The Craftsman: 8
SocketService 3

Robert C. Martin
14 Nov 2002

...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/SDSocketServerR2_ManyConnections.zip

---

The turbo on level 17 was down again, so I had to use the ladder-shaft. While sliding down the ladder I got to thinking about how remarkable it was to use tests as a design tool. Lost in my thoughts I got a bit careless with the coriolis forces and bumped my elbow against the shaft. It was still stinging when I rejoined Jerry in the lab.

"Are you ready to try out the test that sends the 'hello' message through the sockets?" he asked.

"Sure." I said. We uncommented the `TestSendMessage` method.

```
  public void testSendMessage() throws Exception
  {
    SocketService ss = new SocketService();
    ss.serve(999, new HelloServer());
    Socket s = new Socket("localhost", 999);
    InputStream is = s.getInputStream();
    InputStreamReader isr = new InputStreamReader(is);
    BufferedReader br = new BufferedReader(isr);
    String answer = br.readLine();
    s.close();
    assertEquals("Hello", answer);
  }
```

"As expected, this doesn't compile." Jerry said. "We need to write `HelloServer`."

"I think I know what to do." I said. "`HelloServer` is a class that derives from `SocketServer` and implements the `serve()` method to send the string `'hello'` over the socket." So I grabbed the keyboard and typed the following in the `TestSocketServer.java` module:

```
class HelloServer implements SocketServer {
  public void serve(Socket s) {
    try {
      OutputStream os = s.getOutputStream();
      PrintStream ps = new PrintStream(os);
      ps.println("Hello");
    } catch (IOException e) {
    }
  }
}
```

This compiled, and the tests passed, first time.

"Nicely done." said Jerry. "Now we know we can send a message through the socket."

I knew Jerry was going to start thinking about refactoring now. I wanted to get the jump on him. So I looked carefully at the code, and I remembered what he told me about duplication. "There's some duplicated code in the unit tests." I said. "In each of the three tests we create and close the `SocketService`. We should get rid of that."

"Good eye!" He replied. "Lets move that into the `Setup` and `Teardown` functions." He grabbed the keyboard and made the following changes.

```
  private SocketService ss;
  public void setUp() throws Exception {
    ss = new SocketService();
  }

  public void tearDown() throws Exception {
    ss.close();
  }
```

Then he removed all the `ss = newSocketService();` and `ss.close();` lines from the three tests.

"That's a little better." I said. "So now lets see if we can send a message the other direction."

"Exactly what I was thinking." Said Jerry. "And I've got just the way to do it."

Jerry grabbed the keyboard and began to type a new test case.

```
  public void testReceiveMessage() throws Exception {
    ss.serve(999, new EchoService());
    Socket s = new Socket("localhost", 999);
    InputStream is = s.getInputStream();
    InputStreamReader isr = new InputStreamReader(is);
    BufferedReader br = new BufferedReader(isr);
    OutputStream os = s.getOutputStream();
    PrintStream ps = new PrintStream(os);
    ps.println("MyMessage");
    String answer = br.readLine();
    s.close();
    assertEquals("MyMessage", answer);
  }
```

"Ouch! That's pretty ugly." I said.

"Yeah, it is." Replied Jerry. "Let's get it working and then we'll clean it up. We don't want a mess like that laying around for long! You see where I am going with this though, don't you?"

"Yes, I thnk so." I said. "`EchoService` will recieve a message from the socket and just send it right back. So your test just sends `'MyMessage'`, then reads it back again."

"Right. Care to take a crack at writing `EchoService`?"

"Sure." I said, and I grabbed the keyboard.

```
class EchoService implements SocketServer {
  public void serve(Socket s) {
    try {
      InputStream is = s.getInputStream();
      InputStreamReader isr = new InputStreamReader(is);
      BufferedReader br = new BufferedReader(isr);
      OutputStream os = s.getOutputStream();
      PrintStream ps = new PrintStream(os);
      String token = br.readLine();
      ps.println(token);
```

```
    } catch (IOException e) {
    }
  }
}
```

"Ick." I said. "More of the same kind of ugly code. We keep on having to create `PrintStream` and `BufferedReader` objects from the socket. We really need to clean that up."

"We'll do that just as soon as the test works." Said Jerry. And then he looked at me expectantly.

"Oh!" I said. "I forgot to run the test."

Sheepishly I pushed the test button and watched it pass.

"That wasn't hard to get working." I said. "Now lets get rid of that ugly code.".

Keeping the keyboard, I extracted several functions from the previous mess in `EchoService`.

```
class EchoService implements SocketServer {
  public void serve(Socket s) {
    try {
      BufferedReader br = getBufferedReader(s);
      PrintStream ps = getPrintStream(s);
      String token = br.readLine();
      ps.println(token);
    } catch (IOException e) {
    }
  }

  private PrintStream getPrintStream(Socket s) throws IOException {
    OutputStream os = s.getOutputStream();
    PrintStream ps = new PrintStream(os);
    return ps;
  }

  private BufferedReader getBufferedReader(Socket s) throws IOException {
    InputStream is = s.getInputStream();
    InputStreamReader isr = new InputStreamReader(is);
    BufferedReader br = new BufferedReader(isr);
    return br;
  }
}
```

"Yes." Said Jerry. "That makes the serve method of `EchoService` much better, but it clutters the class quite a bit. Also, it doesn't really help the `testRecieveMessage` function. That function has the same kind of ugliness. I wonder if `getBufferedReader` and `getPrintStream` are in the right place."

"This is going to be a recurring problem." I said. "Anybody who wants to use `SocketService` is going to have to convert the socket into a `BufferedReader` and a `PrintStream`."

"I think that's your answer!" Jerry replied. "The `getBufferedReader` and `getPrintStream` methods really belong in `SocketService`."

This made a lot of sense to me. So I moved the two functions into the `SocketService` class and changed `EchoService` accordingly.

```
public class SocketService {
 [...]
  public static PrintStream getPrintStream(Socket s) throws IOException {
    OutputStream os = s.getOutputStream();
    PrintStream ps = new PrintStream(os);
    return ps;
```

```
  }

  public static BufferedReader getBufferedReader(Socket s) throws IOException {
    InputStream is = s.getInputStream();
    InputStreamReader isr = new InputStreamReader(is);
    BufferedReader br = new BufferedReader(isr);
    return br;
  }
}
```

```
class EchoService implements SocketServer {
  public void serve(Socket s) {
    try {
      BufferedReader br = SocketService.getBufferedReader(s);
      PrintStream ps = SocketService.getPrintStream(s);
      String token = br.readLine();
      ps.println(token);
    } catch (IOException e) {
    }
  }
}
```

The tests all still ran. Keeping my momentum, I said: "Now I should be able to fix the `testReceiveMessage` method too." So, while Jerry watched, I made that change too.

```
  public void testReceiveMessage() throws Exception {
    ss.serve(999, new EchoService());
    Socket s = new Socket("localhost", 999);
    BufferedReader br = SocketService.getBufferedReader(s);
    PrintStream ps = SocketService.getPrintStream(s);
    ps.println("MyMessage");
    String answer = br.readLine();
    s.close();
    assertEquals("MyMessage", answer);
  }
```

"Yeah, that's a lot better." Jerry said.

"Not only that, but the tests still pass." I said. Then I noticed something. "Oops, there's another one in `testSendMessage`." I said. And so I made that change too.

```
  public void testSendMessage() throws Exception {
    ss.serve(999, new HelloServer());
    Socket s = new Socket("localhost", 999);
    BufferedReader br = SocketService.getBufferedReader(s);
    String answer = br.readLine();
    assertEquals("Hello", answer);
  }
```

The tests all still ran. I sat there looking through the TestSocketServer class, trying to find any other code I could eliminate.

"Are you done?" asked Jerry.

After a few more seconds of scanning, I relaxed and said: "I think so."

"Good." He said. "I was starting to see smoke coming out of your ears. I think it's time for a quick break."

"OK, but I have a question first."

"Shoot."

I looked back and forth between Jerry and the code for a few seconds, and then I said: "We haven't made a single change to `SocketService`. We've added two new tests -- `testSendMessage`, and `testReceiveMessage` -- and they both just worked. And yet we spent a lot of time writing those tests, and also refactoring. What good did it do us? We didn't change the production code at all!"

Jerry raised an eyebrow and gave me an impenetrable regard. "It's interesting that you noticed that. Do you think we wasted our time?"

"It didn't feel like a waste, because we proved to ourselves that you could send and receive messages. Still, we wrote and refactored a lot of code that didn't help us add anything to the production code."

"You don't think that `getBufferedReader` and `getPrintStream` were worth putting into production?"

"They're pretty trivial, and all they're really doing is supporting the tests."

Jerry sighed and looked away for a minute. Then he turned back to me and said: "If you were just coming onto this project, and I showed you these tests, what would they teach you?"

That was a strange question. My first reaction was to answer by saying that they'd teach me that the authors had wanted to prove their code worked. But I held that thought back. Jerry wouldn't have asked me that question if he hadn't wanted me to carefully think about the answer.

What would I learn by reading those tests? I'd learn how to create a `SocketService`, and how to bind a `SocketServer` derivative to it. I'd also learn how to send and receive messages. I'd learn the names and locations of the classes in the framework, and how to use them.

So I gathered up my courage and said: "You mean we wrote these tests as examples to show to others?"

"That's part of the reason, Alphonse. Yes. Others will be able to read these tests and see how to work the code we're writing. They'll also be able to work through our reasoning. Moreover, they'll be able to compile and execute these tests in order to prove to themselves that our reasoning was sound."

"There's more to it, than that. " He continued. "But we'll leave that for another time. Now lets take our break."

My elbow still twinged, so I was glad to see that the turbo was repaired. On the way up the lift I kept rolling same thought around in my head. "Tests are a form of documentation. Compilable, executable, and always in sync."

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

www.objectmentor.com/resources/articles/CraftsmanCode/SDSocketServiceR3_SendAndReceive.zip