The Craftsman: 16
SMCRemote Part VI
Wham Bam

Robert C. Martin
18 July, 2003

...Continued from last month. See <<link>> for last month's article, and the code we were working on. You can download that code from:

www.objectmentor.com/resources/articles/CraftsmanCode/Craftsman_15_SMCRemote_V_EssArePee.zip p

---

Jasmine just stood there staring daggers at me. I tried to lower my head, but I couldn't break contact with her eyes. After about half a minute she rolled those eyes, and stared at the ceiling, tapping her foot. Finally, she shook her head, squared her shoulders and strode over to me. She had an odd, sad, look on her face.

"Alphonse," she said sternly, "that is territory that you are never to tread again. Do you understand me?"

Thoroughly ashamed of myself, I nodded and said: "Yes Jasmine. I'm -- uh -- I'm sorry."

"And from now on, call me Ma'am."

"Yes,... Ma'am." Oh god this was excruciating.

She gave an wry little snort that made her hair bounce and shimmer and then she said: "OK, let's see what you've done."

I showed her the `FileCarrier` code, and the tests I had written. She seemed satisfied at first but then said: "Look at the way `FileCarrier` reads and writes the file.

```
public class FileCarrier implements Serializable {
  private String fileName;
  private char[] contents;

  public FileCarrier(String fileName) throws Exception {
    File f = new File(fileName);
    this.fileName = fileName;
    int fileSize = (int)f.length();
    contents = new char[fileSize];
    FileReader reader = new FileReader(f);
    reader.read(contents);
    reader.close();
  }

  public void write() throws Exception {
    FileWriter writer = new FileWriter(fileName);
    writer.write(contents);
    writer.close();
```

```
  }

  public String getFileName() {
    return fileName;
  }
  public char[] getContents() {
    return contents;
  }
}
```

"`FileCarrier` reads the file with a single read, and writes it with a single write. This works for small examples, but there are a few problems with it. First of all, I'm not convinced that the read won't abort early, filling only part of the array. Secondly, the carried file is going to be transmitted over a socket to another system. That system may use a different line-end character. So I don't think `FileCarrier` will work well across foreign systems."

I was listening. I was listening *hard*. But some disconnected part of me watched raptly as she focused those black eyes on the screen. Her lashes, nose, and lips danced as she spoke.

"What do you think we should do about that Alphonse?"

I didn't miss a beat. "Well, -- er -- *Ma'am*, we should probably read and write the files a line at a time, and carry the files in a list of lines."

"All right Alphonse, why don't you make that change?"

I directed my attention to the keyboard and, bit-by-bit, I made the appropriate changes to the `FileCarrier`. I took special care to keep the tests running. Once everything was working I refactored the class so that it read as clearly and cleanly as I could make it. I was on my very best behavior.

```
public class FileCarrier implements Serializable {
  private String fileName;
  private LinkedList lines = new LinkedList();

  public FileCarrier(String fileName) throws Exception {
    this.fileName = fileName;
    loadLines();
  }

  private void loadLines() throws IOException {
    BufferedReader br = makeBufferedReader();
    String line;
    while ((line = br.readLine()) != null)
      lines.add(line);
    br.close();
  }

  private BufferedReader makeBufferedReader()
    throws FileNotFoundException {
    return new BufferedReader(
      new InputStreamReader(
        new FileInputStream(fileName)));
  }

  public void write() throws Exception {
    PrintStream ps = makePrintStream();
    for (Iterator i = lines.iterator(); i.hasNext();)
      ps.println((String) i.next());
    ps.close();
  }

  private PrintStream makePrintStream() throws FileNotFoundException {
    return new PrintStream(
      new FileOutputStream(fileName));
  }
```

```
  public String getFileName() {
    return fileName;
  }
}
```

"That's very good, Alphonse."

"Thank you Ma'am." I suppressed a cringe.

"However, I don't think `FileCarrierTest` really ensures that `FileCarrier` faithfully reproduces the file. I'd like to see some more tests."

She was being too polite. I was starting to wish she'd call me Hot Shot again.

"I agree, Ma'am. I'll improve the test."

Once again I build the code bit-by-bit, keeping the tests running as I made the changes. Once again I refactored with great care until I was very sure that the code was as clean and expressive as I could make it. I didn't want to give Ma'am any more reason to be angry or disappointed.

```
public class FileCarrierTest extends TestCase {
  public void testFileCarrier() throws Exception {
    final String ORIGINAL_FILENAME = "testFileCarrier.txt";
    final String RENAMED_FILENAME = "testFileCarrierRenamed.txt";
    File originalFile = new File(ORIGINAL_FILENAME);
    File renamedOriginal = new File(RENAMED_FILENAME);

    ensureFileIsRemoved(originalFile);
    ensureFileIsRemoved(renamedOriginal);

    createTestFile(originalFile);
    FileCarrier fc = new FileCarrier(ORIGINAL_FILENAME);
    rename(originalFile, renamedOriginal);
    fc.write();

    assertTrue(originalFile.exists());
    assertTrue(filesAreTheSame(originalFile, renamedOriginal));

    originalFile.delete();
    renamedOriginal.delete();
  }

  private void rename(File oldFile, File newFile) {
    oldFile.renameTo(newFile);
    assertTrue(oldFile.exists() == false);
    assertTrue(newFile.exists());
  }

  private void createTestFile(File file) throws IOException {
    PrintWriter w = new PrintWriter(new FileWriter(file));
    w.println("line one");
    w.println("line two");
    w.println("line three");
    w.close();
  }

  private void ensureFileIsRemoved(File file) {
    if (file.exists()) file.delete();
    assertTrue(file.exists() == false);
  }

  private boolean filesAreTheSame(File f1, File f2) throws Exception {
    FileInputStream r1 = new FileInputStream(f1);
    FileInputStream r2 = new FileInputStream(f2);
```

```
    try {
      int c;
      while ((c = r1.read()) != -1) {
        if (r2.read() != c) {
          return false;
        }
      }
      if (r2.read() != -1)
        return false;
      else
        return true;
    } finally {
      r1.close();
      r2.close();
    }
  }
}
```

Ma'am studied the code as I wrote it. She never looked at me -- never once. As uncomfortable as they had made me, her penetrating glares and snide remarks were better than this formal etiquette. The tension between us was palpable.

"That's very nice, Alphonse. Good clean code. I especially like the way you made sure that the original file was renamed, and that the new file was created. The way you've written it, there can't be any doubt that the `FileCarrier` is creating the file. There's no way the old file could have been left behind.

"The one problem I have with it is that I haven't seen the `filesAreTheSame` method fail. Do you think it really works properly?"

I carefully examined the code. I couldn't see any flaw. I was pretty sure it had to work. But I wasn't going to make any assertions without evidence. So I started writing some tests for the `filesAreTheSame` method.

I began by writing a test that showed that the method worked for two files that were equal. Then I wrote a test that showed that two different files did not compare the same. I wrote a test that showed that if one file were a prefix of the other, they would not compare. In the end I wrote five different test cases.

These five test cases had a *lot* of duplicate code. Each wrote two files. Each compared the two files. Each deleted the two files. To get rid of this duplication I used the Template Method pattern. I moved all the common code into an abstract base class called `FileComparator`. I moved all the differentiating code into simple anonymous derivatives. Thus, each test case created a new anonymous derivative that specified nothing more than the contents of the two files, and the sense of the comparison.

As always, I wrote this code one tiny step at a time, running the tests between each step, and carefully refactoring whenever I could. Ma'am never took her eyes off the screen. She was purposefully avoiding any informal contact. Once, our elbows accidentally touched. Shivers ran through me; but she didn't show any reaction at all. A few minutes later she moved her chair a few centimeters further from mine.

```
public class FileCarrierTest extends TestCase {
  private abstract class FileComparator {
    abstract void writeFirstFile(PrintWriter w);
    abstract void writeSecondFile(PrintWriter w);

    void compare(boolean expected) throws Exception {
      File f1 = new File("f1");
      File f2 = new File("f2");
      PrintWriter w1 = new PrintWriter(new FileWriter(f1));
      PrintWriter w2 = new PrintWriter(new FileWriter(f2));
      writeFirstFile(w1);
      writeSecondFile(w2);
      w1.close();
      w2.close();
      assertEquals("(f1,f2)", expected, filesAreTheSame(f1, f2));
```

```
    assertEquals("(f2,f1)", expected, filesAreTheSame(f2, f1));
    f1.delete();
    f2.delete();
  }
}

public void testOneFileLongerThanTheOther() throws Exception {
  FileComparator c = new FileComparator() {
    void writeFirstFile(PrintWriter w) {
      w.println("hi there");
    }

    void writeSecondFile(PrintWriter w) {
      w.println("hi there you");
    }
  };
  c.compare(false);
}

public void testFilesAreDifferentInTheMiddle() throws Exception {
  FileComparator c = new FileComparator() {
    void writeFirstFile(PrintWriter w) {
      w.println("hi there");
    }

    void writeSecondFile(PrintWriter w) {
      w.println("hi their");
    }
  };
  c.compare(false);
}

public void testSecondLineDifferent() throws Exception {
  FileComparator c = new FileComparator() {
    void writeFirstFile(PrintWriter w) {
      w.println("hi there");
      w.println("This is fun");
    }

    void writeSecondFile(PrintWriter w) {
      w.println("hi there");
      w.println("This isn't fun");
    }
  };
  c.compare(false);
}

public void testFilesSame() throws Exception {
  FileComparator c = new FileComparator() {
    void writeFirstFile(PrintWriter w) {
      w.println("hi there");
    }

    void writeSecondFile(PrintWriter w) {
      w.println("hi there");
    }
  };
  c.compare(true);
}

public void testMultipleLinesSame() throws Exception {
  FileComparator c = new FileComparator() {
    void writeFirstFile(PrintWriter w) {
```

```
        w.println("hi there");
        w.println("this is fun");
        w.println("Lots of fun");
      }

      void writeSecondFile(PrintWriter w) {
        w.println("hi there");
        w.println("this is fun");
        w.println("Lots of fun");
      }
    };
    c.compare(true);
  }
```

"Alphonse, this is very nice."

I wanted to scream.

"I love the way you used the Template Method pattern to eliminate duplication. I'm glad you are taking your studies so seriously. Many apprentices don't learn their patterns until their journeymen force them to.

"I also like the way you tested the commutivity of equality. Every comparison happens in both directions. Splendid!"

Egad! When was she going to complain about something? Where had Jasmine gone?

"Thank you, Ma'am."

To be continued...

The code that Alphonse and Jasmine finished can be retrieved from:

www.objectmentor.com/resources/articles/CraftsmanCode/Craftsman_16_SMCRemote_VI_Wham_Bam.zip