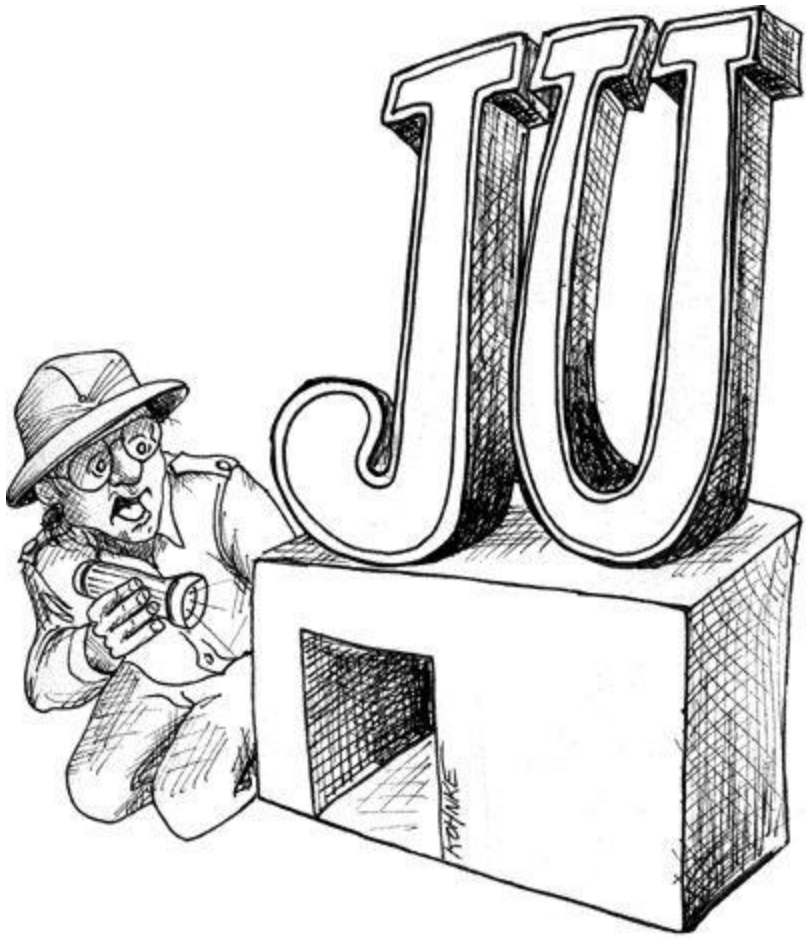
15 JUnit Internals



JUnit is one of the most famous of all Java frameworks. As frameworks go, it is simple in conception, precise in definition, and elegant in implementation. But what does the code look like? In this chapter we'll critique an example drawn from the JUnit framework.

The JUnit Framework

JUnit has had many authors, but it began with Kent Beck and Eric Gamma together on a plane to Atlanta. Kent wanted to learn Java, and Eric wanted to learn about Kent's Smalltalk testing framework. "What could be more natural to a couple of geeks in cramped quarters than to pull out our

laptops and start coding?"[1] After three hours of high-altitude work, they had written the basics of JUnit.

The module we'll look at is the clever bit of code that helps identify string comparison errors. This module is called `ComparisonCompactor`. Given two strings that differ, such as `ABCDE` and `ABXDE`, it will expose the difference by generating a string such as `<…B[X]D…>`.

I could explain it further, but the test cases do a better job. So take a look at Listing 15-1 and you will understand the requirements of this module in depth. While you are at it, critique the structure of the tests. Could they be simpler or more obvious?

Listing 15-1 `ComparisonCompactorTest.java`

```
package junit.tests.framework;
import junit.framework.ComparisonCompactor;
import junit.framework.TestCase;

public class ComparisonCompactorTest extends TestCase {

  public void testMessage() {
    String failure= new ComparisonCompactor(0, "b", "c").compact("a");
    assertTrue("a expected:<[b]> but was:<[c]>".equals(failure));
  }

  public void testStartSame() {
    String failure= new ComparisonCompactor(1, "ba", "bc").compact(null);
    assertEquals("expected:<b[a]> but was:<b[c]>", failure);
  }

  public void testEndSame() {
    String failure= new ComparisonCompactor(1, "ab", "cb").compact(null);
    assertEquals("expected:<[a]b> but was:<[c]b>", failure);
  }

  public void testSame() {
```

```
                String failure= new ComparisonCompactor(1, “ab”, “ab”).compact(null);
      assertEquals(“expected:<ab> but was:<ab>”, failure);
    }

    public void testNoContextStartAndEndSame() {
                String failure= new ComparisonCompactor(0, “abc”, “adc”).compact(null);
      assertEquals(“expected:<…[b]…> but was:<…[d]…>”, failure);
    }
    public void testStartAndEndContext() {
                String failure= new ComparisonCompactor(1, “abc”, “adc”).compact(null);
      assertEquals(“expected:<a[b]c> but was:<a[d]c>”, failure);
    }

    public void testStartAndEndContextWithEllipses() {
      String failure=
   new ComparisonCompactor(1, “abcde”, “abfde”).compact(null);
      assertEquals(“expected:<…b[c]d…> but was:<…b[f]d…>”, failure);
    }

    public void testComparisonErrorStartSameComplete() {
                String failure= new ComparisonCompactor(2, “ab”, “abc”).compact(null);
      assertEquals(“expected:<ab[]> but was:<ab[c]>”, failure);
    }

    public void testComparisonErrorEndSameComplete() {
                String failure= new ComparisonCompactor(0, “bc”, “abc”).compact(null);
      assertEquals(“expected:<[]…> but was:<[a]…>”, failure);
    }

    public void testComparisonErrorEndSameCompleteContext() {
                String failure= new ComparisonCompactor(2, “bc”, “abc”).compact(null);
```

```
    assertEquals("expected:<[]bc> but was:<[a]bc>", failure);
  }

  public void testComparisonErrorOverlapingMatches() {
      String failure= new ComparisonCompactor(0, "abc",
"abbc").compact(null);
    assertEquals("expected:<…[]…> but was:<…[b]…>", failure);
 }

  public void testComparisonErrorOverlapingMatchesContext() {
      String failure= new ComparisonCompactor(2, "abc",
"abbc").compact(null);
    assertEquals("expected:<ab[]c> but was:<ab[b]c>", failure);
  }

  public void testComparisonErrorOverlapingMatches2() {
    String failure= new ComparisonCompactor(0, "abcdde",
"abcde").compact(null);
    assertEquals("expected:<…[d]…> but was:<…[]…>", failure);
  }
  public void testComparisonErrorOverlapingMatches2Context() {
    String failure=
   new ComparisonCompactor(2, "abcdde", "abcde").compact(null);
    assertEquals("expected:<…cd[d]e> but was:<…cd[]e>", failure);
  }

  public void testComparisonErrorWithActualNull() {
      String failure= new ComparisonCompactor(0, "a",
null).compact(null);
    assertEquals("expected:<a> but was:<null>", failure);
  }

  public void testComparisonErrorWithActualNullContext() {
      String failure= new ComparisonCompactor(2, "a",
null).compact(null);
    assertEquals("expected:<a> but was:<null>", failure);
  }
```

```
  public void testComparisonErrorWithExpectedNull() {
            String failure= new ComparisonCompactor(0, null,
"a").compact(null);
    assertEquals("expected:<null> but was:<a>", failure);
  }

  public void testComparisonErrorWithExpectedNullContext() {
            String failure= new ComparisonCompactor(2, null,
"a").compact(null);
    assertEquals("expected:<null> but was:<a>", failure);
  }
  public void testBug609972() {
          String failure= new ComparisonCompactor(10, "S&P500",
"0").compact(null);
    assertEquals("expected:<[S&P50]0> but was:<[]0>", failure);
  }
 }
```

I ran a code coverage analysis on the `ComparisonCompactor` using these tests. The code is 100 percent covered. Every line of code, every `if` statement and `for` loop, is executed by the tests. This gives me a high degree of confidence that the code works and a high degree of respect for the craftsmanship of the authors.

The code for `ComparisonCompactor` is in Listing 15-2. Take a moment to look over this code. I think you'll find it to be nicely partitioned, reasonably expressive, and simple in structure. Once you are done, then we'll pick the nits together.

Listing 15-2 `ComparisonCompactor.java (Original)`

```
package junit.framework;

public class ComparisonCompactor {

  private static final String ELLIPSIS = "…";
  private static final String DELTA_END = "]";
  private static final String DELTA_START = "[";
```

```
    private int fContextLength;
    private String fExpected;
    private String fActual;
    private int fPrefix;
    private int fSuffix;

    public ComparisonCompactor(int contextLength,
                               String expected,
                                 String actual) {
      fContextLength = contextLength;
      fExpected = expected;
      fActual = actual;
    }

    public String compact(String message) {
      if (fExpected == null || fActual == null || areStringsEqual())
        return Assert.format(message, fExpected, fActual);

      findCommonPrefix();
      findCommonSuffix();
      String expected = compactString(fExpected);
      String actual = compactString(fActual);
      return Assert.format(message, expected, actual);
    }

    private String compactString(String source) {
      String result = DELTA_START +
                      source.substring(fPrefix, source.length() - fSuffix + 1) +
DELTA_END;
      if (fPrefix > 0)
        result = computeCommonPrefix() + result;
      if (fSuffix > 0)
        result = result + computeCommonSuffix();
      return result;
    }
    private void findCommonPrefix() {
```

```
    fPrefix = 0;
    int end = Math.min(fExpected.length(), fActual.length());
    for (; fPrefix < end; fPrefix++) {
      if (fExpected.charAt(fPrefix) != fActual.charAt(fPrefix))
        break;
    }
  }

  private void findCommonSuffix() {
    int expectedSuffix = fExpected.length() - 1;
    int actualSuffix = fActual.length() - 1;
    for (;
        actualSuffix >= fPrefix && expectedSuffix >= fPrefix;
         actualSuffix--, expectedSuffix--) {
      if (fExpected.charAt(expectedSuffix) != fActual.charAt(actualSuffix))
        break;
    }
    fSuffix = fExpected.length() - expectedSuffix;
  }

  private String computeCommonPrefix() {
    return (fPrefix > fContextLength ? ELLIPSIS : "") +
           fExpected.substring(Math.max(0, fPrefix - fContextLength),
                               fPrefix);
  }
  private String computeCommonSuffix() {
    int end = Math.min(fExpected.length() - fSuffix + 1 + fContextLength,
                       fExpected.length());
    return fExpected.substring(fExpected.length() - fSuffix + 1, end) +
                       (fExpected.length() - fSuffix + 1 < fExpected.length()
-  fContextLength ? ELLIPSIS : "");

  }

  private boolean areStringsEqual() {
    return fExpected.equals(fActual);
```

```
  }
}
```

You might have a few complaints about this module. There are some long expressions and some strange `+1`s and so forth. But overall this module is pretty good. After all, it might have looked like Listing 15-3.

Listing 15-3 `ComparisonCompator.java (defactored)`

```
  package junit.framework;
public class  ComparisonCompactor {
  private int ctxt;
  private String s1;
  private String s2;
  private int pfx;
  private int sfx;
  public ComparisonCompactor(int ctxt, String s1, String s2) {
    this.ctxt = ctxt;
    this.s1 = s1;
    this.s2 = s2;
  }

  public String compact(String msg) {
    if (s1 == null || s2 == null || s1.equals(s2))
      return Assert.format(msg, s1, s2);

    pfx = 0;
    for (; pfx < Math.min(s1.length(), s2.length()); pfx++) {
      if (s1.charAt(pfx) != s2.charAt(pfx))
        break;
    }
    int sfx1 = s1.length() - 1;
    int sfx2 = s2.length() - 1;
    for (; sfx2 >= pfx && sfx1 >= pfx; sfx2--, sfx1--) {
      if (s1.charAt(sfx1) != s2.charAt(sfx2))
      break;
  }
    sfx = s1.length() - sfx1;
```

```
    String cmp1 = compactString(s1);
    String cmp2 = compactString(s2);
    return Assert.format(msg, cmp1, cmp2);
  }
  private String compactString(String s) {
    String result =
     "[" + s.substring(pfx, s.length() - sfx + 1) + "]";
    if (pfx > 0)
      result = (pfx > ctxt ? "..." : "") +
        s1.substring(Math.max(0, pfx - ctxt), pfx) + result;
    if (sfx > 0) {
      int end = Math.min(s1.length() - sfx + 1 + ctxt, s1.length());
      result = result + (s1.substring(s1.length() - sfx + 1, end) +
        (s1.length() - sfx + 1 < s1.length() - ctxt ? "..." : ""));
    }
    return result;
  }
}
```

Even though the authors left this module in very good shape, the *Boy Scout Rule*[2] tells us we should leave it cleaner than we found it. So, how can we improve on the original code in Listing 15-2?

The first thing I don't care for is the `f` prefix for the member variables [N6]. Today's environments make this kind of scope encoding redundant. So let's eliminate all the `f`'s.

```
    private int contextLength;
  private String expected;
  private String actual;
  private int prefix;
  private int suffix;
```

Next, we have an unencapsulated conditional at the beginning of the `compact` function [G28].

```
    public String compact(String message) {
    if (expected == null || actual == null || areStringsEqual())
      return Assert.format(message, expected, actual);
    findCommonPrefix();
```

```
    findCommonSuffix();
    String expected = compactString(this.expected);
    String actual = compactString(this.actual);
    return Assert.format(message, expected, actual);
  }
```

This conditional should be encapsulated to make our intent clear. So let's extract a method that explains it.

```
    public String compact(String message) {
    if (shouldNotCompact())
      return Assert.format(message, expected, actual);
    findCommonPrefix();
    findCommonSuffix();
    String expected = compactString(this.expected);
    String actual = compactString(this.actual);
    return Assert.format(message, expected, actual);
  }
private boolean shouldNotCompact() {
    return expected == null || actual == null || areStringsEqual();
  }
```

I don't much care for the `this.expected` and `this.actual` notation in the `compact` function. This happened when we changed the name of `fExpected` to `expected`. Why are there variables in this function that have the same names as the member variables? Don't they represent something else [N4]? We should make the names unambiguous.

String **compactExpected** = compactString(**expected**); String **compactActual** = compactString(**actual**);

Negatives are slightly harder to understand than positives [G29]. So let's turn that `if` statement on its head and invert the sense of the conditional.

```
    public String compact(String message) {
  if (canBeCompacted()) {
    findCommonPrefix();
    findCommonSuffix();
    String compactExpected = compactString(expected);
    String compactActual = compactString(actual);
```

```
      return Assert.format(message, compactExpected, compactActual);
    } else {
      return Assert.format(message, expected, actual);
    }
  }
  private boolean canBeCompacted() {
    return expected != null && actual != null && ! areStringsEqual();
  }
```

The name of the function is strange [N7]. Although it does compact the strings, it actually might not compact the strings if `canBeCompacted` returns `false`. So naming this function `compact` hides the side effect of the error check. Notice also that the function returns a formatted message, not just the compacted strings. So the name of the function should really be `formatCompactedComparison`. That makes it read a lot better when taken with the function argument:

```
      public String formatCompactedComparison(String message) {
```

The body of the `if` statement is where the true compacting of the expected and actual strings is done. We should extract that as a method named `compactExpectedAndActual`. However, we want the `formatCompactedComparison` function to do all the formatting. The `compact…` function should do nothing but compacting [G30]. So let's split it up as follows:

```
    …
  private String compactExpected;
  private String compactActual;
…
  public String formatCompactedComparison(String message) {
    if (canBeCompacted()) {
      compactExpectedAndActual();
      return Assert.format(message, compactExpected, compactActual);
    } else {

      return Assert.format(message, expected, actual);
    }
  }
```

```
  private void compactExpectedAndActual() {
    findCommonPrefix();
    findCommonSuffix();
    compactExpected = compactString(expected);
    compactActual = compactString(actual);
  }
```

Notice that this required us to promote `compactExpected` and `compactActual` to member variables. I don't like the way that the last two lines of the new function return variables, but the first two don't. They aren't using consistent conventions [G11]. So we should change `findCommonPrefix` and `findCommonSuffix` to return the prefix and suffix values.

```
    private void compactExpectedAndActual() {
   prefixIndex =findCommonPrefix();
    suffixIndex =findCommonSuffix();
    compactExpected = compactString(expected);
    compactActual = compactString(actual);
  }
  private int findCommonPrefix() {
    int prefixIndex = 0;
    int end = Math.min(expected.length(), actual.length());
    for (; prefixIndex < end; prefixIndex++) {
      if (expected.charAt(prefixIndex) != actual.charAt(prefixIndex))
        break;
    }
    return prefixIndex;
  }
  private int findCommonSuffix() {
    int expectedSuffix = expected.length() - 1;
    int actualSuffix = actual.length() - 1;
    for (; actualSuffix >= prefixIndex && expectedSuffix >= prefixIndex;
        actualSuffix--, expectedSuffix--) {
      if (expected.charAt(expectedSuffix) != actual.charAt(actualSuffix))
        break;
    }
```

```
    return expected.length() - expectedSuffix;
  }
```

We should also change the names of the member variables to be a little more accurate [N1]; after all, they are both indices.

Careful inspection of `findCommonSuffix` exposes a *hidden temporal coupling* [G31]; it depends on the fact that `prefixIndex` is calculated by `findCommonPrefix`. If these two functions were called out of order, there would be a difficult debugging session ahead. So, to expose this temporal coupling, let's have `findCommonSuffix` take the `prefixIndex` as an argument.

```
    private void compactExpectedAndActual() {
    prefixIndex = findCommonPrefix();
    suffixIndex = findCommonSuffix(prefixIndex);
    compactExpected = compactString(expected);
    compactActual = compactString(actual);
  }
  private int findCommonSuffix(int prefixIndex) {
    int expectedSuffix = expected.length() - 1;
    int actualSuffix = actual.length() - 1;
    for (; actualSuffix >= prefixIndex && expectedSuffix >= prefixIndex;
        actualSuffix--, expectedSuffix--) {
      if (expected.charAt(expectedSuffix) != actual.charAt(actualSuffix))
        break;
    }
    return expected.length() - expectedSuffix;
  }
```

I'm not really happy with this. The passing of the `prefixIndex` argument is a bit arbitrary [G32]. It works to establish the ordering but does nothing to explain the need for that ordering. Another programmer might undo what we have done because there's no indication that the parameter is really needed. So let's take a different tack.

```
    private void compactExpectedAndActual() {
    findCommonPrefixAndSuffix();
    compactExpected = compactString(expected);
    compactActual = compactString(actual);
```

```
  }
  private void findCommonPrefixAndSuffix() {
    findCommonPrefix();
    int expectedSuffix = expected.length() - 1;
    int actualSuffix = actual.length() - 1;
    for (;
        actualSuffix >= prefixIndex && expectedSuffix >= prefixIndex;
        actualSuffix--, expectedSuffix--
      ) {

      if (expected.charAt(expectedSuffix) != actual.charAt(actualSuffix))
        break;
    }
    suffixIndex = expected.length() - expectedSuffix;
  }
  private void findCommonPrefix() {
    prefixIndex = 0;
    int end = Math.min(expected.length(), actual.length());
    for (;   prefixIndex < end; prefixIndex++)
      if (expected.charAt(prefixIndex) != actual.charAt(prefixIndex))
        break;
  }
```

We put `findCommonPrefix` and `findCommonSuffix` back the way they were, changing the name of `findCommonSuffix` to `findCommonPrefixAndSuffix` and having it call `findCommon-Prefix` before doing anything else. That establishes the temporal nature of the two functions in a much more dramatic way than the previous solution. It also points out how ugly `findCommonPrefixAndSuffix` is. Let's clean it up now.

```
    private void findCommonPrefixAndSuffix() {
    findCommonPrefix();
    int suffixLength = 1;
    for (; !suffixOverlapsPrefix(suffixLength); suffixLength++) {
      if (charFromEnd(expected, suffixLength) !=
          charFromEnd(actual, suffixLength))
        break;
    }
```

```
    suffixIndex = suffixLength;
  }
  private char charFromEnd(String s, int i) {
    return s.charAt(s.length()-i);}
  private boolean suffixOverlapsPrefix(int suffixLength) {
    return actual.length() - suffixLength < prefixLength ||
      expected.length() - suffixLength < prefixLength;
  }
```

This is much better. It exposes that the `suffixIndex` is really the length of the suffix and is not well named. The same is true of the `prefixIndex`, though in that case "index" and "length" are synonymous. Even so, it is more consistent to use "length." The problem is that the `suffixIndex` variable is not zero based; it is 1 based and so is not a true length. This is also the reason that there are all those `+1`s in `computeCommonSuffix` [G33]. So let's fix that. The result is in Listing 15-4.

Listing 15-4 `ComparisonCompactor.java (interim)`

```
  public class ComparisonCompactor {
…
  private int suffixLength;
…
  private void findCommonPrefixAndSuffix() {
    findCommonPrefix();
    suffixLength = 0;
    for (; !suffixOverlapsPrefix(suffixLength); suffixLength++) {
      if (charFromEnd(expected, suffixLength) !=
          charFromEnd(actual, suffixLength))
        break;
    }
  }
  private char charFromEnd(String s, int i) {
    return s.charAt(s.length() - i - 1);
  }
  private boolean suffixOverlapsPrefix(int suffixLength) {
    return actual.length() - suffixLength <= prefixLength ||
      expected.length() - suffixLength <= prefixLength;
```

```
    }
…
    private String compactString(String source) {
      String result =
        DELTA_START +
        source.substring(prefixLength, source.length() - suffixLength) +
        DELTA_END;
      if (prefixLength > 0)
        result = computeCommonPrefix() + result;
      if (suffixLength > 0)
        result = result + computeCommonSuffix();
      return result;
    }
  …
    private String computeCommonSuffix() {
      int end = Math.min(expected.length() - suffixLength +
        contextLength, expected.length()
      );
      return
        expected.substring(expected.length() - suffixLength, end) +
        (expected.length() - suffixLength <
          expected.length() - contextLength ?
          ELLIPSIS : "");
    }
```

We replaced the `+1`s in `computeCommonSuffix` with a `-1` in `charFromEnd`, where it makes perfect sense, and two `<=` operators in `suffixOverlapsPrefix`, where they also make perfect sense. This allowed us to change the name of `suffixIndex` to `suffixLength`, greatly enhancing the readability of the code.

There is a problem however. As I was eliminating the +1s, I noticed the following line in `compactString`:

```
if (suffixLength > 0)
```

Take a look at it in Listing 15-4. By rights, because `suffixLength` is now one less than it used to be, I should change the `>` operator to a `>=` operator. But that makes no sense. It makes sense *now!* This means that it

didn't use to make sense and was probably a bug. Well, not quite a bug. Upon further analysis we see that the `if` statement now prevents a zero length suffix from being appended. Before we made the change, the `if` statement was nonfunctional because `suffixIndex` could never be less than one!

This calls into question *both* `if` statements in `compactString`! It looks as though they could both be eliminated. So let's comment them out and run the tests. They passed! So let's restructure `compactString` to eliminate the extraneous `if` statements and make the function much simpler [G9].

```
    private String compactString(String source) {
    return
      computeCommonPrefix() +
      DELTA_START +
      source.substring(prefixLength, source.length() - suffixLength) +
      DELTA_END +
      computeCommonSuffix();
}
```

This is much better! Now we see that the `compactString` function is simply composing the fragments together. We can probably make this even clearer. Indeed, there are lots of little cleanups we could do. But rather than drag you through the rest of the changes, I'll just show you the result in Listing 15-5.

Listing 15-5 `ComparisonCompactor.java (final)`

```
    package junit.framework;
public class ComparisonCompactor {

  private static final String ELLIPSIS = "…";
  private static final String DELTA_END = "]";
  private static final String DELTA_START = "[";
  private int contextLength;
  private String expected;
  private String actual;
  private int prefixLength;
  private int suffixLength;
```

```
 public ComparisonCompactor(
   int contextLength, String expected, String actual
 ) {
   this.contextLength = contextLength;
   this.expected = expected;
   this.actual = actual;
 }

 public String formatCompactedComparison(String message) {
   String compactExpected = expected;
   String compactActual = actual;
   if (shouldBeCompacted()) {
     findCommonPrefixAndSuffix();
     compactExpected = compact(expected);
     compactActual = compact(actual);
   }
   return Assert.format(message, compactExpected, compactActual);
 }

 private boolean shouldBeCompacted() {
   return !shouldNotBeCompacted();
 }

private boolean shouldNotBeCompacted() {
  return expected == null ||
         actual == null ||
         expected.equals(actual);
}

private void findCommonPrefixAndSuffix() {
  findCommonPrefix();
  suffixLength = 0;
  for (; !suffixOverlapsPrefix(); suffixLength++) {
    if (charFromEnd(expected, suffixLength) !=
        charFromEnd(actual, suffixLength)
    )
```

```
      break;
    }
  }
  private char charFromEnd(String s, int i) {
    return s.charAt(s.length() - i - 1);
  }
  private boolean suffixOverlapsPrefix() {
    return actual.length() - suffixLength <= prefixLength ||
      expected.length() - suffixLength <= prefixLength;
  }

  private void findCommonPrefix() {
    prefixLength = 0;
    int end = Math.min(expected.length(), actual.length());
    for (; prefixLength < end; prefixLength++)
      if (expected.charAt(prefixLength) != actual.charAt(prefixLength))
        break;
  }

  private String compact(String s) {
    return new StringBuilder()
      .append(startingEllipsis())
      .append(startingContext())
      .append(DELTA_START)
      .append(delta(s))
      .append(DELTA_END)
      .append(endingContext())
      .append(endingEllipsis())
      .toString();
  }
  private String startingEllipsis() {
    return prefixLength > contextLength ? ELLIPSIS : "";
  }
  private String startingContext() {
    int contextStart = Math.max(0, prefixLength - contextLength);
    int contextEnd = prefixLength;
```

```
    return expected.substring(contextStart, contextEnd);
  }
  private String delta(String s) {
    int deltaStart = prefixLength;
    int deltaEnd = s.length() - suffixLength;
    return s.substring(deltaStart, deltaEnd);
  }
  private String endingContext() {
    int contextStart = expected.length() - suffixLength;
    int contextEnd =
      Math.min(contextStart + contextLength, expected.length());
    return expected.substring(contextStart, contextEnd);
  }
  private String endingEllipsis() {
    return (suffixLength > contextLength ? ELLIPSIS : "");
  }
}
```

This is actually quite pretty. The module is separated into a group of analysis functions and another group of synthesis functions. They are topologically sorted so that the definition of each function appears just after it is used. All the analysis functions appear first, and all the synthesis functions appear last.

If you look carefully, you will notice that I reversed several of the decisions I made earlier in this chapter. For example, I inlined some extracted methods back into `formatCompactedComparison`, and I changed the sense of the `shouldNotBeCompacted` expression. This is typical. Often one refactoring leads to another that leads to the undoing of the first. Refactoring is an iterative process full of trial and error, inevitably converging on something that we feel is worthy of a professional.

Conclusion

And so we have satisfied the Boy Scout Rule. We have left this module a bit cleaner than we found it. Not that it wasn't clean already. The authors had done an excellent job with it. But no module is immune from

improvement, and each of us has the responsibility to leave the code a little better than we found it.