The Craftsman: 7
SocketService 2

Robert C. Martin
15 Oct 2002

Last time, Alphonse and Jerry started working on a simple java framework for supporting socket services. Their first test case uncovered a race condition that they were able to resolve. The current suite of unit tests is shown in Listing 1, and the production code is in Listing 2.

Listing 1.

```
import junit.framework.TestCase;
import junit.swingui.TestRunner;

import java.io.IOException;
import java.net.Socket;

public class TestSocketServer extends TestCase {
  public static void main(String[] args) {
    TestRunner.main(new String[]{"TestSocketServer"});
  }

  public TestSocketServer(String name) {
    super(name);
  }

  public void testOneConnection() throws Exception {
    SocketService ss = new SocketService();
    ss.serve(999);
    connect(999);
    ss.close();
    assertEquals(1, ss.connections());
  }

  private void connect(int port) {
    try {
      Socket s = new Socket("localhost", port);
      try {
        Thread.sleep(100);
      } catch (InterruptedException e) {
      }
      s.close();
    } catch (IOException e) {
      fail("could not connect");
    }
  }
}
```

Listing 2.

```
import java.io.IOException;
import java.net.*;

public class SocketService {
  private ServerSocket serverSocket = null;
  private int connections = 0;
  private Thread serverThread = null;

  public void serve(int port) throws Exception {
    serverSocket = new ServerSocket(port);
    serverThread = new Thread(
      new Runnable() {
        public void run() {
          try {
            Socket s = serverSocket.accept();
            s.close();
            connections++;
          } catch (IOException e) {
          }
        }
      }
    );
    serverThread.start();
  }

  public void close() throws Exception {
    serverSocket.close();
  }

  public int connections() {
    return connections;
  }
}
```

We came back from our break, ready to continue the `SocketService`.

"We've proven that we can connect once. Lets try to connect several times." said Jerry.

"Sounds good." I said. So I wrote the following test case.

```
  public void testManyConnections() throws Exception {
    SocketService ss = new SocketService();
    ss.serve(999);
    for (int i = 0; i < 10; i++)
      connect(999);
    ss.close();
    assertEquals(10, ss.connections());
  }
```

"OK, this fails." I said.

"As it should." Jerry replied. "The `SocketService` only calls `accept` once. We need to put that call into a loop."

"When should the loop terminate?" I asked.

Jerry thought for a second and said: "When we call `close` on the `SocketService`.".

"Like this?" And I made the following changes:

```
public class SocketService {
```

```
  private ServerSocket serverSocket = null;
  private int connections = 0;
  private Thread serverThread = null;
  private boolean running = false;

  public void serve(int port) throws Exception {
    serverSocket = new ServerSocket(port);
    serverThread = new Thread(
      new Runnable() {
        public void run() {
          running = true;
          while (running) {
            try {
              Socket s = serverSocket.accept();
              s.close();
              connections++;
            } catch (IOException e) {
            }
          }
        }
      }
    );
    serverThread.start();
  }

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

I ran the tests, and they both passed.

"Good." I said. "Now we can connect as many times as we like. Unfortunately the `SocketService` doesn't do very much when we connect to it. It just closes."

"Yeah, let's change that." said Jerry. "Lets have the `SocketService` send us a "Hello" message when we connect to it."

I didn't care for that. I said: "Why should we pollute `SocketService` with a "Hello" message just to satisfy our tests? It would be good to test that the `SocketService` can send a message, but we don't want the message to be part of the `SocketService` code!"

"Right!" Jerry agreed. "We want the message to be specified by, and verified by, the test."

"How do we do that?" I asked.

Jerry smiled and said: "We'll use the *Mock Object* pattern. In short, we create an interface that the `SocketService` will execute after receiving a connection. We'll have the test implement that interface to send the "Hello" message. Then we'll have the test read the message from the client socket and verify that it was sent correctly."

I didn't know what the *Mock Object* pattern was, and his description of interfaces confused me. "Can you show me?" I asked.

So Jerry took the keyboard and began to type.

"First we'll write the test."

```
  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);
  }
```

I examined this code carefully. "OK, so you're creating something called a `HelloServer` and passing it into the `serve` method. That's going to break all the other tests!"

"Good Point!" Jerry exclaimed. "That means we need to refactor those other tests before we continue."

"But the services in the other two tests don't *do* anything." I complained.

"Of course they do – they count connections! Remember how much you didn't like that connections variable, and it's accessor? Well now we're going to get rid of them."

"We are?"

"Just watch." Jerry laughed. "First we'll change the two tests, and add a `connections` variable to the test case."

```
  public void testOneConnection() throws Exception
  {
    ss.serve(999, connectionCounter);
    connect(999);
    assertEquals(1, connections);
  }

  public void testManyConnections() throws Exception
  {
    ss.serve(999, connectionCounter);
    for (int i=0; i<10; i++)
      connect(999);
    assertEquals(10, connections);
  }
```

"Next we'll make the interface."

```
import java.net.Socket;

public interface SocketServer {
  public void serve(Socket s);
}
```

"Next we'll create the `connectionCounter` variable and initialize it in the `TestSocketServer` constructor with an anonymous inner class that bumps the `connections` variable.

```
public class TestSocketServer extends TestCase {
  private int connections = 0;
  private SocketServer connectionCounter;

  public static void main(String[] args) {
    TestRunner.main(new String[]{"TestSocketServer"});
  }

  public TestSocketServer(String name) {
    super(name);
    connectionCounter = new  SocketServer() {
      public void serve(Socket s) {
```

```
        connections++;
      }
    };
  }
  ...
```

"Finally, we'll make it all compile by adding the extra argument to the `serve` method of the `SocketService`, and commenting out the new test.

```

  public void serve(int port, SocketServer server) throws Exception {
    ...
  }
```

"OK, I see what you are doing." I said. The two old tests should fail now because `SocketService` never invokes the `serve` method of its `SocketServer` argument."

Sure enough the tests failed for exactly that reason.

I knew what to do next. I grabbed the keyboard and made the following change:

```
public class SocketService {
  private ServerSocket serverSocket = null;
  private int connections = 0;
  private Thread serverThread = null;
  private boolean running = false;
  private SocketServer itsServer;

  public void serve(int port, SocketServer server) throws Exception {
    itsServer = server;
    serverSocket = new ServerSocket(port);
    serverThread = new Thread(
      new Runnable() {
        public void run() {
          running = true;
          while (running) {
            try {
              Socket s = serverSocket.accept();
              itsServer.serve(s);
              s.close();
              connections++;
            } catch (IOException e) {
            }
          }
        }
      }
    );
    serverThread.start();
  }
  ...
```

This made all the tests run.

"Great!" Said Jerry. Now we've got to make the new test work."

So I uncommented the test and compiled it. It complained about `HelloServer`.

"Oh, right. We need to implement the HelloServer. It's just going to spit the word "hello" out the socket, right?"

"Right." Jerry confirmed.

So I wrote the following new class in the `TestSocketServer.java` file.

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

The tests all passed.

"That was pretty easy." Said Jerry.

"Yeah. That *Mock Object* pattern is pretty useful. It let us keep all the test code in the test module. The `SocketService` doesn't know anything about it."

"It's even more useful than that." Jerry replied. "Real servers will also implement the `SocketServer` interface."

"I can see that." I said. "Interesting that the needs of a unit test drove us to create a design that would be generally useful.

"That happens all the time." Said Jerry. "Tests are users too. The needs of the tests are often the same as the needs of the real users."

"But why is it called *Mock Object*?"

"Think of it this way. The HelloServer is a substitute for, or a mock-up of, a real server. The pattern allows us to substitute test mock-ups into real application code."

"I see." I said. "Well, we should clean this code up now and get rid of that useless connections variable."

"Agreed."

So we did a little cleanup and took another break. The resulting `SocketService` is shown below.

```
import java.io.IOException;
import java.net.*;

public class SocketService {
  private ServerSocket serverSocket = null;
  private Thread serverThread = null;
  private boolean running = false;
  private SocketServer itsServer;

  public void serve(int port, SocketServer server) throws Exception {
    itsServer = server;
    serverSocket = new ServerSocket(port);
    serverThread = makeServerThread();
    serverThread.start();
  }

  private Thread makeServerThread() {
    return new Thread(
      new Runnable() {
        public void run() {
          running = true;
          while (running) {
            acceptAndServeConnection();
          }
        }
```

```
      }
    );
  }

  private void acceptAndServeConnection() {
    try {
      Socket s = serverSocket.accept();
      itsServer.serve(s);
      s.close();
    } catch (IOException e) {
    }
  }

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