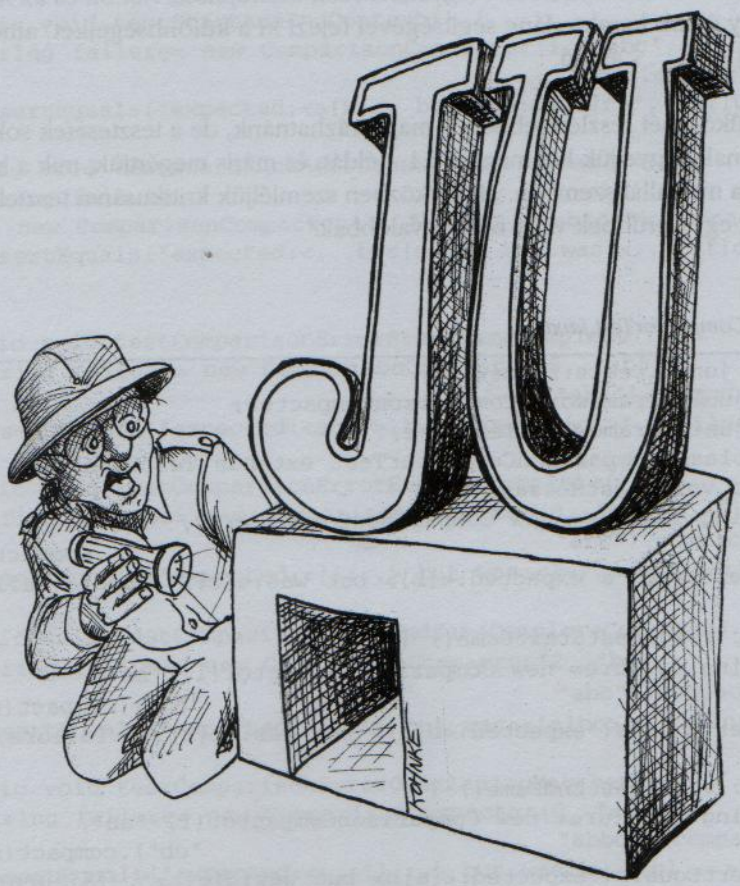
# 15

# A JUnit belső részletei




A JUnit az egyik legismertebb Java-keretrendszer – mint ilyen, egyszerű elgondolásra épül, pontosan meghatározott, a megvalósítása pedig elegáns. De hogy is néz ki a kódja? Ebben a fejezetben ezt vizsgáljuk, a JUnit környezet egy részletét kiemelve.

## A JUnit keretrendszer

A JUnit-nak manapság rengeteg szerzője van, de a kezdet kezdetén a történet úgy indult, hogy Kent Beck és Eric Gamma együtt utaztak egy repülőn Atlanta felé. Kent Javát akart tanulni, Eric pedig Kent Smalltalk-tesztkörnyezetéről szeretett volna többet megtudni. „Végy két számítógépőrültet, zárd össze őket egy szűk helyre, és mit kapsz? Hát persze, hogy programozni kezdenek!"[1] Az eredmény nem maradt el – három órai magasröptű munka után lerakták a JUnit alapjait.

Az általunk vizsgált modul okos módszerekkel segít a karakterláncok összehasonlításánál előforduló hibák azonosításában. A modul neve `ComparisonCompactor`. Ha megadunk két eltérő karakterláncot – legyenek ezek mondjuk az `ABCDE` és az `ABXDE`–, a modul egy újabb karakterlánc segítségével fejezi ki a különbségeiket, amely ez esetben `<...B[X]D...>` alakú.

A modul működését részletesebben is magyarázhatnánk, de a tesztesetek sokkal jobb áttekintést adnak. Figyeljük hát meg a 15.1. példát, és máris megértjük, mik a követelmények ezzel a modullal szemben. Mindeközben szemléljük kritikusan a tesztek szerkezetét. Lehetnének egyszerűbbek vagy nyilvánvalóbbak?

### 15.1. példa

*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);
  }
```

*folytatódik*

[1] *JUnit Pocket Guide*, Kent Beck, O'Reilly, 2004, 43. oldal.

### 15.1. példa

*ComparisonCompactorTest.java – folytatás*

```
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);
```

*folytatódik*

### 15.1. példa

*ComparisonCompactorTest.java – folytatás*

```
    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);
  }
}
```

A fenti tesztekkel végrehajtottam egy kódlefedettség-elemzést a `ComparisonCompactor` *modulon. Az eredmény: a lefedettség 100 százalékos. A tesztek végrehajtanak minden kód*sort – egyetlen `if` utasítás vagy `for` ciklus sem maradhat ki. Mindez megalapozza a kódba vetett bizalmunkat, és csak elismerőleg szólhatunk a szerzők mesterségbeli tudásáról.

A `ComparisonCompactor` kódját a 15.2. példában láthatjuk. Szenteljünk némi időt az átolvasásának. Úgy vélem, szépen felosztott, kellően kifejező és egyszerű szerkezetű kódnak találjuk majd. Ha ezzel végeztünk, rátérhetünk a részletekre.

### 15.2. példa

*ComparisonCompactor.java (eredeti)*

```
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());
```

*folytatódik*

### 15.2. példa

*ComparisonCompactor.java (eredeti) – folytatás*

```
        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);
    }
}
```

Lehetnek persze apró fenntartásaink a modullal kapcsolatban. Akad néhány hosszú kifejezés, néhány furcsa +1, és így tovább. Mindezek mellett a modul jónak mondható – képzeljük el mi lenne, ha úgy festene, mint a 15.3. példában látható változat.

### 15.3. példa

*ComparisonCompator.java (rontott változat)*

```
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;
   }
}
```

Jóllehet a szerzők igen jó munkát végeztek, a *cserkészek szabálya* (Boy Scout Rule)[2] szerint tisztábban kell átadnunk, mint ahogy rátaláltunk. Kérdés, miként javíthatunk a 15.2. példa tartalmán?

Az első ilyen apróság a tagváltozók `f` előtagja [N6]. A mai környezetek feleslegessé teszik a hatókör ilyen megjelölését. Szabaduljunk hát meg az `f`-ektől:

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

Láthatunk egy egységbe nem zárt feltételes utasítást a `compact` függvény elején [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);
}
```

A feltételt egységbe kell zárnunk, hogy a szándékainkat tisztáztuk. Emeljünk hát ki egy tagfüggvényt, amely világossá teszi a helyzetet:

```
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();
}
```

A `compact` függvényben szereplő `this.expected` és `this.actual` jelölések akkor jöttek be a képbe, amikor az `fExpected` helyett áttértünk az `expected` névre. Jó az, ha a függ-

[2] Lásd a cserkészek szabályát az 1. fejezetben.

vényben ugyanolyan nevű változók vannak, mint az osztály tagváltozói? Nem lehet, hogy valami mást jelenítenek meg [N4]? Jobb, ha feloldjuk ezt az ellentmondást:

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

A negatív állításokat mindig nehezebb megérteni, mint a pozitívakat [G29]. Állítsuk hát a feje tetejére az `if` utasítást, és fordítsuk meg a feltétel értelmét:

```
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();
}
```

A függvény neve meglehetősen fura [N7]. Jóllehet valóban tömöríti (compact) a karakterláncokat, de előfordulhat, hogy mégsem teszi ezt meg, amennyiben a `canBeCompacted` a `false` értékkel tér vissza. Vagyis, a `compact` elnevezés valójában elrejti azt a tényt, hogy időközben egy hibaellenőrzés is történik. Vegyük észre azt is, hogy a függvény egy formázott üzenetet ad vissza, nem csak a tömörített karakterláncokat. Így tehát jobb, ha a függvénynek a `formatCompactedComparison` nevet adjuk. Így a paraméterével együtt sokkal olvashatóbbá válik:

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

Az elvárt és a megadott karakterlánc tömörítése voltaképpen az `if` utasítás törzsében megy végbe. Ezt a műveletet kiemelhetjük egy `compactExpectedAndActual` nevű tagfüggvénybe. Mindazonáltal azt szeretnénk, hogy a teljes formázást a `formatCompactedComparison` végezze. A `compact...` függvény a tömörítésen kívül nem tehet mást [G30]. Osszuk hát fel az alábbiak szerint:

```
...
  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);
  }
```

Fontos látnunk, hogy ezzel a `compactExpected` és a `compactActual` értékét tagváltozóként kell továbbadnunk. Nem igazán tetszetős, hogy a függvény utolsó két sora visszaad valamilyen értéket, míg az első kettő nem. Ez arra utal, hogy nincs egységes használati alakjuk [G11]. Ezért módosítanunk kell a `findCommonPrefix` és a `findCommonSuffix` függvényeket úgy, hogy ezen túl visszaadják az elő- és utótagokat:

```
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;
}
```

A pontosság kedvéért módosítanunk kell a tagváltozók neveit is [N1], hiszen sorszámokról van szó.

A `findCommonSuffix` alapos vizsgálatával ráakadhatunk egy *rejtett ideiglenes csatolásra* is [G31], aminek az az alapja, hogy a `prefixIndex` értékét a `findCommonPrefix` számítja ki. Ha ezt a két függvényt nem a megfelelő sorrendben hívjuk meg, komoly hibakeresésnek nézhetünk elébe. A csatolás felszínre hozatalához illesszük be a `prefixIndex` értékét a `findCommonSuffix` paraméterei közé:

```
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;
}
```

Ezzel azonban még mindig nem lehetünk elégedettek. A `prefixIndex` paraméter átadása némiképp légből kapott ötletnek tűnik [G32]. Nagyszerűen kikényszeríti a műveletek sorrendjét, de továbbra sem tudjuk, miért is van szükség erre a sorrendre. Egy programozótársunk a későbbiekben nyugodtan visszavonhatja ezt az átalakítást, mert semmi sem jelzi, hogy a paraméterre valóban szükség van. Próbáljunk ki egy másik megközelítést:

```
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;
    }
```

Visszaállítottuk a `findCommonPrefix` és a `findCommonSuffix` eredeti alakját, majd az utóbbinak a `findCommonPrefixAndSuffix` nevet adtuk, és úgy módosítottuk a kódját, hogy minden egyéb műveletet megelőzően hívja meg a `findCommonPrefix` függvényt. Így a két függvény időbeli sorrendje sokkal nyilvánvalóbbá válik, mint a korábbi megoldásban. Ráadásul azt is észrevehetjük, milyen csúnya a `findCommonPrefixAndSuffix` kódja. Tisztítsuk is meg gyorsan:

```
    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;
    }
```

No, ez már sokkal kellemesebb! Ugyanakkor észrevehetjük, hogy a `suffixIndex` voltaképpen az utótag hosszát takarja, így nem igazán megfelelő a neve. Hasonló a helyzet a `prefixIndex` esetében, jóllehet az „index" (sorszám) ez esetben a hossz szinonimája. Akárhogy is, a „length" (hossz) használata sokkal következetesebb megoldást jelent. További gondokat okoz, hogy a `suffixIndex` változó számozása nem 0-tól, hanem 1-től kezdődik, vagyis nem pontosan a hosszt adja meg. Ez az oka a `+1`-ek megjelenésének is a `computeCommonSuffix` függvényben [G33]. Ha mindezt kijavítottuk, a modul a 15.4. példa szerint alakul.

### 15.4. példa

*ComparisonCompactor.java (köztes állapot)*

```
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 : "");
  }
```

A `computeCommonSuffix` függvényben található `+1`-ek helyett a `charFromEnd` függvényben jelentek meg `-1`-ek – itt azonban már jó helyen vannak. Emellett a `suffixOverlapsPrefix` függvényben is szükség volt két `>=` jelre, amelyek itt már szintén értelmet nyernek. Mindezek után a `suffixIndex` nevét `suffixLength`-re módosítottuk, jelentősen javítva a kód olvashatóságát.

Van azonban itt egy kis gond. A `+1`-ek kiiktatása közben figyelmes lettem a következő sorra a `compactString` függvényben:

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

Figyeljük csak meg a 15.4. példában! Ha minden igaz, mivel a `suffixLength` most eggyel kisebb, mint korábban, a `>` helyett a `>=` műveletet kellene használnunk. Ennek azonban semmi értelme. Az itt látható alak a helyes. Mindez azt jelenti, hogy a kódsor a korábbiakban volt értelmetlen, vagyis valószínűleg egy hibára akadtunk. Nos, talán nem is egy egyszerű hibáról van szó... Ha tovább vizsgáljuk a kódot, láthatjuk, hogy az `if` utasítás most megakadályozza, hogy nulla hosszúságú utótagot fűzzünk a karakterlánchoz. Mielőtt elvégeztük volna a módosításainkat, az `if` utasítás egész egyszerűen felesleges volt, hiszen a `suffixIndex` értéke nem lehetett 1-nél kisebb.

Ennek fényében érdemes megvizsgálnunk a `compactString` mindkét `if` utasítását. Úgy fest, mindkettőjüktől megszabadulhatunk. Tegyük őket megjegyzésbe, és futtassuk a teszteket. Siker! Alakítsuk át a `compactString` függvényt, amely az `if` utasítások nélkül sokkal egyszerűbbé válik [G9]:

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

Így már mindjárt más! Most már világos, hogy a `compactString` egyszerűen összerakja a darabokat. De talán még világosabbá is tehetjük a dolgokat. Nos, valóban, rengeteg apró zugban tisztogathatjuk még a kódot. Ahelyett azonban, hogy elmerülnénk a részletekben, inkább lássuk a végeredményt a 15.5. példában.

### 15.5. példa

*ComparisonCompactor.java (végső változat)*

```
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;
    }
  }
```

*folytatódik*

### 15.5. példa

*ComparisonCompactor.java (végső változat) – folytatás*

```
    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 : "");
    }
}
```

Ez már döfi! A modul szépen szétvált az analízis és a szintézis függvényeire. Az elhelyezkedésük kényelmes, hiszen amint használunk egy függvényt, nyomban megtalálhatjuk a meghatározását. Az analízis feladatait ellátó függvények állnak elöl, míg a szintézis felelősei utánuk következnek.

Ha jobban odafigyelünk, láthatjuk, hogy számos, a korábbiakban hozott döntésemet visszavontam. Így például visszahelyeztem néhány kiemelt tagfüggvény kódját a `formatCompactedComparison` szerkezetébe, és megváltoztattam a `shouldNotBeCompacted` kifejezés értelmét. Ez jellemző eljárás – gyakori ugyanis, hogy egy átalakítás egy másik visszavonását teszi szükségszerűvé. Az átalakítás egy lépésről lépésre végbemenő folyamat rengeteg próbálkozással és kudarccal, de végül elvezet egy olyan kódhoz, amelyet profi programozóhoz méltónak tartunk.

## Összefoglalás

Kielégítettük tehát a cserkészek szabályát, hiszen a modul némiképp tisztábban került ki a kezünk alól, mint ahogy hozzájutottunk. Nem mintha eredetileg nem lett volna elég tiszta – a szerzők kitűnő munkát végeztek. Egyetlen modul sem lehet azonban immunis a fejlesztésre, és mindannyian felelősek vagyunk azért, hogy a kódot kissé tisztábban adjuk tovább, mint ahogy megkaptuk.