The Craftsman: 22
SMCRemote Part XII
Bug Eye.

Robert C. Martin
8 January 2004

...Continued from last month. You can download last month's code from:
w www.objectmentor.com/resources/articles/CraftsmanCode/Craftsman_21_Patchwork.zip

---

Jean invited me join her in the journeyman's lounge, but I needed to clear my head. So I excused myself and went out to an observation bubble. The starbow was a brilliant stripe of color painted across the heavens. It was a concentric with our ship, and so always appeared below us whenever we looked down through the transparent floor of the bubble. The ship's rotation made the starbow look like a river of colored stars slowly streaming under the floor.

It was hard for me to believe that I had just met Jean for the first time this morning. Somehow it seemed like a much longer time. As I looked back on that time I realized that in the few hours we spent together we got quite a bit done; and yet through it all I had felt so *frustrated*. Somehow the way she worked made me feel like were moving so slowly. Jean seemed to think that moving slowly was a good thing. She had given me more than one lecture about taking the time and care to build the best software we can. I couldn't disagree with what she was saying, and I was frankly impressed by how much we had really gotten done today, and yet it still felt too slow.

I guess, during my classes in school, I had gotten used to writing code quickly and spending lots of time debugging. Somehow the debugging didn't seem slow to me. Somehow it felt like I was going fast. But when I worked with Jerry, Jasmine, and Jean we wrote our code much more slowly. And we wrote all those tests that seemed to take such a long time. We hadn't spent *any* time debugging so far; and yet somehow that didn't make it *feel* faster. It probably *was* faster, but it felt ponderous.

As I took the turbo back up to our lab, I resolved to stop feeling frustrated. Our pace was good, our quality was high, and my feelings of frustration were just old baggage that I needed to get rid of.

When I got to the lab a fellow about my age was sitting in my seat. There was no sign of Jean.

"Hello." I said. "Can I help you?"

"Uh… hi….uh, I'm Avery. Jean said I should work with you for the rest of the day."

"Oh. She did? Uh…"

"Yeah, she said you were supposed to show me how to write unit tests."

"Oh…uh…Really? Don't you know?"

"Yeah, I know…uh…Jason fired me."

"Jason was your Journeyman? He *fired* you?"

"Yeah…uh…Not really, he just asked Jean if he could stop working with me, so Jean told me to work with you for today."

"Why did Jason *do* that?"

"I worked through a break while Jason was gone, and wrote a whole bunch of code without any tests. When he came back he got all upset and told me to delete it. I got mad back at him and told him I wouldn't delete it. So he just left."

I felt a bit sheepish. Did everybody go through something like this? "He just left?" I asked.

"Yeah, he got all bug-eyed and red in the face, and then just walked out of our lab. Next thing I know Jean is telling me to work with you while she finds me a new journeyman. She said you'd understand."

Now I felt even more sheepish. "Uh, yeah, I guess I do. When did you start your apprenticeship?"

"Last week, same as you. I worked with Jimmy, and Joseph last week, and with Jason today. I got along OK with the other two, but there was just something about Jason that set me off. I guess I set him off too."

"I guess that happens sometimes. Shall we get to work?"

"Why not?"

I told him about the `SMCRemote` project. I explained what Jean and I had been up to for the last few hours. I told him about the client software that we had gotten working this morning and the server software that Jean and I hade been putting together since then. When I finished he said: "It sounds like you are about ready to have the server run a compile."

"Yes, I think we are. Tell you what, if I write the test, will you make it pass?"

"Sure, why not. I guess I'm going to have to get used to this testing stuff."

"Yes, I think you are. OK, so here's the test I'm thinking of."

```
  public void testCompileIsRunInEmptyDirectory() throws Exception {
    File dummy = new File("dummyFile");
    dummy.createNewFile();
    CompileFileTransaction cft = new CompileFileTransaction("dummyFile");
    dummy.delete();

    CompilerResultsTransaction crt = SMCRemoteServer.compile(cft, "ls >files");

    File files = new File("files");
    assertFalse(files.exists());

    crt.write();
    assertTrue(files.exists());

    BufferedReader reader = new BufferedReader(new FileReader(files));
    HashSet lines = new HashSet();
    String line;
    while ((line = reader.readLine()) != null) {
      lines.add(line);
    }
    reader.close();
    files.delete();

    assertEquals(2, lines.size());
    assertTrue(lines.contains("files"));
    assertTrue(lines.contains("dummyFile"));
  }
```

"Yikes!" said Avery. "OK, let me see if I understand this. You create a empty dummy file and load it into a `CompileFileTransaction`. Then you 'compile' it but use a '`ls >files`' command instead of the normal compile command. You expect the compile function to return a `CompilerResultsTransaction`. You expect that transaction to have the file that contains the output of the '`ls`' command. You read that file and make sure that the '`ls`' command saw nothing but the `dummyFile` and the `files` file. I guess that proves that the compile was run in a directory whose sole contents was the `dummyFile`."

This Avery character was no fool. "Yes, that's how I see the test working. Can you make it pass?"

"Jimmy told me you should always make the test fail before you try to make it pass. He said to do the easiest thing that would make the test fail. So, I guess something like this."

```
public static CompilerResultsTransaction
compile(CompileFileTransaction cft, String command)
{
  return null;
}
```

Avery ran the tests, and they failed. "OK, this fails because it returns a null pointer. We can fix that like so.

```
public static CompilerResultsTransaction
compile(CompileFileTransaction cft, String command) throws Exception
{
  return new CompilerResultsTransaction("");
}
```

"OK, that fails because there's no filename for the `CompilerResultsTransaction`. So we're going to need a file. We get that file by executing the compiler.

```
public static CompilerResultsTransaction
compile(CompileFileTransaction cft, String command) throws Exception
{
  executeCommand(command);
  return new CompilerResultsTransaction("files");
}
```

"Hmm, this fails for the same reason. The `files` file doesn't exist. But how could that be."

After some research we found that the `Runtime.exec()` function does not interpret the > as a file redirection command. Fixing that was a matter of changing the test as follows:

```
    CompilerResultsTransaction crt =
      SMCRemoteServer.compile(cft, "sh -c \"ls >files\"");
```

"OK, now the test fails at the `assertFalse(files.exists())` line. This is because we are executing the `ls` command in the current directory. We need to create a subdirectory and execute it there."

```
public static CompilerResultsTransaction
compile(CompileFileTransaction cft, String command) throws Exception
{
  File wd = makeWorkingDirectory();
  //How do I execute the command in the working directory?
  executeCommand(command);
  return new CompilerResultsTransaction("files");
}
```

"Alphonse, I can create the working directory with that function that you and Jean wrote. But how do I get the command to execute in that directory. There doesn't seem to be any way to tell `Runtime.exec()` what directory to run in."

Avery and I searched the documents, but we couldn't find any obvious way to solve this. Then I had an idea.

"Avery, why don't we prefix the command with a "`cd workingDirectory;`".

"Hmmm. That might work but it means that we put the `sh -c` stuff in the wrong place. We should have put it in `SMCRemoteServer.executeCommand`. I'll bet that the `Runtime.exec()` function can't deal with multiple commands and semicolons."

"You're probably right. Even if you aren't, the executeCommand function is still a better place for the sh –c stuff. Let's move it.

```
  public static boolean executeCommand(String command) {
    Runtime rt = Runtime.getRuntime();
    try {
      Process p = rt.exec("sh -c \"" + command + "\"");
      p.waitFor();
      return p.exitValue() == 0;
    }
    catch (Exception e) {
      return false;
    }
  }
```

"OK, now let's do that cd thing.

```
  public static CompilerResultsTransaction
  compile(CompileFileTransaction cft, String command) throws Exception
  {
    File workingDirectory = makeWorkingDirectory();
    String wd = workingDirectory.getName();
    executeCommand("cd " + wd + ";" + command);
    return new CompilerResultsTransaction("files");
  }
```

"And now, once again, it fails at the new CompilerResultsTransaction("files") line. And it's obvious why! That function is not executing in the subdirectory. We need to add the subdirectory path to the file name."

```
  return new CompilerResultsTransaction(wd + "/files");
```

"No, that doesn't work either. Now it fails because the CompilerResultsTransaction thinks the name includes the subdirectory path, and the write function is trying to write the file back into the subdirectory."

I sighed. "This is complicated."

"No, " said Avery, "we just have to pass the subdirectory to the CompilerResultsTransaction constructor so that it can load the file without storing the path in the filename.

```
  public static CompilerResultsTransaction
  compile(CompileFileTransaction cft, String command) throws Exception
  {
    File workingDirectory = makeWorkingDirectory();
    String wd = workingDirectory.getName();
    executeCommand("cd " + wd + ";" + command);
    return new CompilerResultsTransaction(workingDirectory, "files");
  }
```

```
public class CompilerResultsTransaction implements Serializable {
  private FileCarrier resultFile;

  public CompilerResultsTransaction(File subdirectory,
                                    String filename) throws Exception {
    resultFile = new FileCarrier(subdirectory, filename);
  }

  public void write() throws Exception {
    resultFile.write();
  }
}
```

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

  public FileCarrier(String fileName) throws Exception {
    this(null, fileName);
  }

  public FileCarrier(File subdirectory, String fileName) throws Exception {
    this.fileName = fileName;
    this.subdirectory = subdirectory;
    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(new File(subdirectory, 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;
  }
}
```

“Whoah! Now it’s getting pretty far. It’s failing at the `assertEquals(2, lines.size())` line of the test. And again it’s pretty clear why. We never wrote the input file. That should be easy to fix!”

“Not that easy. We’ll have to make sure we write it into the subdirectory.”

“Ah, you’re right! We have to add the `workingDirectory` to the `FileCarrier.write` function. Good catch!”

```
  public static CompilerResultsTransaction
  compile(CompileFileTransaction cft, String command) throws Exception
  {
    File workingDirectory = makeWorkingDirectory();
    String wd = workingDirectory.getName();
    cft.sourceFile.write(workingDirectory);
    executeCommand("cd " + wd + ";" + command);
    return new CompilerResultsTransaction(workingDirectory, "files");
  }
```

```
  public void write() throws Exception {
    write(null);
  }

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

  private PrintStream
  makePrintStream(File subdirectory) throws FileNotFoundException {
    return new PrintStream(
      new FileOutputStream(new File(subdirectory,fileName)));
```

"Bang! And that's it! Now it passes."

"Sweet!"

To be continued...

The code that Alphonse and Avery finished can be retrieved from:
w www.objectmentor.com/resources/articles/CraftsmanCode/Craftsman_22_BugEye.zip