The Craftsman: 9
SocketService 4

Robert C. Martin
20 Dec 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/SDSocketServiceR3_SendAndReceive.zip

---

Alphonse awoke to the gentle bleating of his PDA. Shaking the sleep from his brain he switched off the sleeping field and climbed into the shower. As the hypersonic spray scrubbed and massaged him he allowed his mind to wander to the events of the previous day.

...He had come back from break, still thinking about the documentation value of tests. Jerry was waiting for him and said:

"I'm glad you're back. I'm just finishing up the next test case. Take a look at it and see if you can guess it's purpose."

```
  public void testMultiThreaded() throws Exception {
    ss.serve(999, new EchoServer());
    Socket s1 = new Socket("localhost", 999);
    BufferedReader br = SocketService.getBufferedReader(s1);
    PrintStream ps = SocketService.getPrintStream(s1);

    Socket s2 = new Socket("localhost", 999);
    BufferedReader br2 = SocketService.getBufferedReader(s2);
    PrintStream ps2 = SocketService.getPrintStream(s2);

    ps2.println("MyMessage");
    String answer2 = br2.readLine();
    s2.close();

    ps.println("MyMessage");
    String answer = br.readLine();
    s1.close();

    assertEquals("MyMessage", answer2);
    assertEquals("MyMessage", answer);
  }
```

"It's a little complicated; but it looks like you are trying to prove that the `SocketService` can deal with two simultaneous connections."

"Right." said Jerry. "Did you notice that the first connection was the last to close?"

"No; but now that you mention it I can see that that's true. Why did you do that?"

"I wanted two sessions open simultaneously. "

"Why?"

Jerry gave me a curious look and said: "Because then the `serve` method in the `SocketService` class will have been entered twice, in two different threads, before either had had a change to exit. When a function is entered more than once before it exits, it is called *reentrant*."

"But why do you want to test it?"

"Because reentrant functions often give us the most interesting kinds of problems."

I didn't understand this; but I knew that Jerry would explain himself eventually, so I just nodded.

"OK." He said. "Let's run this test."

I compiled the test and ran it. A green bar moved quickly across the test window, telling us that all our previous tests were still working. But then it froze, just before the end. I waited for a few seconds to see if it would wake up and finish; but it never did. It just hung there.

"Uh..." I uttered intelligently.

"Yeah." said Jerry. "What do you think is wrong?"

I studied the code in `SocketService.serve` for a minute, and then I said: "Oh, that's easy. Look at this loop:"

```
while (running) {
   try {
     Socket s = serverSocket.accept();
     itsServer.serve(s);
     s.close();
   } catch (IOException e) {
   }
}
```

The first connection is hung in this part:
String token = bufferedReader.readLine();
--- because there is nothing to read

"`itsServer.serve` isn't returning to catch the second connection. The first connection is hung in the `EchoServer` waiting for you to send a message. So we never go around the loop to call `accept` for the second socket connection."

Jerry beamed at me. "Well done! Now what do we do about it?"

"We need to put itsServer.serve in its own thread so that the loop can return without waiting for it."

"Right again! Care to take a stab at it?"

So I took the keyboard and changed the `SocketService.serve` method as follows:

```
while (running) {
  try {
    Socket s = serverSocket.accept();
    new Thread(new ServiceRunnable(s)).start();
  } catch (IOException e) {
  }
}
```

Then I added the new `ServiceRunnable` class as an inner class within `SocketService`.

```
class ServiceRunnable implements Runnable {
  private Socket itsSocket;

  ServiceRunnable(Socket s) {
    itsSocket = s;
  }

  public void run() {
    try {
      itsServer.serve(itsSocket);
      itsSocket.close();
    } catch (IOException e) {
    }
```

```
    }
  }
```

"That should do it." I said. So I hit the test button and was rewarded with success.

"Hit it a few more times." Jerry said.

"Oh no, not one of these." I complained, remembering the intermittent failure we had when we first started. So I hit the test button a few more times and, sure enough, I saw a failure.

be very careful with this, you never encountered this NullPointerException in your practice project

```
1) testMultiThreaded(TestSocketServer)
java.lang.NullPointerException
    at SocketService.close(SocketService.java:32)
    at TestSocketServer.tearDown(TestSocketServer.java:30)
```

"What the deuce?" I said, and I looked at line 32 of `SocketService.java`.

```
30  public void close() throws Exception {
31    running = false;
32    serverSocket.close();
33  }
```

"Now wait a minute." I said. "How could it be getting a null pointer exception there?" I pulled up `TestSocketServer` line 30.

```
29  public void tearDown() throws Exception {
30    ss.close();
31  }
```

"Now wait just a darned minute here." I repeated. "This doesn't make any sense. Teardown is closing the `SocketService`, just like it should, and yet the `serverSocket` is `null`? But if `serverSocket` were `null`, we'd have had errors at the beginning of `testMultiThreaded`, not in `tearDown`."

Jerry must have felt like being useful, because he said: "Yeah."

"Jerry, what's going on? This doesn't make any sense."

"You said that."

"But it's true. The `serverSocket` variable *can't* be `null`!"

"Alphonse. Let's think for a minute. What state are all the threads in?"

"Huh?"

"The threads. What are they all doing when `tearDown` gets called?"

I thought about this for a minute. Clearly the test case passed; otherwise `tearDown` would not have been called. That meant that both socket connections were accepted, and that the `serverThread` had been around its loop twice. The `serverThread` was either blocked on its third `accept` call, or it had not yet returned from the `start` function that kicked off the second `ServiceRunnable` thread.

The first `ServiceRunnable` thread had entered `EchoServer`, which had read and written the message. But it might not have terminated yet. It could still be waiting to return from the `println` that sent the message back to the test case. But the second `ServiceRunnable` thread had surely had enough time to exit. It has received and sent its message long since.

I explained all this to Jerry, and he nodded.

"Yes." He said. "That's the way I figure it too."

"So why the null pointer exception?" I asked.

"I don't know." He said. "But the fact that it's breaking when we close the server socket makes me think that we're leaving some thread running that is interfering with the socket library."

"You mean you think there's a bug in the socket library?"

Jerry just stared at the screen and said: "I'm not sure. Let's try a few experiments."

The code that Jerry and Alphonse finished can be retrieved from:
www.objectmentor.com/resources/articles/CraftsmanCode/SDSocketServiceR3_SendAndReceive.zip