The Craftsman: 11
SMCRemote Part I
What's main got to do with it?

Robert C. Martin
18 Feb, 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/SDSocketServiceR5_DanglingThreads.zip

---

My mind kept reviewing dangling threads as I absentmindedly ate my spaghetti. After lunch I returned to the lab to find Jerry waiting for me.

"Mr. C. thinks the `SocketServer` is ready to use now, and he'd like us to get to work on the `SMCRemote` application."

"Oh right!" I said. "That's what the `SocketServer` was for. We were building an application that would invoke the SMC compiler remotely, and ship the source files to the server, and the compiled files back."

Jerry looked at me expectantly and asked: "How do you think we should start?"

"I think I'd like to know how the user will use it." I responded.

"Excellent! Starting from the user's point of view is always a good idea. So what's the simplest thing the user can do with this tool?"

"He can request that a file be compiled. The command might look like this:" I drew the following on the wall:

```
java SMCRemoteClient myFile.sm
```

"Yes, that looks good to me." Said Jerry. "So how should we begin?"

The spaghetti was sitting warmly in my stomach, and I was feeling pretty confident after getting `SocketServer` to work, so I grabbed the keyboard and started typing:

```
public class SMCRemoteClient {
  public static void main(String args[]) {
    String fileName = args[0];
  }
}
```

"Excuse me!" Jerry Interrupted. "Do you have a test for that?"

"Huh? What do you mean? This is trivial code, why should we write a test for it?"

"If you don't write a test for it, how do you know you need it?"

That stopped me for a second. "I think it's kind of obvious." I said at last.

"Is it?" Jerry replied. "I'm not convinced. Lets try a different approach."

Jerry took the keyboard and deleted the code I had written. The old anger flared for a second, but I wrestled it back. After all, it was only four lines of code.

"OK, what functions do we know we need to have?" He asked.

I thought for a second and said: "We need to get the file name from the command line. I don't see how you do that without the code you just deleted."

Jerry gave me wry glance and said: "I think I do." He began to type.

First he wrote the, now familiar, test framework code:

```
import junit.framework.*;

public class TestSMCRemoteClient extends TestCase {
    public TestSMCRemoteClient(String name) {
        super(name);
    }
}
```

He compiled and ran it, making sure it failed for lack of tests, and then he added the following test:

```
    public void testParseCommandLine() throws Exception {
        SMCRemoteClient c = new SMCRemoteClient();
        c.parseCommandLine(new String[]{"filename"});
        assertEquals("filename", c.filename());
    }
```

"OK." I said. "It looks like you are going to get the command line argument in some function named `parseCommandLine`, instead of from `main`. But why bother?"

"So I can test it." said Jerry.

"But there's barely anything to test." I complained.

"Which means the test is real cheap to write." He responded.

I knew I wasn't going to win this battle. I just heaved a sigh, took the keyboard and wrote the code that would make the test pass.

```
public class SMCRemoteClient {
    private String itsFilename;

    public void parseCommandLine(String[] args) {
        itsFilename = args[0];
    }

    public String filename() {
        return itsFilename;
    }
}
```

Jerry nodded and said: "Good!, that passes."

Then he quietly wrote the next test case.

```
    public void testParseInvalidCommandLine() {
        SMCRemoteClient c = new SMCRemoteClient();
        boolean result = c.parseCommandLine(new String[0]);
        assertTrue("result should be false", !result);
    }
```

I should have known he would do this. He was showing me why it was a good idea to write the test that I thought was unnecessary.

"OK." I admitted. "I guess getting the command line argument is just a bit less trivial than I thought. It probably does deserve a test of its own." So I took the keyboard and made the test pass.

```
public boolean parseCommandLine(String[] args) {
    try {
        itsFilename = args[0];
    } catch (ArrayIndexOutOfBoundsException e) {
        return false;
    }
    return true;
}
```

For good measure, I refactored the `c` variable out, and initialized it in the `setUp` function. The tests all passed.

Before Jerry could suggest the next test case, I said: "It's possible that the file might not exist. We should write a test that shows we can handle that case."

"True." said Jerry; as he pried the keyboard from my clutches. "But let me show you how I like to do that."

```
public void testFileDoesNotExist() throws Exception {
  c.setFilename("thisFileDoesNotExist");
  boolean prepared = c.prepareFile();
  assertEquals(false, prepared);
}
```

"You see?" He explained. "I like to evaluate each command line argument in its own function, rather than mixing all that parsing and evaluating code together."

I noted that for future reference, privately rolled my eyes, took the keyboard, and made this test pass.

```
public void setFilename(String itsFilename) {
  this.itsFilename = itsFilename;
}

public boolean prepareFile() {
  File f = new File(itsFilename);
  if (f.exists()) {
    return true;
  } else
    return false;
}
```

All the tests passed. Jerry looked at me, and then at the keyboard. It was clear he wanted to drive. He seemed to be infused with ideas today, so with a fatalistic shrug I passed the keyboard over to him.

"OK, now watch this!" he said, his engines clearly revving.

```
public void testCountBytesInFile() throws Exception {
  File f = new File("testFile");
  FileOutputStream stream = new FileOutputStream(f);
  stream.write("some text".getBytes());
  stream.close();

  c.setFilename("testFile");
  boolean prepared = c.prepareFile();
  f.delete();
  assertTrue(prepared);
  assertEquals(9, c.getFileLength());
}
```

I studied this code for a few seconds and then I replied: "You want `prepareFile()` to find the length of the file? Why?"

"I think we're going to need it later." He said. "And it's a good way to show that we can deal with an existing file."

"What are we going to need it for?"

"Well, we're going to have to ship the contents of the file through the socket to the server, right?"

"Yeah."

"OK, well we'll need to know how many characters to send."

"Hmmm. Maybe. "

"Trust me. I'm the Journeyman after all."

"OK, never mind that. Why are you creating the file in the test? Why don't you just keep the file around instead of creating it every time?"

Jerry sneered and got serious. "I hate keeping external resources around for tests. Whenever possible I have the tests create the resources they need. That way there's no chance that I'll lose a resource, or that it will become corrupted."

"OK, that makes sense to me; but I'm still not crazy about this file length stuff."

"Noted. You'll see!"

So I took the keyboard and started working on making the test pass. While I was typing, it struck me as odd that I was writing all the production code, and yet the design was all Jerry's. And yet, all Jerry was doing was writing little test cases. Can you really specify a design by writing test cases?

```
public long getFileLength() {
    return itsFileLength;
}

public boolean prepareFile() {
    File f = new File(itsFilename);
    if (f.exists()) {
        itsFileLength = f.length();
        return true;
    } else
        return false;
}
```

Jerry's next test case created a mock server, and tested the ability of `SMCRemoteClient` to connect to it.

```
public void testConnectToSMCRemoteServer() throws Exception {
  SocketServer server = new SocketServer(){
    public void serve(Socket socket) {
      try {
        socket.close();
      } catch (IOException e) {
      }
    }
  };
  SocketService smc = new SocketService(SMCPORT, server);
  boolean connection = c.connect();
  assertTrue(connection);
}
```

I was able to make that pass with very little trouble:

```
public boolean connect() {
    try {
        Socket s = new Socket("localhost", 9000);
        return true;
    } catch (IOException e) {
    }
    return false;
}
```

"Great!", said Jerry. "Let's take a break."

"OK, but before we do, lets write `main()`."

"What's `main()` got to do with anything?"

"Huh? It's the main program!"

"So what. All it's going to do is call `parseCommandLine()`, `parseFile()`, and `connect()`. It'll be a long time before we have a test for that!

I left the lab and headed for the break room. I had always thought that `main()` was the first function to write; but Jerry was right. In the end, `main()` is a pretty uninteresting function.

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

www.objectmentor.com/resources/articles/CraftsmanCode/Craftsman_11_SMCRemote_1_WhatsMain.zip

The Craftsman: 12
SMCRemote Part II
Three Ugly Lines

Robert C. Martin
18 March, 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_11_SMCRemote_1_WhatsMain.zip

---

I spent my break up in the observation deck. We had a little bit of excitement as the ice shield passed through a thick patch of particles that made the ice flicker with blue flashes and transient patterns all over it's surface.

As always, Jerry was waiting for me after break. I wondered if he ever actually took any of this breaks. He beamed at me and said:

"OK, let's ship a file over the socket."

"Did you watch the Cherenkov display?" I asked.

"A beauty!" he said. I guess he *does* take his breaks. But where does he go?

"OK, lets ship a file." I said. "I'll write the test case." I grabbed the keyboard and started to type. The first thing I wrote was the code that created the file to be sent through the socket.

```
public void testSendFile() throws Exception {
    File f = new File("testSendFile");
    FileOutputStream stream = new FileOutputStream(f);
    stream.write("I am sending this file.".getBytes());
    stream.close();
}
```

"I know you like to create your data files in the test code rather than depending on them to just be there."

"Right." said Jerry. "But do you notice that you have quite a bit of duplication?"

I looked around in the test class and immediately saw that we had written almost this exact same code in the `testCountBytesInFile()` method that we wrote before our break.

"It's just a four lines of code." I said. "Hardly major duplication."

"True." Jerry replied. "But duplication should always be removed as soon as possible. If you allow duplicate code to accumulate in your project you'll find yourself with a huge source of confusion and bugs."

"OK, it's easy enough to fix." So I extracted a new function called `createTestFile()` and changed both `testCountBytesInFile()`, and `testSendFile()` to call it.

```
private File createTestFile(String name, String content)
```

```
    throws IOException  {
        File f = new File(name);
        FileOutputStream stream = new FileOutputStream(f);
        stream.write(content.getBytes());
        stream.close();
        return f;
    }
```

I ran the tests just to make sure I hadn't broken anything, and then continued writing the test. I knew that the test had to simulate `main()`, so I called all the functions that `main()` would have to call. Then I added one final call that would send the file.

```
    public void testSendFile() throws Exception {
        File f = createTestFile("testSendFile", "I am sending this file.");
        c.setFilename("testSendFile");
        assertTrue(c.connect());
        assertTrue(c.prepareFile());
        assertTrue(c.sendFile());
    }
```

"Good." Jerry said. You are anticipating that we'll need a method in the client named `sendFile()`."

"Right." I said. "This method will send the prepared file."

I turned back to the test and then got stuck. I didn't know how to proceed. How was I going to test that this file that I had created and "sent" actually got sent to the server. We didn't even *have* a server? Did I have to write the whole server before I could test this? What was I testing anyway?

I sat there for about a minute while Jerry watched expectantly. Then I turned to him and explained my dilemma.

"No, you don't need to write the server." Jerry said. "At this point all we are testing is the client's ability to *send* the file, not the servers ability to receive it. "

"How can I send the file without having a server to receive it?"

"You can create a stub server that does as little work as possible. It doesn't have to receive the file at first, it just has to acknowledge that you sent the file correctly."

"So...hmmm...something like this?"

```
        assertTrue(server.fileReceived);
```

"Yah, that ought to do." Jerry nodded. "Now why don't you make that test pass?"

I thought about this for a minute and then realized that this shouldn't be very hard. So I wrote a simple mock server that did nothing.

```
class TestSMCRServer implements SocketServer {
    public boolean fileReceived = false;
    public void serve(Socket socket) {
    }
}
```

Jerry said: "Ah, more duplication!"

I looked around and saw that before the break we had implemented a very similar mock server in the `testConnectToSMCRemoteServer()` method, so I eliminated it.

```
    public void testConnectToSMCRemoteServer() throws Exception {
        boolean connection = c.connect();
        assertTrue(connection);
    }
```

Then I started the server in the `SetUp()` method of the test, and closed it in `TearDown()`. Thus, before every test method was called, the server would be started; and then it would be closed when the test method returned.

```
protected void setUp() throws Exception {
    c = new SMCRemoteClient();
    server = new TestSMCRServer();
    smc = new SocketService(SMCPORT, server);
}

protected void tearDown() throws Exception {
    smc.close();
}
```

Finally, I wrote a dummy `sendFile()` method in `SMCRemoteClient`.

```
public boolean sendFile() {
    return false;
}
```

The tests failed as expected. I sighed and looked over at Jerry. "Now what?" I said.

"Send the file." he said.

"You mean just open the file and shove it over the socket?"

Jerry thought for a minute and said: "No, we probably need to tell the server to expect a file. So lets send a simple message and follow it with the contents of the file."

Jerry grabbed the keyboard and made the following changes to `SMCRemoteClient`. "First we have to get the streams out of the socket." He said.

```
public boolean connect() {
    boolean connectionStatus = false;
    try {
        Socket s = new Socket("localhost", 9000);
        is = new BufferedReader(new InputStreamReader(s.getInputStream()));
        os = new PrintWriter(new OutputStreamWriter(s.getOutputStream()));
        connectionStatus = true;
    } catch (IOException e) {
        e.printStackTrace();
        connectionStatus = false;
    }
    return connectionStatus;
}
```

"Then we have to make the file available for reading."

```
public boolean prepareFile() {
    boolean filePrepared = false;
    File f = new File(itsFilename);
    if (f.exists()) {
        try {
            itsFileLength = f.length();
            fileReader = new BufferedReader(
                            new InputStreamReader(new FileInputStream(f)));
            filePrepared = true;
        } catch (FileNotFoundException e) {
            filePrepared = false;
            e.printStackTrace();
        }
```

```
        }
        return filePrepared;
    }
```

"Finally, we can send the file."

```
public boolean sendFile() {
    boolean fileSent = false;
    try {
      writeSendFileCommand();
      fileSent = true;
    } catch (Exception e) {
      fileSent = false;
    }
    return fileSent;
  }

  private void writeSendFileCommand() throws IOException {
    os.println("Sending");
    os.println(itsFilename);
    os.println(itsFileLength);
    char buffer[] = new char[(int) itsFileLength];
    fileReader.read(buffer);
    os.write(buffer);
    os.flush();
  }
```

"Wow!" I said. "That was a lot of typing without testing."

Jerry looked at me sheepishly. "Yes, I'm a little nervous about that."

He hit the test button, and the test failed because `server.fileReceived` returned `false`.

"Whew!" said Jerry. "We dodged a muon pulse."

"So." I said. "You are going to precede the file with three lines. The first contains the string "Sending". The second contains the file name, and the third contains the length of the file. After that you send the file in character array."

"Right. I told you before the break that we'd need that file length."

"Hmmm. I guess so." I said.

"So now, all we have to do is receive the file in the mock server. Would you like to take a try?"

I was pretty sure I knew what to do. So I made the following changes. First I changed the test to make sure we got the filename, length, and the file contents.

```
  public void testSendFile() throws Exception {
    File f = createTestFile("testSendFile", "I am sending this file.");
    c.setFilename("testSendFile");
    assertTrue(c.connect());
    assertTrue(c.prepareFile());
    assertTrue(c.sendFile());
    Thread.sleep(50);
    assertTrue(server.fileReceived);
    assertEquals("testSendFile", server.filename);
    assertEquals(23, server.fileLength);
    assertEquals("I am sending this file.", new String(server.content));
    f.delete();
  }
```

Next I changed the mock server to parse the incoming data and make sure it was correct.

```
class TestSMCRServer implements SocketServer {
  public String filename = "noFileName";
  public long fileLength = -1;
  public boolean fileReceived = false;
  private PrintStream os;
  private BufferedReader is;
  public char[] content;
  public String command;

  public void serve(Socket socket) {
    try {
      os = new PrintStream(socket.getOutputStream());
      is = new BufferedReader(new InputStreamReader(socket.getInputStream()));
      os.println("SMCR Test Server");
      os.flush();
      parse(is.readLine());
    } catch (Exception e) {
    }
  }

  private void parse(String cmd) throws Exception {
    if (cmd != null) {
      if (cmd.equals("Sending")) {
        filename = is.readLine();
        fileLength = Long.parseLong(is.readLine());
        content = new char[(int)fileLength];
        is.read(content,0,(int)fileLength);
        fileReceived = true;
      }
    }
  }
}
```

Finally, I modified `SMCRemoteClient.connect()` to wait for the SMCR message sent by the mock server.

```
public boolean connect() {
   ...
   String headerLine = is.readLine();
   connectionStatus = headerLine != null &&
                      headerLine.startsWith("SMCR");
   ...
}
```

I didn't type this all at once. I didn't want to have to dodge another muon pulse. So I made the changes in much smaller steps, running the tests in between each step. I could tell that Jerry was impressed. He was still embarrassed about making such a big change. Eventually, when all the tests passed, I felt just a little superior, so I risked an observation.

"Jerry." I said. "This code is pretty ugly."

"What do you mean?"

"Well, sending those three lines, the filename, the length, and the string "Sending". That's ugly."

Jerry looked at me skeptically. He sat up as straight as he could and gave me a condescending look. "I suppose you know a better way."

"I think I do." I said, and I started to type...

To be continued.

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

www.objectmentor.com/resources/articles/CraftsmanCode/Craftsman_12_SMCRemote_I1_ThreeUgly Lines.zip

The Craftsman: 13 SMCRemote Part III Objects

Robert C. Martin
21 April, 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_12_SMCRemote_I1_ThreeUglyLines.zip

---

At our current τ of .045 *Destination* was still many lifetimes in the future. Each generation since *Departure* felt those lifetimes stretch inexhaustibly before them. Sometimes the feeling was depressing -- but not today. Today I used a bit of that inexhaustible time to taunt Jerry.

I stretched my hands out in front of the keyboard and cracked my knuckles. I rocked my head back and forth, pretending to work out the kinks. I paused, gazing off into the distance, putting on an air of deep contemplation. "Oh Yes!" I said, "I think I know a better way!" Jerry rolled his eyes and sighed, waiting for me to get on with it. I decided I had better not push my luck, so I started to work.

In order to write a file across the socket, Jerry had sent three lines of text first. One had the string "Sending", the next had the name of the file, and the last was length of the file. Then Jerry sent the file itself as an array of `chars`. The code looked like this:

```
private void writeSendFileCommand() throws IOException {
    os.println("Sending");
    os.println(itsFilename);
    os.println(itsFileLength);
    char buffer[] = new char[(int) itsFileLength];
    fileReader.read(buffer);
    os.write(buffer);
    os.flush();
}
```

When reading the file back in from the socket, he called `readLine` three times, once for each of the three lines he sent. He used the "Sending" string as a transaction identifier. He saved the second as the file name. The third was the length of the file. He used it to allocate an array of `chars` to use as a buffer. Then he used the length to read the appropriate number of `chars` from the socket.

```
private void parse(String cmd) throws Exception {
  if (cmd != null) {
    if (cmd.equals("Sending")) {
      filename = is.readLine();
      fileLength = Long.parseLong(is.readLine());
      content = new char[(int)fileLength];
```

```
        is.read(content,0,(int)fileLength);
        fileReceived = true;
      }
    }
  }
```

This all worked just fine, but I thought it was ugly. I was convinced there was a better way. So I started to make some simple changes. First I changed the test to read objects instead of lines:

```
  public void serve(Socket socket) {
    try {
      os = new PrintStream(socket.getOutputStream());
      is = new ObjectInputStream(socket.getInputStream());
      os.println("SMCR Test Server");
      os.flush();
      parse((String)is.readObject());
    } catch (Exception e) {
    }
  }

  private void parse(String cmd) throws Exception {
    if (cmd != null) {
      if (cmd.equals("Sending")) {
        filename = (String)is.readObject();
        fileLength = is.readLong();
        content = (char[]) is.readObject();
        fileReceived = true;
      }
    }
  }
```

Next I changed the SMCRemoteClient to write objects instead of strings.

```
  public boolean connect() {
    ...
    os = new ObjectOutputStream(smcrSocket.getOutputStream());
    ...
  }

  private void writeSendFileCommand() throws IOException {
      os.writeObject("Sending");
      os.writeObject(itsFilename);
      os.writeLong(itsFileLength);
      char buffer[] = new char[(int) itsFileLength];
      fileReader.read(buffer);
      os.writeObject(buffer);
      os.flush();
  }
```

I ran all the tests, and they worked just fine. "See?" I said. "I figured it would be better to write objects than it would be to write strings."

I looked at Jerry, but something had changed. Jerry's eyes weren't focusing. He got up and started to pace. Occasionally he would stop, look at the screen, look at me, shake his head and start pacing again. He kept mumbling something about years, experience, and stupidity. It was a little scary.

Finally he stopped, looked me square in the eye, and said: "Well, Alphonse, you've gone and done it."

"What did I *do*, Jerry?"

He stared at me for another couple of seconds. Then turned towards the turbo and said: "Follow me."

The ride down the turbo was silent. Jerry's mood was hard to read. He wasn't exactly angry, but he was certainly annoyed, and I was somehow involved with his annoyance. As we rode, silently shifting our

weight to adjust for the coriolis deflection, I tried to figure out why my simple code change would have such a profound effect upon him.

I followed Jerry into a lounge on one of the low-g levels. Apprentices weren't normally allowed below .49g. I hadn't been reading the wall markers on the way down, but this one felt to be less than .4g. Inside the lounge were five other journeymen programmers. Jerry introduced me to the group. I made sure to remember everyone's name: Johnson, Jasmine, Jason, Jasper, and Jennifer. Jerry told me to stand in the center of the lounge, while he and everyone else sat on couches around me. Then Jerry turned to the group and with a grimace he said:

"Well, it's happened. I believe Alphonse is the first apprentice this year to *Micah* his Journeyman." I felt my heart skip a beat, and my eyes got wide. It was such a simple thing! I hadn't expected *this*!

"Have any of you been Micahed so far this year?" asked Jerry. There was a murmur around the room, but everyone shook their heads. Apparently nobody had.

Jasmine looked at me long and hard. She locked her eyes on mine and said to Jerry: "OK, Jer (She pronounced it Jair). Tell us your story."

Jerry sighed and shrugged. He made a visible effort to gather himself together, and then began to speak.

"As you know, Mr. C. asked me to get the SMCRemote stuff working." They all grunted and nodded. Apparently they all knew about it. "Alphonse and I had spent a day on the SocketServer exercise; and he had done very well."

Another shock: SocketServer had been an *exercise?!!?*

"Once we got it working we started putting together the client portion of SMCRemote. One of our first test cases was to ship a file from the client to the server over the socket." There were more nods and grunts around the room.

Jerry was getting visibly more nervous. He squirmed in his seat and avoided eye contact. "I set up a file-send transaction by shipping three text lines followed by an array of chars. The first line was the transaction id, the second was the file name, and the third was the file length." More nods. This wasn't surprising to any of them. "Clearly this was just a simple way to get the tests to pass so we could refactor into a better form." More nods, more agreeing mumbles.

"And then..." Jerry paused. "And then, Alphonse said he thought he had a -- er -- a better idea."

There was silence in the room. Jasmine's eyes were still locked on mine, but her look shifted from appraisal to speculation. One by one I could sense the gaze of the other journeymen land on me. *What was the big deal?* Why were they claiming a Micah for me?

It was Johnson who broke the spell.

"You don't me to tell us..."

Out of the corner of my eye I could see that Jerry was nodding. Nodding about *what*?

I couldn't stand it any more. I broke away from Jasmine's gaze, looked each of the other journeymen briefly in the eye, and then said: "All I did was suggest that we ship objects instead of strings! I don' t see that as a Micah!"

Jennifer stepped up to me and said: "Yes, that's *all* you did. And, no, I don't suppose *you* think all that much of it. But to us, it's a big deal."

"But why?"

"Because", said Jeremy, "the most important trait of a good programmer is the ability to think abstractly. Very few programmers can actually do that. You have just proven that you can."

I was incredulous. "It was just an object." I said.

"Exactly." said Jennifer. They all nodded earnestly.

I shook my head. "Well then if this is such a good thing -- a Micah, and all -- why is Jerry so annoyed?"

"Oh that!" laughed Jasmine. "Jerry came down here at his last break and told us all about the file length issue you had with him. He was sure you were going to be impressed with him when you saw how that file length just fell into position in the file-send transaction. He was anticipating how awed you would

be."

"Yeah", said Jasper, "and you went and showed him how the length was irrelevant."

"I did?"

Jerry stood up and said: "Think about it Alphonse, if you are sending the character array as an object, why do you need to send the length separately? These guys are going to be ribbing me about this for the next six months."

"We sure are!" said Jennifer enthusiastically. "Every time we review any of his code we'll be asking him where the file length parameter is!" She giggled as Jerry grimaced and hung his head.

"You see, Alphonse", said Jasmine, "not only did you make a worthwhile leap of abstraction, your solution was *simpler* than Jerry's. What's more, it was simple in a way that invalidated Jerry's anticipated need for the file length. You *Micahed* him!"

I was beginning to understand. At least I wasn't in any kind of trouble...

"I think", said Jasmine, "that an event like this calls for a change of partners. Jerry, I'll swap apprentices with you. You take Andy, and I'll work with Alphonse for a few days."

...or was I?

To be continued.

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

www.objectmentor.com/resources/articles/CraftsmanCode/Craftsman_13_SMCRemote_III_Objects.zip

The Craftsman: 14 SMCRemote Part IV Transactions

Robert C. Martin
19 May, 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_14_SMCRemote_IV_Transactions.zip

---

"OK Hotshot, let's see what you've got."

The intensity of Jasmine's stare nailed me to my chair. "W-What do you mean?" I stammered.

"Look, Hotshot, you aren't going to embarrass *me* the way you did Jerry. I mean Jerry's good and all, but I'm a *lot* better."

"I wasn't trying to embarrass anyone, I..."

"Yah, sure. Let's just get on with this, shall we? What's the next change you expect to make?"

We were sitting in the lab, looking at the code that Jerry and I had just written. I had showed Jasmine how I had changed Jerry's code that sent strings across the socket to send objects instead.

"I -- er -- I don't know. I just thought sending objects would be --uh -- better than strings." God, she made me nervous. She is stunningly beautiful, in an austere kind of way. Intensity spills out of her every look and action. And those deep black eyes!

Jasmine rolled those eyes now. "THINK! Hotshot, THINK! You aren't just going to wrap a couple of strings and integers into an object are you? What does that wrapping imply? What can you *do* with it?"

"I, uh..." If she'd stop pinning me with those eyes I might be able to THINK. I simultaneously wanted to be anywhere else, and nowhere else. It felt like being torn in two.

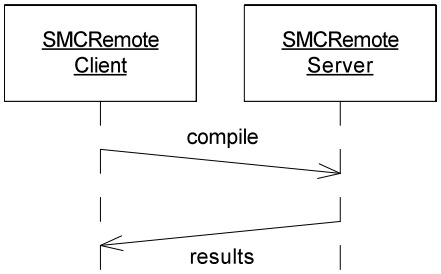
I closed my eyes and mentally recited a quick calming mantra. I forced the tension out of my mind and body. Within a few seconds I was able to consider the question she had provoked me with.

The test case we were working on verified that we could send a file over the socket. The code that sent the file looked like this:

```
private void writeSendFileCommand() throws IOException {
    os.writeObject("Sending");
    os.writeObject(itsFilename);
    os.writeLong(itsFileLength);
    char buffer[] = new char[(int) itsFileLength];
    fileReader.read(buffer);
    os.writeObject(buffer);
    os.flush();
}
```

But *why* were we sending a file? We were sending it to the SMCRemoteServer to be compiled. Then the server would send back the compiled file. Why had Jerry sent the "Sending" string first? He said that it was to alert the server that a file was coming. But we don't want to tell the server that a file is coming, we want to tell the server to compile a file, and send back the results.

I was thinking hard, but some part of my brain continued to recite the calming mantra. Almost as if in a trance, I walked to the wall and drew the following diagram:



I looked over at Jasmine, and noticed a smile flicker across her severe expression. "I like the way you are thinking Hotshot, but don't stop there."

There were four pieces of data being sent to the server. The file name, the file length, the file contents, and the "Sending" string. Why were these being sent separately? They were all part of the same packet of information, the same -- Transaction! That was it!

I shook myself out of my calming trance and made the following change to the test:

```
public void testCompileFile() throws Exception {
  File f = createTestFile("testSendFile", "I am sending this file.");
  c.setFilename("testSendFile");
  assertTrue(c.connect());
  assertTrue(c.prepareFile());
  assertTrue(c.compileFile());
  Thread.sleep(50);
  assertTrue(server.fileReceived);
  assertEquals("testSendFile", server.filename);
  assertEquals(23, server.fileLength);
  assertEquals("I am sending this file.", new String(server.content));
  f.delete();
}
```

Then I modified the old `sendFile` function as follows:

```
public boolean compileFile() {
  boolean fileSent = false;
  char buffer[] = new char[(int) itsFileLength];
  try {
    fileReader.read(buffer);
    CompileFileTransaction cft =
      new CompileFileTransaction(itsFilename, buffer);
    os.writeObject(cft);
    os.flush();
    fileSent = true;
  } catch (Exception e) {
    fileSent = false;
  }
  return fileSent;
}
```

I looked over at Jasmine, and saw that she was following along *very* closely. I couldn't read her expression through her raw intensity, but I was pretty sure I was on a good track.

Next I wrote the `CompileFileTransaction` class.

```
public class CompileFileTransaction implements Serializable {
  private String filename;
  private char contents[];
  public CompileFileTransaction(String filename, char buffer[]) {
    this.filename = filename;
    this.contents=buffer;

  }
  public String getFilename() {
    return filename;
  }
  public char[] getContents() {
    return contents;
  }
}
```

This allowed the project to compile. Of course the tests failed big time. So I made the following changes to the mock server in the test.

```
  public void serve(Socket socket) {
    try {
      os = new PrintStream(socket.getOutputStream());
      is = new ObjectInputStream(socket.getInputStream());
      os.println("SMCR Test Server");
      os.flush();
      parse(is.readObject());
    } catch (Exception e) {
    }
  }

  private void parse(Object cmd) throws Exception {
    if (cmd != null) {
      if (cmd instanceof CompileFileTransaction) {
        CompileFileTransaction cft = (CompileFileTransaction) cmd;
        filename = cft.getFilename();
        content = cft.getContents();
        fileLength = content.length;
        fileReceived = true;
      }
    }
  }
```

These changes made all the tests pass. I looked over at Jasmine and asked: "Is this what you had in mind?"

"Well, Hotshot, it's a start. It's certainly a lot better than converting all the data elements to strings the way Jerry did it. And it's also a lot better than shipping each data element as its own individual object."

"How would you make it better?" I asked, hoping to get her focus off me, and onto the project.

"Later." she said. "Right now, let's complete the transaction. You've got to get the client to accept server's response."

"That shouldn't be hard." I said. So I added the following three lines to the end of the `testCompileFile` test case:

```
    File resultFile = new File("resultFile.java");
    assertTrue("Result file does not exist", resultFile.exists());
    resultFile.delete();
```

I ran the test and verified that it failed.

"After we call `compileFile` the results should be written into a file." I explained to Jasmine. "At the moment I don't care what's in that file I just want to make sure that it gets created."

"So how are you going to create it?" she challenged.

"I'll show you." I said, and I made the following change to the mock server in the test.

```
  private void parse(Object cmd) throws Exception {
    if (cmd != null) {
      if (cmd instanceof CompileFileTransaction) {
        CompileFileTransaction cft = (CompileFileTransaction) cmd;
        filename = cft.getFilename();
        content = cft.getContents();
        fileLength = content.length;
        fileReceived = true;
        CompilerResultsTransaction crt =
          new CompilerResultsTransaction("resultFile.java");
        os.writeObject(crt);
        os.flush();
      }
    }
  }
```

Then I made this compile by creating a skeleton of the `CompilerResultsTransaction` class.

```
public class CompilerResultsTransaction implements Serializable {
  public CompilerResultsTransaction(String filename) {

  }

  public void write() {

  }
}
```

Of course the test still failed. So I made the following change to `compileFile`.

```
  public boolean compileFile() {
    boolean fileCompiled = false;
    char buffer[] = new char[(int) itsFileLength];
    try {
      fileReader.read(buffer);
      CompileFileTransaction cft =
        new CompileFileTransaction(itsFilename, buffer);
      os.writeObject(cft);
      os.flush();
      Object response = is.readObject();
      CompilerResultsTransaction crt = (CompilerResultsTransaction)response;
      crt.write();
      fileCompiled = true;
    } catch (Exception e) {
      fileCompiled = false;
    }
    return fileCompiled;
  }
```

And then finally I fleshed out the transaction.

```
public class CompilerResultsTransaction implements Serializable {
  private String filename;
  public CompilerResultsTransaction(String filename) {
```

```
    this.filename = filename;
  }

  public void write() throws Exception {
    File resultFile = new File(filename);
    resultFile.createNewFile();
  }
}
```

"Good enough for now." Jasmine said, and leaned back in her chair. "I'm going to go on break in a few minutes while you get the `CompilerResultsTransaction` to actually write the file instead of just create it. Also, I'd like you to do a bit of cleanup of the code. There's a lot of cruft left behind from when you and Jerry were messing around with sending strings and integers. But before I go, I'd like your thoughts on that `instanceof` you used in the mock server."

"That was the simplest thing I could think of to check whether or not the incoming object was a `CompileFileTransaction`." I said. "Is there something wrong with it?"

She stood up, preparing to leave. She looked down at me and said: "No, nothing horribly wrong. But do you think that's what the real server is going to do? Do you think the real server will have a long `if/else` chain of `instanceof` expressions to parse the incoming transactions?"

"I hadn't really thought that far ahead." I said.

"No." she said flatly. "I imagine you hadn't." And she strode out of the room.

The room felt distinctly empty without her in it, as though my presence didn't count at all. I let out a long sigh, and shook my head. I could see that the working with Jasmine was going to be both exhausting and educational in many different ways.

One thing I was sure of, I *really* hated being called Hotshot. I sighed again, turned back to the computer, and started working on the tasks he gave me.

To be continued.

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

www.objectmentor.com/resources/articles/CraftsmanCode/Craftsman_13_SMCRemote_III_Objects.zip

The Craftsman: 15
SMCRemote Part V
Ess Are Pee

Robert C. Martin
18 June, 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_14_SMCRemote_IV_Transactions.zip

---

The last thing I saw, before the door slid shut, was Jasmine's long black hair swaying and bouncing to the rhythm of her purposeful stride. As her saturating presence drained from the room, I felt my lungs release the breath I hadn't known they were holding. My eyes lost their focus, and for several quasi-conscious minutes I just sat and blindly gazed at the blur that was the door.

"Whew!" I said to myself. "Working with Jasmine is *not* going to be easy."

"Easy or not, " I replied to myself, *"she asked you to get that file written and to clean up the code. So you'd better get cracking."*

I sighed. "Agreed, agreed." I agreed with myself. "I can just imagine what Jasmine would say if I didn't have something to show her. She'd say: *'Some Hotshot you turned out to be! What did you do, just sit here the whole time doing nothing?'*"

"Okay, Jasmine, okay. So, what should we do first?"

"Well, Hotshot, we have to get the `CompilerResultsTransaction` *to carry the contents of a file from the server to the client; and then write that file on the client."*

"Oh right." I said. "The compiler will create an output file on the server, and we have to move it to the client. This is just what we did in the `CompileFileTransaction`."

```
public class CompileFileTransaction implements Serializable {
  private String filename;
  private char contents[];
  public CompileFileTransaction(String filename, char buffer[]) {
    this.filename = filename;
    this.contents=buffer;

  }
  public String getFilename() {
    return filename;
  }
  public char[] getContents() {
    return contents;
  }
}
```

"We opened and read the file in the `compileFile` method, and then constructed and passed the file name and the array of chars into the constructor to the `CompileFileTransaction`."

```
public boolean compileFile() {
    char buffer[] = new char[(int) itsFileLength];
    try {
      fileReader.read(buffer);
      CompileFileTransaction cft =
        new CompileFileTransaction(itsFilename, buffer);
```

"Right, Hotshot, that's just what we did. Is there something you don't like about that?"

"I don't like having to write the same code twice. Jerry made a big point out of not duplicating code."

"Jerry's not exactly the sharpest knife in the drawer, Hotshot".

"Maybe not, but I think he was right about that. So I think I want to write a class that carries a file across the socket."

"Something like a `FileCarrier`?"

"Yeah, that's a good name for it!"

"OK, Hottie, write a test for it."

"Hottie? -- Er, OK. How about this?"

```
public class FileCarrierTest extends TestCase {
  public void testAFile() throws Exception {
    final String TESTFILE = "testFile.txt";
    final String TESTSTRING = "test";
    createFile(TESTFILE, TESTSTRING);
    FileCarrier fc = new FileCarrier(TESTFILE);
    fc.write();
    assertTrue(new File(TESTFILE).exists());
    String contents = readFile(TESTFILE);
    assertEquals(TESTSTRING, contents);
  }
}
```

"Yes, very good, Hot Stuff, keep going."

"OK, beautiful. Here are the two utility functions..."

```
  private String readFile(final String TESTFILE) throws IOException {
    BufferedReader reader = new BufferedReader(new FileReader(TESTFILE));
    String line = reader.readLine();
    return line;
  }

  private void createFile(final String TESTFILE,
                          final String TESTSTRING) throws IOException {
    PrintWriter writer = new PrintWriter(new FileWriter(TESTFILE));
    writer.println(TESTSTRING);
    writer.close();
  }
```

"...and here is the stubbed implementation of `FileCarrier` that will make this test compile and fail.

```
public class FileCarrier {
  public FileCarrier(String fileName) {
  }

  public void write() {
  }
```

```
}
```

"So now, Jazzy-wazzy, all we have to do to make the test pass is read the file in the constructor and write it in the `write()` function."

"Not yet, over-temp, first run the test to make sure it will fail."

"Jay-girl, there's no implementation in `FileCarrier`. Of course it's going to fail.

"Well, exotherm, you and I both know that. But does the program?"

"I love it when you make me run a test! OK, here goes."

*** the test passes***

"Huh? What? How can that be? Do you understand that Jazz?"

"Gosh, boiling point, I sure don't. How can the test pass when there's nothing in `FileCarrier` *to..."*

"Oh. Egad. Are we dumb or what. We never deleted the test file."

"Ah, yes, I see that now, Mr. tepid breath. So why don't you fix that?"

"OK, here."

```
  public void testAFile() throws Exception {
    final String TESTFILE = "testFile.txt";
    final String TESTSTRING = "test";
    createFile(TESTFILE, TESTSTRING);
    FileCarrier fc = new FileCarrier(TESTFILE);
    new File(TESTFILE).delete();
    fc.write();
    assertTrue(new File(TESTFILE).exists());
    String contents = readFile(TESTFILE);
    assertEquals(TESTSTRING, contents);
  }
```

"OK, great, now it fails. But -- latent-heat -- can you make it pass?"

"Sure I can, JJ, just watch me!"

```
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();
  }
}
```

"Yee Ha! electron-volt, you are cooking now!"

"Why, thanks! Gorgeous! But you haven't seen anything yet. Watch how I integrate the `FileCarrier` into the `CompileFileTransaction`! That should turn your head."

```
public class CompileFileTransaction implements Serializable {
```

```
  FileCarrier sourceFile;
  public CompileFileTransaction(String filename) throws Exception {
    sourceFile = new FileCarrier(filename);
  }
  public String getFilename() {
    return sourceFile.getFileName();
  }
  public char[] getContents() {
    return sourceFile.getContents();
  }
}
```

"And now I'll change the `compileFile` function to use the new `CompileFileTransaction`!"

```
      CompileFileTransaction cft = new CompileFileTransaction(itsFilename);
      os.writeObject(cft);
      os.flush();
      Object response = is.readObject();
      CompilerResultsTransaction crt = (CompilerResultsTransaction)response;
      crt.write();
```

"And now I'll run all the tests and ... see? They all pass!"

"Oh, fever man, stop it, stop it, you're making me crazy!"

"It's all part of my plan, sweet-eyes, all part of my plan. Now, watch with bated breath as I put the `FileCarrier` into the `CompilerResultsTransaction`!"

"Ooooh!"

```
public class CompilerResultsTransaction implements Serializable {
  private FileCarrier resultFile;
  public CompilerResultsTransaction(String filename) throws Exception {
    resultFile = new FileCarrier(filename);
  }

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

"Gasp!"

"That's right! And look how masterfully I change the test to use the new transaction!"

```
  private void parse(Object cmd) throws Exception {
    if (cmd != null) {
      if (cmd instanceof CompileFileTransaction) {
        CompileFileTransaction cft = (CompileFileTransaction) cmd;
        filename = cft.getFilename();
        content = cft.getContents();
        fileLength = content.length;
        fileReceived = true;
        TestSMCRemoteClient.createTestFile("resultFile.java", "Some content.");
        CompilerResultsTransaction crt =
          new CompilerResultsTransaction("resultFile.java");
        os.writeObject(crt);
        os.flush();
      }
    }
  }
```

"More, Alphonse, Please, More!"

You want more Jasmine? I can give you more. Lots more. You see Jasmine, I *know* about objects. Oh yes, I know about them. `FileCarrier` is an *Object* Jasmine. Do you see how it can be used in more than one place, Jasmine? Do you see how it encapsulates a single responsibility, Jasmine? Do you? Do you know the **Single Responsibility Principle** Jasmine? Have you heard of it? Have you studied it? Well I have. Do you know what it says, Jasmine? It says that a class should have one and only one reason to change, Jasmine. It says that all the functions and variables of a class should work together towards a single goal, Jasmine. It says that a class should not try to accomplish more than one goal...Are you *listening* Jasmine?

"Yes, Alphonse. I'm listening. Please, don't stop!"

"You'd better not ask me to stop! You see, Jasmine, those of us in the know call this principle the SRP. That's ESS ARE PEE Jasmine.

"ESS ARE PEE, Alphonse. ess are pee."

"Remember the `compileFile` function Jasmine, remember how it used to read the file and pass a `char` array into the `CompileFileTransaction`? You asked me what I didn't LIKE about that function. Well I'll TELL you what I didn't LIKE about it, Jasmine; it was VIOLATING the SRP! It had TWO reasons to change, instead of ONE. It depended BOTH on the details of reading a file, AND on the policy of building and sending transactions. That's TOO MUCH RESPONSIBILITY Jasmine."

"You're scaring me, Alphonse."

"You SHOULD be scared, Jasmine. I'm..."

"Oh! Alphonse!"

~ ~ ~

Then several things happened at once:

1. I noticed that the door was open.
2. I noticed that the chair next to me was empty.
3. I heard the echoes of my falsetto imitation of Jasmine still reverberating off the walls.
4. I saw Jasmine standing in the doorway. Her eyes were cold steel.

To be continued...maybe.

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

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

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

The Craftsman: 17
SMCRemote Part VII
Call the Guards

Robert C. Martin
17 Aug, 2003

...Continued from last month. You can download last month's code from:

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

---

Dear Diary, my first week as an apprentice for Mr. C. is over. I've learned a lot in that week, and I've also made a real mess of things. Here's a recap: On Monday Jerry had me write a program to generate prime numbers. On Tuesday he had me write a program to generate prime factors. Wednesday was spent getting the SocketServer working. Thursday we started working on SMCRemoteClient, and I Micahed Jerry. I met Jasmine that afternoon in the Journeymen's lounge. Friday was the most embarrassing day of my life. I haven't seen Jasmine (Ms J) since we finished working Friday afternoon. What a roller coaster of a week. (Diary, just what is a roller coaster anyway?)

I spent the weekend pretty depressed. I was sure that I was going to be transferred to sanitation & recycling, reactor cleanup, or mildew scrubbing. I barely bothered to get dressed at eat.

Anyway, now it's Monday evening, and I'm writing to tell you about the first day of my second week. It didn't turn out badly at all. In fact, I'm starting to feel optimistic again.

I woke up on Monday full of dread. After breakfast I shambled over to the workroom expecting to see Ms. J waiting for me at my workstation. Instead I saw a plumpish middle-aged woman with shoulder length blonde hair, and a grandmother smile that spoke of younger years. When she looked at me her eyes twinkled and she gave me a wide grin. "You must be Alphonse." She said, holding out her hand. "My name is Jean. I've heard a great deal about you. Are you hungry? Young men like you are always hungry. I've got a nice sandwich here in my basket if you'd like it, or perhaps you'd like an apple or a banana.

There was, indeed, a basket on the floor next to her. It looked like it contained quite a bit more than just sandwiches and bananas. I shook her hand and said: "Yes Ma'am -- I mean, no Ma'am -- I mean -- Yes I am Alphonse, and no thank you for the sandwich, and I'm -- er -- pleased to meet you."

She looked at me sternly, yet with a big motherly grin. "Now we'll have none of that Ma'am nonsense. I'm certainly not anybody's Ma'am. The very idea! You just call me Jean, dear."

"Uh, well, thank you Jean. Can you tell me where Ms. J is?"

"Who dear?"

"er -- Jasmine. She and I are supposed to be working together."

"Goodness me, I've never heard her called Ms. J before. Did she ask you to call her that? I don't think anyone else calls her anything but Jasmine. Such a lovely girl, isn't she? Anyway, dear, Mr. C. asked her to work on a different project for the time being, and so I'll be working with you from now on."

"You?" I was caught completely unprepared for this. "Oh." I said intelligently. "Uh..."

"Now you sit right down here, dear, and let me tell you what I've been thinking."

"Thinking?"

"Yes dear! I've been looking over this program of yours for the last half-hour -- I like to get to work early you know -- not that you need to get here any earlier than usual -- I know a growing boy needs his sleep and breakfast. Anyway I've been quite pleased with this program. You've got a very pleasant suite of tests, and the code is quite easy to read and, really quite nicely structured, I'd say. But there's one thing that puzzles me dear."

"Uh -- puzzles?"

"Yes, dear! I've been looking all through this program and I can't find the main function. When were you going to write it dear?"

"Uh, well, Jerry said..."

"Oh let me just guess what Jerry said. Jerry's a nice lad, dear, but sometimes I think he could use a few more clues about things. But I shouldn't be saying anything bad about anyone. Jerry is a fine programmer, dear, and you just never mind about what I said. Now, let's write that main function. Would you like to start? My fingers are feeling a bit stiff this morning. Take my advice and don't get old, dear."

Dazed by the sheer inundation of words I took the keyboard and began typing:

```
public void main(
```

"Oh, now, dear! How long have you been working here? You've got to write a test first, dear. You can't just go off writing the main function! Where would we be if everyone just wrote the functions without writing the tests first? I can tell you where -- In a pickle, that's where. No, dear, you delete that and write a test first.

She took a pair of knitting needles out of her basket and began working on a project of some kind. It looked vaguely like a shawl. She started softly humming a simple tune while I deleted the characters I had written and started over.

How do you test the main function? I guess the best answer is simply to call it and then make sure it did what it was supposed to do. The main function interprets the command line options, so calling it is just a matter of passing in the right options. So I started typing again:

```
public void testMain() throws Exception {
  SMCRemoteClient.main(new String[]{"myFile.sm"});
```

Jean looked up from her knitting and said: "That's nice dear, but where does `myFile.sm` come from? We can't just have that laying around, can we? No, we'll have to create it right here, won't we dear? And don't forget to delete it when you are done, dear. Nothing worse than a bunch of old files hanging around, I always say."

So I continued typing:

```
public void testMain() throws Exception {
  File f = createTestFile("myFile.sm", "the content");
  SMCRemoteClient.main(new String[]{"myFile.sm"});
  f.delete();
  File resultFile = new File("resultFile.java");
  assertTrue(resultFile.exists());
  resultFile.delete();
}
```

Jean had finished another row of her shawl while I typed. She looked up just as I finished and said: "Now, dear, that's just fine, but I do think that `main` might not have enough time to finish executing before that delete takes effect. Remember, dear, you've got all those socket threads running, and one of them could easily still be going by the time `main` returns. No, dear, don't do anything about it now; just keep it in mind. Now I think you'd better write `main`, don't you?"

I thought to myself that this was a pretty sharp old lady, and I started writing the main function.

```
public static void main(String[] args) {
  SMCRemoteClient client = new SMCRemoteClient();
  client.setFilename(args[0]);
  if (client.prepareFile())
    if (client.connect())
      if (client.compileFile())
        client.close();
      else {  // compileFile
        System.out.println("failed to compile");
      }
    else { // connect
      System.out.println("failed to connect");
    }
  else { // prepareFile
    System.out.println("failed to prepare");
  }
}
```

"Oh my, now nice that looks! I think it's very clever how you commented those `else` statements. Though I wonder if it wouldn't be more readable if you inverted the sense of the `if` statements and used them as guards. No, dear, don't change it yet. Let's see if it works first. It's never worth making lots of changes until you know whether the program work or not, don't you agree? First make it work, *then* make it right."

So I ran the test, and it worked first time.

"Oh, that's just splendid!" She said while glancing up from her knitting. "Now, let's change that `if` statement."

So I changed the function so that the `if` statements were guards.

```
public static void main(String[] args) {
  SMCRemoteClient client = new SMCRemoteClient();
  client.setFilename(args[0]);
  if (!client.prepareFile()) {
    System.out.println("failed to prepare");
    return;
  }
  if (!client.connect()) {
    System.out.println("failed to connect");
    return;
  }
  if (!client.compileFile()) {
    System.out.println("failed to compile");
    client.close();
    return;
  }
  client.close();
}
```

Jean put her knitting down into her basket and looked carefully at the code. "Well I do think that looks a little better, though I can't say I care for that duplicated close statement. And the violation of single/entry, single/exit is a bit bothersome. Still, it's better than all that indentation, don't you agree? Of course we could change those three functions to throw exceptions, but then we'd have to catch them, and that would be a bother. No, let's leave it like that for the time being. Now, I think it's time for a break, don't you? Would you like to carry my basket to the break room for me? I'm always over-packing it and it gets so heavy after awhile you know."

To be continued...

The code that Alphonse and Jasmine finished can be retrieved from:
www.objectmentor.com/resources/articles/CraftsmanCode/ Craftsman_17_Call_the_Guards.zip

The Craftsman: 18
SMCRemote Part VIII
Slow and Steady

Robert C. Martin
24 Sept, 2003

...Continued from last month. You can download last month's code from:
www.objectmentor.com/resources/articles/CraftsmanCode/Craftsman_17_Call_the_Guards.zip

---

"All right then dear, I think it's time we started to work on the server part of this application, don't you? After all the client portion seems to be working as well as we can tell, and a client without a server just seems so lonely. Don't you agree?"

Jean had just settled her ample behind into her seat. As she spoke she reached into her bag and pulled out her knitting. The shawl (or whatever it was) had grown considerably during our break (during which I had learned all about Jean's children, grandchildren, nephews, and nieces)

"Er, server?" Two words were all I could ever seem to muster in response to one of Jean's vocal deluges.

"Yes dear, the server. The server is going to have to receive that file you've been so diligently sending, save it to disk, and then compile it with SMC. Remember dear, we are building a remote compiler for SMC, the State Machine Compiler. Now, dear, what's the first test case we should write?

What test case indeed? I thought furiously about this. The server side of the pair of programs had to listen at a socket and receive `CompileFileTransaction` objects. These objects contain encapsulated files in `FileCarrier` sub-objects. The server needs to write these encapsulated files to a safe place on the disk, compile them, and then send the results back to the client in a `CompilerResultsTransaction` object.

My first worry was how to compile the files. Somehow we needed to invoke the compiler and get it to compile the saved file.

"Uh, could we write a test case for invoking the compiler?"

Jean smiled the way a grandmother smiles at an infant taking its first steps. "Why of course, dear, that would be a lovely place to start. Do you know how to invoke the compiler? Let's see, I think we just have to build the right command line and then invoke it. Now what was that command line syntax? It's been so long since I used SMC, I don't quite remember. Hear, dear, type this command: `java smc.Smc`. That's right dear, now what does that error message say? Class def not found? Oh yes, we forgot the classpath. Don't get old dear; you start forgetting things. Type this: `java -cp C:/SMC/smc.jar smc.Smc`.[1] That's better, now what does that message say?"

The message on the screen was usage error that described all the command line arguments for SMC. "Er, it's a usage message."

---

[1] You can download `smc.jar` from:
`http://www.objectmentor.com/resources/downloads/smc_java`

"So it is, dear, so it is. And it looks to me as though we want to invoke SMC with the following command: `java -cp C:/SMC/smc.jar smc.Smc -f myFile.sm`. Don't you agree dear? Why don't you write a test case that will generate that command line?"

So I created a new unit test class named `SMCRemoteServerTest`.

```
public class SMCRemoteServerTest extends TestCase {
  public void testBuildCommandLine() throws Exception {
    assertEquals("java -cp C:/SMC/smc.jar -f myFile.sm",
      SMCRemoteServer.buildCommandLine("myFile.sm"))  ;
  }
}
```

And I wrote the `SMCRemoteServer` class too.

```
public class SMCRemoteServer {
  public static String buildCommandLine(String file) {
    return null;
  }
}
```

"Very good, dear. The test fails, as it should. You've been learning your lessons well. Now it should be a simple matter to make that test pass, don't you think?

"Er -- sure."

```
  public static String buildCommandLine(String file) {
    return "java -cp C:/SMC/smc.jar smc.Smc -f " + file;
  }
```

"That's fine dear, now run it for me, won't you? Good. Oh! Why, the test failed dear, what could be the matter?"

The message on the screen was: `expected:<.....> but was:<...smc.Smc ...>`. I looked carefully at the code and realized that I had forgotten the `smc.Smc` in the test case. "Er, I guess I wrote the test wrong."

"So you did, so you did. Yes, sometimes we do write the tests wrong. Bugs can occurr anywhere. That's why it's always a good idea to write both tests and code. That way you are writing everything twice. I have a nephew who does accounting in ship stores, and he always enters every transaction into two distinct ledgers. What did he call that? Oh, yes, *dual entry bookkeeping*. He says it helps him prevent and track down errors. And that's just what we're doing, isn't it dear. We write every bit of functionality twice. Once in a test, and once in code. And it *does* help us to prevent and track down errors doens't it dear?"

While she was droning, I fixed the test case. Jean is a nice old lady, and pretty smart too, but my ears were ringing from the continual chatter.

```
  public void testBuildCommandLine() throws Exception {
    assertEquals("java -cp C:/SMC/smc.jar smc.Smc -f myFile.sm",
      SMCRemoteServer.buildCommandLine("myFile.sm"))  ;
  }
```

This made the test pass.

"Wonderful, dear, wonderful; but the code is a bit messy, don't you think? Let's clean it up before we go any further. Clean your messes before they start, I always say."

I looked at the code and didn't see anything particularly messy. So I looked back and Jean and said: "Er, where would you like to start?"

"My goodness dear; why it's as plain as the nose on your face! That `buildCommandLine` function needs a bit of straightening. That string in the return statement is just full of unexplained assumptions that are sure to confuse the next person to read this code. We don't want to confuse anyone, do we? I should

say we don't. Here, dear, give me the keyboard."

She set down her knitting (is that the beginnings of a sleeve) and with considerable effort took control of the keyboard. Her typing was slow but deliberate, as if each keystroke caused her pain. Eventually she changed the return statement as follows:

```
return "java -cp " + "C:/SMC/smc.jar" + " " + "smc.Smc" + " -f " + file;
```

Then she ran the test, and it passed.

"There, dear, do you see now? We need to replace some of those strings with constants that explain how the command line is built. Do you want to do it dear, or shall I?"

Her pained expression was answer enough. I took the keyboard and added the constants I thought she was after.

```
public class SMCRemoteServer {
  private static final String SMC_CLASSPATH = "C:/SMC/smc.jar";
  private static final String SMC_CLASSNAME = "smc.Smc";

  public static String buildCommandLine(String file) {
    return "java -cp " +
      SMC_CLASSPATH + " " +
      SMC_CLASSNAME +
      " -f " + file;
  }
}
```

"Yes, dear, that's just what I was after. Don't you think that others will make a bit more sense out of it now? I know I would if I were them. Now dear, why don't you and I go to the break room and settle our brains?"

Something inside me snapped.

"Jean, we've barely gotten anything done!"

"Why dear, whatever do you mean, we've gotten quite a bit done."

I wasn't thinking about whom I was talking to. I kept right on blathering. "We've only gotten one stupid function done. If we keep moving at this snail's pace, Mr. C is going to transfer us to the husbandry deck to clean out the Dribin cages!"

"Really dear, why ever would you think such a thing? I've been working in this department for well over thirty years, dear, and nobody has ever suggested that I should be cleaning up after Dribins."

She stopped for a moment as if composing her thoughts. Then she looked at me with a kindly, but stern, expression on her face.

"Alphonse dear, the only way to get done with a program quickly, is to do the very best job you can with it. If you rush, or if you take shortcuts, you will pay for it later in debugging time. Mr. C. is paying you for your time, dear, and he wants the very best work you can do. The code is your product, dear, and Mr. C. doesn't pay for poor products. He doesn't want to hear that you saved a day by being sloppy because he knows that he'll have to pay dearly for that sloppiness later. What he wants is the very best code you can write. He wants that code to be as clean, expressive, and well tested as you can make it. And, dear, Mr. C. knows that if you do that, you'll be done with it quicker than all those young fools who think they can go faster by cutting corners. No, dear, we aren't going too slow. We're moving along at just the right pace to get done as quickly as possible. Now, let's go get a cup of tea, shall we?"

To be continued...

The code that Alphonse and Jean finished can be retrieved from:
www.objectmentor.com/resources/articles/CraftsmanCode/Craftsman_17_Slow_And_Steady.zip

The Craftsman: 19
SMCRemote Part IX
Tolerance

Robert C. Martin
16 Oct, 2003

...Continued from last month. You can download last month's code from:
www.objectmentor.com/resources/articles/CraftsmanCode/Craftsman_18_Slow_And_Steady.zip

---

Jean took me down the turbo to the journeyman's lounge at .36g in the gamma arm. She said the low-g did wonders for her joints. When we got there I noticed Jerry, Jasmine, and a few other journeymen clustered at a table. Jean noticed them too and started walking over to them. I followed reluctantly.

"Hello my dears!" she exclaimed as though she hadn't seen them in weeks. They all greeted her with similar warmth. Then she excused herself and trotted over to a vacant head, leaving me to face my two previous mentors alone.

"Hello Jerry; Ms. J. How are you?"

Jerry looked perplexed. "What's this Ms. J. stuff about?"

Jasmine impatiently brushed this aside and said: "Drop the Ms. J stuff hotshot. We've had enough of that." Her eyes were as disarming as ever, and I found myself unable to answer her intelligently.

A coffee servitor rolled by. Grateful for the distraction I grabbed a cup and then faced my mentors again.

"How's Jean treating you hotshot?" Jasmine asked.

Jerry leaped in with: "Yeah, how'd you score a gig with *her*?"

After barely being able to say more than three words at a time for the whole morning with Jean, my frustrations, came pouring out.

"I didn't score anything, she just showed up this morning. Frankly working with her is a little frustrating. She's always calling me "Dear"; and she talks a *lot* and takes a *lot* of breaks. We barely get anything done. I'm not sure I want to keep working with her."

Jasmine and Jerry looked at me, then at each other, then starting laughing. Jerry settled down quickly; but Jasmine couldn't seem to gain control of her self. Every time she looked at me she spurted out new laughs that sounded like a sea lion calling to its mate.

"What?" I said.

Jerry took me aside while Jasmine continued to emit barks of laughter like shrill hiccoughs. "Alphonse, you don't understand how lucky you are. Any of us here would give our eyeteeth to work with Jean. I know she's a little quirky; but don't let the grandmother facade fool you. She's one of the best there is, and you'll learn a lot from her."

I was dumbfounded; but I couldn't respond because Jean chose that moment to toddle back from the head.

"My Goodness, I feel much better now."

This was more information than I wanted, so I changed the subject by offering to call the coffee servitor.

"Oh, my no! Dear. Why that'd just bring me right back here in ten or twenty minutes. No, I think we ought to go back up to our uncomfortable .6g lab and keep working on SMCRemote, don't you? We've got a lot of ground yet to cover today, and we certainly aren't getting it done down here, are we? ...What is Jasmine making that terrible sound for? Sounds like some animal dying. Jasmine, drink some water dear...

We took the turbo back out to the rim and walked back to the lab. The nearly doubled g slowed Jean down quite a bit. She was much spryer down in the lounge.

After we got situated back at our workstation, she said: "Now then Alphonse dear, I think we'd better see if we can execute that command we just built. What do you think?"

I had been wondering about how we were going to do this, so I agreed. "Er, yes."

"Good dear. Now, since the command we are about to execute invokes the SMC compiler, I think the first thing we need to do is create a simple source file for that compiler to read. So let's write a test case that creates this file, then invokes the compiler, and then checks to see that the compiler has created the proper output files."

She had pulled out her knitting again, and showed no sign that she wanted to touch the keyboard, so I took it and started to type:

```
  public void testExecuteCommand() throws Exception {
    File sourceFile = new File("simpleSourceFile.sm");
    PrintWriter pw = new PrintWriter(new FileWriter(sourceFile));
    pw.println("
  }
```

I didn't know how to continue, so I looked expectantly at her.

"That's fine dear. Here, I'll type in the SMC syntax for you."

```
  public void testExecuteCommand() throws Exception {
    File sourceFile = new File("simpleSourceFile.sm");
    PrintWriter pw = new PrintWriter(new FileWriter(sourceFile));
    pw.println("Context C");
    pw.println("FSMName F");
    pw.println("Initial I");
    pw.println("{I{E I A}}");
    pw.close();
  }
}
```

"What does that mean, Jean?"

"Well, dear, for our purposes it means that the compiler will create a file named F.java. That's all you really need to know about it for now. However, once you get back to your quarters it would behoove you to look up the SMC document and read it. I wrote it quite a few years ago dear, and I still think it's one of my better documents. It's called: 'Care and Feeding of the State Map Compiler'[1]. Now then, let's see if we can execute that compile command."

I wasn't quite sure how to execute the command; but I'd learned from Jerry and Jasmine that it's often better just to write the calls that express my intention. So I continued to type:

```
  public void testExecuteCommand() throws Exception {
    File sourceFile = new File("simpleSourceFile.sm");
    PrintWriter pw = new PrintWriter(new FileWriter(sourceFile));
    pw.println("Context C");
    pw.println("FSMName F");
    pw.println("Initial I");
```

[1] You can get this document from: http://www.objectmentor.com/resources/downloads/bin/smcJava.zip

```
    pw.println("{I{E I A}}");
    pw.close();

    String command = SMCRemoteServer.buildCommandLine("simpleSourceFile.sm");
    assertEquals(true, SMCRemoteServer.executeCommand(command));

    File outputFile = new File("F.java");
    assertTrue(outputFile.exists());
    assertTrue(outputFile.delete());
    assertTrue(sourceFile.delete());
  }
```

"Very nicely done, Alphonse. I think that captures the test case quite admirably. We haven't written `executeCommand` yet, but we can certainly express how we want it to be called, can't we. Now, let's stub that function out and watch this test case fail. It's always fun to watch them fail, don't you think?"

So I clicked on `executeCommand` and selected "Create Method", and my IDE created just the method stub I wanted. And, sure enough, the test failed.

```
public class SMCRemoteServer {
 ...
  public static boolean executeCommand(String command) {
    return false;
  }
}
```

"All right dear, now let's make that test pass."

Jean was sounding tired again. I knew she'd want another break soon. It was almost lunchtime, so I was hoping she could hold out until then. I quickly searched the javadocs to find out how to execute a command. Jean saw what I was doing and said:

"It's in the Runtime class, dear. There's a method named exec."

Sure enough, it was right where she said. So I typed the code that I thought would work.

```
  public static boolean executeCommand(String command) {
    Runtime rt = Runtime.getRuntime();
    try {
      rt.exec(command);
      return true;
    } catch (IOException e) {
      return false;
    }
  }
```

But when I ran the test it failed to find the output file. I looked in the directory and, sure enough, `F.java` was not there.

"Why is this failing, Jean?"

She looked up from her knitting and said: "You didn't wait for the process to finish, dear. When you execute a command it creates a new process that runs concurrently with yours. You have to wait for it to complete before you exit from `executeCommand`."

I examined the JavaDoc for `Runtime.exec`, and found that it returned a `Process` object. I also found that you could wait for the `Process` object to complete, and that you could query it for its exit status. So I made the following changes.

```
  public static boolean executeCommand(String command) {
    Runtime rt = Runtime.getRuntime();
    try {
      Process p = rt.exec(command);
```

```
      p.waitFor();
      return p.exitValue() == 0;
    } catch (Exception e) {
      return false;
    }
  }
```

This time the test passed.

"That wasn't that hard." I said.

"Of course not dear. All we are doing is running a command. Of course it'll get a bit more complicated when we have to capture the messages that the compiler usually prints on the console. We'll have to bind to the standard output, and standard error of the `Process`. But let's do that after lunch dear, I'm starting to get hungry, aren't you?"

I sighed and stood up. It still seemed to me that we were going slowly. Though in hindsight I see that in my first half day with Jean we had finished up the client, and gotten the server to run a compile. We hadn't spent any time debugging. Perhaps we were moving faster than I thought.

Jean put the knitterbot back in her bag and held out a sweater. "Hear, dear, try this on."

If nothing else, this "gig" with Jean was going to teach me a new kind of tolerance.

To be continued...

The code that Alphonse and Jean finished can be retrieved from:
www.objectmentor.com/resources/articles/CraftsmanCode/Craftsman_19_Tolerance.zip

The Craftsman: 20 SMCRemote Part X Backslide.

Robert C. Martin
28, Nov, 2003

...Continued from last month. You can download last month's code from:
www.objectmentor.com/resources/articles/CraftsmanCode/Craftsman_19_Tolerance.zip

---

I returned to the lab after lunch, but Jean was not there. There was neither note, nor email from her; and her knitting basket was nowhere to be seen. After waiting a few minutes, I decided to sit down and continue working on the `SCMRemoteServer`.

So far the server doesn't serve anything. It's just a couple of functions that build and execute the SMC command line. It seemed reasonable to me that the server code should open a socket and accept connections from the `SMCRemoteClient` that we wrote earlier.

I was pretty frustrated with our pace today. We'd worked the whole morning and only gotten two small functions, and their attendant unit tests written. I felt like making *progress*. So I grabbed the keyboard and began to type.

"First," I thought, "this server needs to serve something. So lets use the `SocketServer` class that Jerry and I built last week."

Getting the server to serve was pretty simple. I just needed to write a constructor that created a `SocketService` object and passed in a `SocketServer`. Then the `SocketServer.serve` method would automatically be called as soon as `SMCRemoteClient` tried to connect.

```
public SMCRemoteServer() throws Exception {
  service = new SocketService(9000, new SocketServer(){
        public void serve(Socket socket) {
          // SMCRemoteClient has connected.
        }
      });
}
```

I looked in the `SMCRemoteClient` code and saw that the client expects a string to be sent upon connection. That string should begin with "SMCR". So I wrote that string in the `serve()` method.

```
        public void serve(Socket socket) {
          // SMCRemoteClient has connected.
          try {
            ObjectOutputStream os =
                  new ObjectOutputStream(socket.getOutputStream());
            os.writeObject("SMCR");
          } catch (IOException e) {
```

```
    }
  }
```

Next the server needs to read a `CompileFileTransaction` from the client. It needs so write the files contained in that transaction, invoke the compiler, and then return the resulting files in a `CompilerResultsTransaction`. This didn't seem like a hard thing to write so...

```
public void serve(Socket socket) {
  // SMCRemoteClient has connected.
  try {
    ObjectOutputStream os =
          new ObjectOutputStream(socket.getOutputStream());
    os.writeObject("SMCR");
    ObjectInputStream is =
          new ObjectInputStream(socket.getInputStream());
    CompileFileTransaction cft =
          (CompileFileTransaction)is.readObject();
    String filename = cft.getFilename();
    cft.sourceFile.write();
    String command = buildCommandLine(filename);
    executeCommand(command);
    //OK, what file do I put into the Result?
  } catch (Exception e) {

  }
}
```

Hmmm. Compiling the file didn't seem very difficult, but what output file should I put into the result transaction? What is its name? I remember from this morning that Jean had coached me in writing a test for invoking a compile. I looked at that test and found that the input file has a `.sm` suffix, and the output file as a `.java` suffix. So all I had to do is replace "`.sm`" with "`.java`" in the filename.

```
public void serve(Socket socket) {
  // SMCRemoteClient has connected.
  try {
    ObjectOutputStream os =
          new ObjectOutputStream(socket.getOutputStream());
    os.writeObject("SMCR");
    ObjectInputStream is =
          new ObjectInputStream(socket.getInputStream());
    CompileFileTransaction cft =
          (CompileFileTransaction)is.readObject();
    String filename = cft.getFilename();
    cft.sourceFile.write();
    String command = buildCommandLine(filename);
    executeCommand(command);
    //Figure out the file name.
    String compiledFile = filename.replaceAll("\\.sm", ".java");
    CompilerResultsTransaction crt =
          new CompilerResultsTransaction(compiledFile);
    os.writeObject(crt);
    socket.close();
  } catch (Exception e) {

  }
}
```

This looked like it should work. All I needed to do was to start up the server and run the client. That should be pretty simple. So I created a source file in my directory named F.sm, just like the one that Jean

had me create this morning.

```
Context C
FSMName F
Initial I
{I{E I A}}
```

And then I wrote a simple main function in `SMCRemoteServer`.

```
public static void main(String[] args) throws Exception {
  SMCRemoteServer server = new SMCRemoteServer();
}
```

And then I ran it. And it just hung there, waiting for a client to connect. Now *this* was exciting! I was getting something *done*! So next I ran the client with "`F.sm`" as its argument. And it *RAN!* Or rather it exited after several seconds with a normal exit code, and without any error messages either from the client or the server. Cool!

But what had it done? I couldn't immediately tell. So I checked the directory and found an `F.java` file sitting there! It had the right date on it, so it must have been created by my run of the client. Way cool! I opened the file and it looked like generated Java code. It even said it was generated by SMC. My code worked!

It had been a couple of weeks since I had felt this good; since before working with Jerry as a matter of fact. *This* was what programming was all about! I was filled with the rush of getting code working. I was *invincible*! I got up and did a little jig around my seat chanting "Oh yes! Oh yes! I'm a programmer. Oh yes!"

Jerry must have been nearby, because he came into the lab at that point. "Hay, Alphonse, what's are you celebrating?"

"Oh, hi Jerry! Look at this! I just got the `SMCRemoteServer` working!"

"Really? That's great Alphonse." The look on his face was funny -- as if he didn't believe me. So I showed him. I showed him how the server was sitting there running. I ran the client again. I showed him that an `F.java` file with the right date was created. I even showed him that it contained generated java code.

Jerry looked at me almost fearfully, and then nervously glanced at the door. "Alphonse, where are your tests?"

"Jerry, you don't need unit tests for something this simple. Look, it was just a dozen lines of code or so. Jerry, I made *progress* here. I got something *done*. And it didn't have to take all day! I think you guys waste too much time on all those unit tests!"

Jerry just stared at me for a few seconds, as if he couldn't quite process what I was telling him. Then he shut the door to the lab and sat down next to me.

"Alphonse, has Jean seen this?"

"No, she hasn't come back from lunch yet."

"Delete it, Alphonse."

"Delete what?"

"Delete the code you just wrote. It's worthless."

I had half expected this kind of reaction from someone. I drew my mouth into a sneer and said: "Oh, come on Jerry! It works! How could it be worthless?"

"Are you sure it works, Alphonse?"

"You saw it for yourself!"

"Are YOU sure it works?"

"Of course it works. The `F.java` file is proof!"

"Alphonse, why is the date of the `F.sm` file exactly the same as the date on the `F.java` file?

"What?" I looked at the directory, and sure enough they were identical to the second. But that didn't make any sense. I had written the `F.sm` file by hand several minutes ago. Why did it have the same date as the `F.Java` file that was created seconds ago when I demonstrated the system to Jerry? I stared at it for awhile, and then admitted that I didn't know.

"Delete the code, Alphonse. You don't understand it."

"Aw, come one Jerry, I understand it. I just can't figure out why the dates don't look right. It works Jerry...it works." But I wasn't as sure any more. *Why was that date different?*

"Alphonse, by any chance did you happen to run the client and server in the same directory?"

"Uh..." I hadn't specified a different directory for them so I guessed they must have been running in the same place. "...I guess so."

"So the client and the server were both reading and writing the same files in the same directory?"

"...oh..."

Jerry nodded grimly. "Yeah."

I shook my head. "OK, Jerry, you've got a point. I don't understand everything that's going on. But look, there's an `F.java` file there. *Something* had to be working!"

"So what? You don't know what is and what isn't working. You don't understand what you've done. The things that appear to be working may be working by accident. Is it possible, for example, that that `F.java` file was left over from a previous unit test?"

It *was* possible! Yikes, Jean and I had caused the system to write an `F.java` file this morning! "Damn! Yes, it's *possible* -- but it's not very likely!"

"Delete the code, Alphonse. It's crap."

I just stared at him. I had made *progress* damn it! Now he wanted me to delete all that progress. But he was right. I didn't understand this code. And I had no tests that would demonstrate, step by step, that everything was working as it should. If my code was working (and now I was really starting to wonder just how much of it was working) it was working more by accident than by design.

"Delete the code, Alphonse. You don't want Jean to see this. Right now she things the world of you, and this would be very disappointing for her."

My resolve finally failed. My shoulders fell, my head hung, and I reached over and deleted the code.

Jerry walked out of the room, shaking his head.

A few minutes later, Jean walked in. "Hello Alphonse dear. I'm sorry I'm late, but I got into a conversation with an old friend of mine and we started comparing pictures of our grandchildren. I love showing the pictures of my grandchildren. Have you seen them dear? Oh, never mind dear, we've got work to do you and I. What have you been up to while I was detained?"

"Nothing, Jean, I was just waiting for you."

To be continued...

The code that Alphonse and Jean finished can be retrieved from:
www.objectmentor.com/resources/articles/CraftsmanCode/Craftsman_20_Backslide.zip

The Craftsman: 21 SMCRemote Part XI Patchwork.

Robert C. Martin
14 December 2003

...Continued from last month. You can download last month's code from:
www.objectmentor.com/resources/articles/CraftsmanCode/Craftsman_20_Backslide.zip

---

"Well now, Alphonse Dear, where do you think we should run the compile?"

I was still reeling from having just deleted all that code I wrote, so I missed the question.

"I'm sorry Jean, what did you say?"

"Well, we can't just run the SMC compiler any old place, can we dear? No, we've got to run it in a new directory. We have to copy all the incoming files into that directory, run the compile, and then send all the resulting files back to the caller."

"How many resulting files are there?" I asked. "I thought that SMC just produced a `.java` file."

"The Java code generator does dear. But the C++ code generator creates a `.h` and a `.cc` file. Other generators might produce other kinds of files. I'm afraid that it's a bit unpredictable. Anyway, dear, we want the compiler to run in a clean environment, and that means an empty directory with just the input `.sm` file present."

I thought about this for a second, and realized that she was right. I was starting to be very glad that I had deleted that old code. "So we should have a special directory to run the compiles in?"

"Oh, more than that dear. I think we should create a working directory as soon as we receive a `CompileFileTransaction`. Then we should write the input file into that directory. It'll be the only file there, and so things will be nice and clean, don't you agree? And then we can run the compiler, delete the input file, and gather up any remaining files and put them into the `CompilerResultsTransaction` for shipment back to the client. Oh my, this sounds like such *fun*. Why don't you write a little unit test that makes sure we can create a new directory, dear?"

I took the keyboard and started to type. While I was typing, Jean pulled a stack of brightly colored 6" cloth squares from her bag and started to sew them together.

```
public void testMakeWorkingDirectory() throws Exception {
  File workingDirectory = SMCRemoteServer.makeWorkingDirectory();
  assertTrue(workingDirectory.exists());
  assertTrue(workingDirectory.isDirectory());
}
```

"Oh, that's fine dear. First you create the directory, and then you make sure that it exists, and that it is, in fact, a directory. Now make that fine test pass."

I know Jean doesn't mean to be condescending, but every word grates on my nerves. I squared my

shoulders and wrote the code to make the test pass. First I clicked on the missing function and had my IDE add it for me. Then I filled it in with the following code.

```
public static File makeWorkingDirectory() {
  File workingDirectory = new File("workingDirectory");
  workingDirectory.mkdir();
  return workingDirectory;
}
```

This code passed the test on the first try, of course, so I sat back and waited for Jean's next instruction.

"What are you waiting for dear?"

"I'm waiting for you to tell me what to do next."

"Are you? My goodness, I'm just a foolish, talkative old lady. Don't you have any ideas of your own? A bright young man such as yourself, you must be full of ideas. What do *you* think we should do next?

This was even more condescending than before. Jasmine and Jerry seemed to think that it was some kind of honor working with Jean; but I was finding it pretty painful.

"We should probably delete the directory when we are done with it." I said grumpily.

"I think that's a splendid idea. We'll have to make sure that the entire directory is deleted along with any files inside it."

So I wrote the following unit test while Jean continued sewing squares together.

```
public void testDeleteWorkingDirectory() throws Exception {
  File workingDirectory = SMCRemoteServer.makeWorkingDirectory();
  File someFile = new File(workingDirectory, "someFile");
  someFile.createNewFile();
  assertTrue(someFile.exists());
  SMCRemoteServer.deleteWorkingDirectory(workingDirectory);
  assertFalse(someFile.exists());
  assertFalse(workingDirectory.exists());
}
```

Jean looked up from her sewing and said: "Very nice, Alphonse. That's just what we want."

So then I wrote the following code:

```
public static void deleteWorkingDirectory(File workingDirectory) {
  workingDirectory.delete();
}
```

But this didn't pass the test. The test complained that `someFile` still existed.

"I think you have to delete all the files in the directory before you can delete the directory itself, Alphonse. I've never understood why they made that rule. I think it would be better if they believed you when you said you wanted the directory deleted. Oh well, I guess you'll have to loop through all the files and delete them one by one, dear."

So I wrote the following:

```
public static void deleteWorkingDirectory(File workingDirectory) {
  File[] files = workingDirectory.listFiles();
  for (int i = 0; i < files.length; i++) {
    files[i].delete();
  }
  workingDirectory.delete();
}
```

This made the tests pass. However I didn't like it. "Jean, this works, but what if there are

subdirectories?"

"Yes, dear, we can't be sure that one of the SMC code generators won't create a subdirectory or two. I think you'd better augment the test to handle that case."

So I changed the test as follows:

```
public void testDeleteWorkingDirectory() throws Exception {
  File workingDirectory = SMCRemoteServer.makeWorkingDirectory();
  File someFile = new File(workingDirectory, "someFile");
  someFile.createNewFile();
  assertTrue(someFile.exists());

  File subdirectory = new File(workingDirectory, "subdirectory");
  subdirectory.mkdir();
  File subFile = new File(subdirectory, "subfile");
  subFile.createNewFile();
  assertTrue(subFile.exists());

  SMCRemoteServer.deleteWorkingDirectory(workingDirectory);
  assertFalse(someFile.exists());
  assertFalse(subFile.exists());
  assertFalse(subdirectory.exists());
  assertFalse(workingDirectory.exists());
}
```

Now the test failed as expected. The `subFile` was not getting deleted. So next I changed the code to make the test pass.

```
public static void deleteWorkingDirectory(File workingDirectory) {
  File[] files = workingDirectory.listFiles();
  for (int i = 0; i < files.length; i++) {
    if (files[i].isDirectory())
      deleteWorkingDirectory(files[i]);
    files[i].delete();
  }
  workingDirectory.delete();
}
```

"Oh that's just fine, dear, just fine. Very nicely done. Now, what next Alphonse?"

Something had been nagging at me, so I said: "Jean, this is a server isn't it?"

"Yes, dear. It waits on a socket for our client code to connect to it."

"How many clients might there be?"

"Oh, there could be dozens, perhaps more."

"Then if we get two clients connecting at the same time, we'll try to create the subdirectory twice, won't we?"

"Why yes, dear, I suppose that's true. That could cause quite a bit of trouble for us couldn't it. We'd have more than one session copying files into the directory, running compiles in the directory, copying files out of the directory, and deleting the directory too. That would be terrible wouldn't it? How do you think we should solve this Alphonse?"

"What if each working directory had a unique name? That way each of the incoming connections would have it's own directory to work in."

"That's just a superb idea, Alphonse. Why don't you write the test cases for that?"

I thought it was a superb idea too, so I wrote the following test.

```
public void testWorkingDirectoriesAreUnique() throws Exception {
  File wd1 = SMCRemoteServer.makeWorkingDirectory();
```

```
    File wd2 = SMCRemoteServer.makeWorkingDirectory();
    assertFalse(wd1.getName().equals(wd2.getName()));
  }
```

This test failed as expected. To make it pass I needed some way to create unique names. "I suppose I could use the time of day to generate a unique name." I said more to myself than to Jean.

"Yes, why don't you try that dear?"

So I wrote the following code.

```
  public static File makeWorkingDirectory() {
    File workingDirectory = new File(makeUniqueWorkingDirectoryName());
    workingDirectory.mkdir();
    return workingDirectory;
  }

  private static String makeUniqueWorkingDirectoryName() {
    return "workingDirectory" + System.currentTimeMillis();
  }
```

The test passed, but I saw a funny look in Jean's face. So I hit the test button again. She nodded while the test passed again. I hit the test button several more times, and sure enough there was an occasional failure.

"Yeah," I said, "if the two calls to `makeUniqueWorkingDirectoryName` are too close together, then `currentTimeMillis` returns the same time and the names aren't unique."

"Oh dear." said Jean.

"I suppose I could just use a static integer and increment for each name, but then the names would start over each time we restarted the server. That might cause collisions too."

"Good thinking dear." said Jean.

"Why don't we use both techniques." and I stopped talking and started typing.

```
public class SMCRemoteServer {
...
  private static int workingDirectoryIndex = 0;
...
  private static String makeUniqueWorkingDirectoryName() {
    return "workingDirectory" +
           System.currentTimeMillis() + "_" +
           workingDirectoryIndex++;
  }
...

}
```

Now the tests worked repeatedly and I was quite satisfied with myself. I was certain Jean would want a break soon, and we had gotten something done. On the other hand, what we got done was something very different from the code that I deleted before Jean came in.

"Alphonse."

"Yes, Jean."

"Would you mind deleting all those left over working directories dear? They're causing quite a big of clutter in our development directory. I just hate clutter, don't you? I hate it in my code, I hate it in my room, and I hate it in my directories."

I looked at the directory and noticed that there were dozens and dozens of left over working directories. I hung my head as I realized that my tests had not been cleaning up after themselves. I sheepishly made the required changes.

```
public void testMakeWorkingDirectory() throws Exception {
  File workingDirectory = SMCRemoteServer.makeWorkingDirectory();
  assertTrue(workingDirectory.exists());
  assertTrue(workingDirectory.isDirectory());
  SMCRemoteServer.deleteWorkingDirectory(workingDirectory);
}

public void testWorkingDirectoriesAreUnique() throws Exception {
  File wd1 = SMCRemoteServer.makeWorkingDirectory();
  File wd2 = SMCRemoteServer.makeWorkingDirectory();
  assertFalse(wd1.getName().equals(wd2.getName()));
  SMCRemoteServer.deleteWorkingDirectory(wd1);
  SMCRemoteServer.deleteWorkingDirectory(wd2);
}
```

"That's much better dear. Thank you. Now, I think it's time for a break, don't you?"

"Sounds good to me."

Jean had sewn 16 squares together into a 4X4 grid.

To be continued...

The code that Alphonse and Jean finished can be retrieved from:
www.objectmentor.com/resources/articles/CraftsmanCode/Craftsman_21_Patchwork.zip

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

The Craftsman: 23
SMCRemote Part XIII
Raggedy.

Robert C. Martin
19 February 2004

...Continued from last month. You can download last month's code from:
www.objectmentor.com/resources/articles/CraftsmanCode/Craftsman_22_BugEye.zip

---

20 Feb 2002, 14:00

Avery looked at me with a smug smile on his pimply face. His bushy red eyebrows added intensity to the look. He had gained confidence over the last hour, and his stammering had disappeared. "OK, so now the server executes compiles in an empty directory. Now I think its time to test the whole server from end to end."

"You mean you want to open a socket with the server, send in a `CompileFileTransaction` and see if the appropriate `CompilerResponseTransaction` comes out?"

"Precisely, Alphonse. Precisely." As he gained confidence he started speaking with an odd formality. It was a bit strange, but also a bit fun. Playing along I said: "I see, Avery. Then I suggest that you compose the appropriate test case, and then I'll endeavor to satisfy it."

"Excellent suggestion, Alphonse. Shall I proceed?"

"Please do."

Avery faced the screen, making a show of stretching his fingers and squaring his shoulders before touching the keyboard. He typed:

```
public void testServerEndToEnd() throws Exception {

}
```

He paused with mock intensity and said: "Do you agree with the name?"

"Indeed!" I replied. I was getting into this. He continued to type while he talked.

"Our first goal should be to instantiate the `SMCRemoteServer` and bind it to some port number that is convenient for our test case."

```
public void testServerEndToEnd() throws Exception {
   SMCRemoteServer server = new SMCRemoteServer(999);
}
```

"I appreciate the intent of that code, but notice that it fails to compile. Apparently there is no constructor that takes an `int`."

"Well spotted, Alphonse. Well spotted. We must therefore create just such a constructor."

```
public class SMCRemoteServer {
   …
 public SMCRemoteServer(int port) {
 }
   …
}
```

"Nicely done, Avery. I suggest that this test will pass in its current form."

"I believe you may be correct, Alphonse. Shall we try?"

"Yes, I believe we should."

Avery pushed the test button and the green bar flashed on the screen indicating that the test had passed.

"As expected." Avery said with a self-satisfied tone. "Next we'll have our test connect to that socket like so:"

```
public void testServerEndToEnd() throws Exception {
  SMCRemoteServer server = new SMCRemoteServer(999);
  Socket client = new Socket("localhost", 999);
}
```

"I expect, Avery, that this will fail."

"I agree, Alphonse. We never created the server socket." Pushing the test button quickly yielded a red bar on the screen. We looked at each other and nodded, pleased with our mutual prescience.

I held my hands out in front of me, as a polite gesture requesting Avery to slide the keyboard in my direction. He did so with a flourish and a knowing look.

"Thank you Avery. To make this test pass we'll use a utility named `SocketService` that Jerry and I wrote last week."

```
public SMCRemoteServer(int port) throws Exception {
  SocketService service = new SocketService(port, new SocketServer() {
    public void serve(Socket theSocket) {
      try {
        theSocket.close();
      } catch (IOException e) {
      }
    }
  });
}
```

"Nicely done, Alphonse. Shall I run the test?"

"Certainly!" I slid the keyboard back to him.

Again we smiled and nodded – making a show of it -- as the green bar flashed on the screen.

"Now then," harrumphed Avery, "The server should respond with a connection message."

"Yes indeed it should, Avery." I said, remembering back to last Thursday. "Jerry and I decided that the message would be a string that begins with SMCR."

"Well then, that's easy enough to test."

```
public void testServerEndToEnd() throws Exception {
  SMCRemoteServer server = new SMCRemoteServer(999);
  Socket client = new Socket("localhost", 999);
  ObjectInputStream is = new ObjectInputStream(client.getInputStream());
  String header = (String) is.readObject();
  assertTrue(header.startsWith("SMCR"));
```

```
}
```

"Yes, that looks quite proper. Quite proper." I said accepting the keyboard as Avery brandished it in my direction. I ran the test, and noted with satisfaction as it failed due to an `EOFException`. Now then, to make this pass we merely create the appropriate string and send it out the socket– like so!"

```
public SMCRemoteServer(int port) throws Exception {
  SocketService service = new SocketService(port, new SocketServer() {
    public void serve(Socket theSocket) {
      try {
        ObjectOutputStream os =
          new ObjectOutputStream(theSocket.getOutputStream());
        os.writeObject("SMCR");
        theSocket.close();
      } catch (IOException e) {
      }
    }
  });
}
```

The test produced a green bar, and we repeated the smile and nod ritual. I passed the keyboard back to Avery holding it as though I were passing a fencing foil to my opponent at the start of a match. "Your weapon sir."

"At your disposal, sir." He responded. "I believe we must now prepare to send the `CompileFileTransaction`. This will require us to first write a source file containing code that the SMC compiler can compile."

"Ah, indeed, Jean and I created such a file just this morning. The code is in the `testExecuteCommand()` function. We should be able to extract it into a function of its own named `writeSourceFile()`."

```
private File writeSourceFile(String theSourceFileName) throws IOException {
  File sourceFile = new File(theSourceFileName);
  PrintWriter pw = new PrintWriter(new FileWriter(sourceFile));
  pw.println("Context C");
  pw.println("FSMName F");
  pw.println("Initial I");
  pw.println("{I{E I A}}");
  pw.close();
  return sourceFile;
}
```

"Excellent! Thank you Alphonse. So now I can create a `CompileFileTransaction`, send it to the server, and expect a `CompilerResultsTransaction` back – right?"

"I'd say that was correct, Avery."

```
public void testServerEndToEnd() throws Exception {
  SMCRemoteServer server = new SMCRemoteServer(999);
  Socket client = new Socket("localhost", 999);
  ObjectInputStream is = new ObjectInputStream(client.getInputStream());
  ObjectOutputStream os = new ObjectOutputStream(client.getOutputStream());
  String header = (String) is.readObject();
  assertTrue(header.startsWith("SMCR"));
  File sourceFile = writeSourceFile("mySourceFile.sm");
  CompileFileTransaction cft = new CompileFileTransaction("mySourceFile.sm");
  os.writeObject(cft);
  os.flush();
  CompilerResultsTransaction crt =
```

```
    (CompilerResultsTransaction) is.readObject();
    assertNotNull(crt);
  }
```

With the red bar from the failing test on the screen, Avery said. "Alphonse, do you agree that this code expresses my intent."

"Yes, I think it expresses it very well. And now if you'll hand me the keyboard, I'll make it pass."

```
    public void serve(Socket theSocket) {
      try {
        ObjectOutputStream os =
          new ObjectOutputStream(theSocket.getOutputStream());
        ObjectInputStream is =
          new ObjectInputStream(theSocket.getInputStream());
        os.writeObject("SMCR");
        os.flush();
        CompileFileTransaction cft =
          (CompileFileTransaction) is.readObject();
        CompilerResultsTransaction crt = new CompilerResultsTransaction();
        os.writeObject(crt);
        os.flush();
        theSocket.close();
      } catch (Exception e) {
      }
    }
```

"And sure enough, pass it does."

"Well done Alphonse!"

"Thank you Avery. There really isn't anything to it."

"In that case, I'll give you one more challenge." He took the keyboard and began to type again.

```
  public void testServerEndToEnd() throws Exception {
    SMCRemoteServer server = new SMCRemoteServer(999);
    Socket client = new Socket("localhost", 999);
    ObjectInputStream is = new ObjectInputStream(client.getInputStream());
    ObjectOutputStream os = new ObjectOutputStream(client.getOutputStream());
    String header = (String) is.readObject();
    assertTrue(header.startsWith("SMCR"));
    File sourceFile = writeSourceFile("mySourceFile.sm");
    CompileFileTransaction cft = new CompileFileTransaction("mySourceFile.sm");
    os.writeObject(cft);
    os.flush();
    CompilerResultsTransaction crt =
      (CompilerResultsTransaction) is.readObject();
    assertNotNull(crt);
    File resultFile = new File("F.java");
    assertFalse(resultFile.exists());
    crt.write();
    assertTrue(resultFile.exists());
  }
```

"Aha!" I said. "You actually want me to invoke the compiler, eh?"

"That I do, my dear Alphonse. That I do."

"Then prepare yourself for compilation, young Avery."

```
    public void serve(Socket theSocket) {
      try {
        ObjectOutputStream os =
          new ObjectOutputStream(theSocket.getOutputStream());
```

```
        ObjectInputStream is =
          new ObjectInputStream(theSocket.getInputStream());
        os.writeObject("SMCR");
        os.flush();
        CompileFileTransaction cft =
          (CompileFileTransaction) is.readObject();
        String command = buildCommandLine(cft.getFilename());
        CompilerResultsTransaction crt = compile(cft, command);
        os.writeObject(crt);
        os.flush();
        theSocket.close();
      } catch (Exception e) {
      }
    }
```

We both leaned over the monitor as I pushed the test button.

Green Bar. Smile. Nod.

"We've done well, Avery. But I think it's time we cleaned this code up a bit."

"I quite agree, Alphonse, it has gotten a little, shall we say, raggedy?"

Together we worked through the new server code and split it up into a batch of smaller methods that worked together to tell the story. When we were done it looked like this; and all the tests still passed.

```
public SMCRemoteServer(int port) throws Exception {
  SocketService service =
    new SocketService(port, new SMCRemoteServerThread());
}

  private static class SMCRemoteServerThread implements SocketServer {
  private ObjectOutputStream os;
  private ObjectInputStream is;
  private CompileFileTransaction cft;
  private CompilerResultsTransaction crt;

  public void serve(Socket theSocket) {
    try {
      initializeStreams(theSocket);
      sayHello();
      readTransaction();
      doCompile();
      writeResponse();
      theSocket.close();
    } catch (Exception e) {
    }
  }

  private void readTransaction() throws IOException, ClassNotFoundException {
    cft = (CompileFileTransaction) is.readObject();
  }

  private void initializeStreams(Socket theSocket) throws IOException {
    os = new ObjectOutputStream(theSocket.getOutputStream());
    is = new ObjectInputStream(theSocket.getInputStream());
  }

  private void writeResponse() throws IOException {
    os.writeObject(crt);
    os.flush();
  }

  private void sayHello() throws IOException {
```

```
    os.writeObject("SMCR");
    os.flush();
  }

  private void doCompile() throws Exception {
    String command = buildCommandLine(cft.getFilename());
    crt = compile(cft, command);
  }
}
```

"OK, that's much better. I think it's time for a break, don't you?" I said to Avery.

"Yeah, I guess we've been at it for a couple of hours. Let's go to the game room and play LandCraft."

To be continued...

The code that Alphonse and Avery finished can be retrieved from:
www.objectmentor.com/resources/articles/CraftsmanCode/Craftsman_23_Raggedy.zip