

Let's write a second test to see if our concerns are warranted. Taking a look at the assignment to the match local variable (starting at line 30 in Profile.java), it appears that match gets set to true when the criterion weight is DontCare. Code in the remainder of the method suggests that matches() should return true if a sole criterion sets match to true.

Each unit test in JUnit requires its own context: JUnit doesn't run tests in any easily determinable order, so we can't have one test depend on the results of another. Further, JUnit creates a new instance of the ProfileTest class for each of its two test methods.

We must then make sure our second test, matchAnswersTrueForAnyDontCareCriteria, similarly creates a Profile object, a Question object, and so on:

iloveyouboss/9/test/iloveyouboss/ProfileTest.java

```
@Test
```

We must then make sure our second test, matchAnswersTrueForAnyDontCareCriteria, similarly creates a Profile object, a Question object, and so on:

iloveyouboss/9/test/iloveyouboss/ProfileTest.java

```
@Test
public void matchAnswersTrueForAnyDontCareCriteria() {
   Profile profile = new Profile("Bull Hockey, Inc.");
   Question question = new BooleanQuestion(1, "Got milk?");
   Answer profileAnswer = new Answer(question, Bool.FALSE);
   profile.add(profileAnswer);
   Criteria criteria = new Criteria();
   Answer criteriaAnswer = new Answer(question, Bool.TRUE);
➤  Criterion criterion = new Criterion(criteriaAnswer, Weight.DontCare);
   criteria.add(criterion);

   boolean matches = profile.matches(criteria);
➤  assertTrue(matches);
}
```

The second test looks darn similar to matchAnswersFalseWhenMustMatchCriteriaNotMet. In fact, the two highlighted lines are the only real difference between the two tests. Maybe we can reduce the 150 potential lines of test code by eliminating some of the redundancy across tests. Let's do a bit of *refactoring*.

Initializing Tests with @Before Methods

The first thing to look at is common *initialization* code in all (both) of the tests in ProfileTest. If both tests have such duplicate logic, move it into an @Before method. For each test JUnit runs, it first executes code in any methods marked with the @Before annotation.

The tests in ProfileTest each require the existence of an initialized Profile object and a new Question object. Move that initialization to an @Before method named create() (or bozo() if you want to irritate your teammates—the name is arbitrary).

iloveyouboss/10/test/iloveyouboss/ProfileTest.java

```
public class ProfileTest {
➤  private Profile profile;
➤  private BooleanQuestion question;
➤  private Criteria criteria;
➤
➤  @Before
➤  public void create() {
➤     profile = new Profile("Bull Hockey, Inc.");
➤     question = new BooleanQuestion(1, "Got bonuses?");
➤     criteria = new Criteria();
➤  }

   @Test
   public void matchAnswersFalseWhenMustMatchCriteriaNotMet() {
      Answer profileAnswer = new Answer(question, Bool.FALSE);
      profile.add(profileAnswer);
      Answer criteriaAnswer = new Answer(question, Bool.TRUE);
      Criterion criterion = new Criterion(criteriaAnswer, Weight.MustMatch);
      criteria.add(criterion);

      boolean matches = profile.matches(criteria);

      assertFalse(matches);
   }

   @Test
   public void matchAnswersTrueForAnyDontCareCriteria() {
      Answer profileAnswer = new Answer(question, Bool.FALSE);
      profile.add(profileAnswer);
      Answer criteriaAnswer = new Answer(question, Bool.TRUE);
      Criterion criterion = new Criterion(criteriaAnswer, Weight.DontCare);
      criteria.add(criterion);

      boolean matches = profile.matches(criteria);

      assertTrue(matches);
   }
}
```

```
}
```

The initialization lines moved to @Before disappear from each of the two tests, making them a little bit easier to read.

Imagining that JUnit chooses to run matchAnswersTrueForAnyDontCareCriteria first, here's the sequence of events:

1. JUnit creates a new instance of ProfileTest, which includes the uninitialized profile, question, and criteria fields.
2. JUnit calls the @Before method, which initializes each of profile, question, and criteria to appropriate instances.

3. JUnit calls matchAnswersTrueForAnyDontCareCriteria, executing each of its statements and marking the test as passed or failed.
4. JUnit creates a new instance of ProfileTest, because it has another test to process.
5. JUnit calls the @Before method for the new instance, which again initializes fields.
6. JUnit calls the other test, matchAnswersFalseWhenMustMatchCriteriaNotMet.

If you don't believe that JUnit creates a new instance for each test it runs, crank up your debugger or drop in a few System.out.println calls. JUnit works that way to force the issue of independent unit tests. If both ProfileTest tests ran in the same instance, you'd have to worry about cleaning up the state of the shared Profile object.

You want to minimize the impact any one test has on another (which means you also want to avoid `static` fields in your test classes). Imagine you have several thousand unit tests, with numerous interdependencies among tests. If test *xyz* fails, your effort to determine why increases dramatically, because you must now look at all the tests that run before *xyz*.

Our tests read a bit better now, but let's make another pass at cleaning them up. We *inline* some local variables, creating a more condensed yet slightly more readable arrange portion of each test:

iloveyouboss/11/test/iloveyouboss/ProfileTest.java

```java
@Test
public void matchAnswersFalseWhenMustMatchCriteriaNotMet() {
   profile.add(new Answer(question, Bool.FALSE));
   criteria.add(
         new Criterion(new Answer(question, Bool.TRUE), Weight.MustMatch));

   boolean matches = profile.matches(criteria);

   assertFalse(matches);
}

@Test
public void matchAnswersTrueForAnyDontCareCriteria() {
   profile.add(new Answer(question, Bool.FALSE));
   criteria.add(
         new Criterion(new Answer(question, Bool.TRUE), Weight.DontCare));

   boolean matches = profile.matches(criteria);

   assertTrue(matches);
}
```