The Craftsman: 6
SocketService 1

Robert C. Martin
16 Sept 2002

Yesterday left me drained. Jerry and I had solved the prime factors problem by sneaking up on it one tiny test case at a time. It was the strangest way to solve a problem that I have ever seen; but it worked better than my original solution.

I wandered aimlessly around the corridors thinking about it again and again. I don't remember dinner, or even being in the galley. I fell asleep early and dreamed of sequences of tiny test cases.

When I reported to Jerry this morning he said:

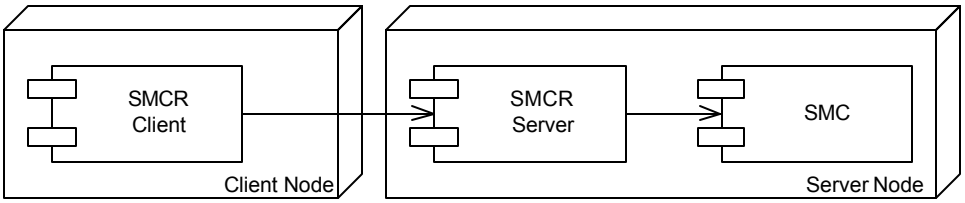
"Good Morning Alphonse. Are you ready to start working on a *real* project?"

"You bet I am!" Farb, yes, I was ready! I was tired of these play examples.

"Good! We've got a program called SMC that compiles a finite state machine grammar into Java. Mr. C. wants us to turn that program into a service delivered over the net."

"What do you mean?", I asked.

Jerry turned to a sketch-wall and began to talk and draw at the same time.

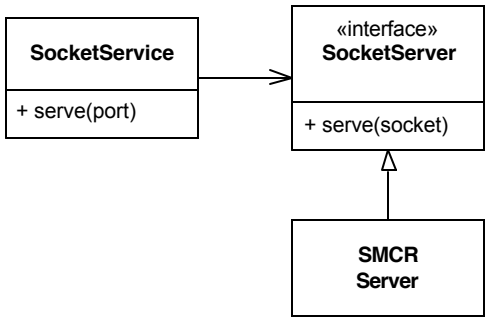


"We're going to write two programs. One called SMCR Client, and the other called SMCR Server. The user who wants to compile a finite state machine will invoke SMCR Client with the name of the file to be compiled. SMCR Client will send that file to a special computer where SMCR Server is running. SMCR Server will run the SMC compiler and then send the resulting compiled files back to SMCR Client. SMCR Client will then write them in the user's directory. As far as the user is concerned, it'll be no different than using SMC directly."

"OK, I think I understand." I said. "It sounds pretty simple."

"It is *pretty* simple." Jerry said. "But working with sockets is always just a little interesting."

So we sat down at a workstation and, as usual, got ready to write our first unit test. Jerry thought for a minute and then went back to the sketch-wall and drew the following diagram.



"Here's what I have in mind for SMCR Server." He said. "We'll put the socket management code in a class called `SocketService`. This class will catch, and manage, connections coming from the outside. When `serve(port)` is called, it'll create the service socket with the given port number and start accepting connections. Whenever a connection comes in it will create a new thread and pass control to the `serve(socket)` method of the `SocketServer` interface. That way we separate socket management code from the code that performs the services we desire."

Not knowing whether this was a good thing or not, I simply nodded. Clearly he had some reason for thinking the way he did. I just went along with it.

Next he wrote the following test.

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

"What I'm doing here is called *Intentional Programming*." Jerry said. I'm calling code that doesn't yet exist. As I do so, I express my intent about what that code should look like, and how it should behave."

"OK." I responded. "You create the `SocketService`. Then you ask it to accept connections on port 999. Next it looks like you are connecting to the service you just created on port 999. Finally you close the `SocketService` and assert that it got one connection."

"Right." Jerry confirmed.

"But how do you know `SocketService` will need the `connections` method?"

"Oh, it probably doesn't. I'm just putting it there so I can test it."

"Isn't that wasteful?" I queried?

Jerry looked me sternly in the eye and replied: "Nothing that makes a test easy is wasted, Alphonse. We often add methods to classes simply to make the classes easier to test."

I didn't like the `connections()` method, but I held my tongue for the moment.

We wrote just enough of the `SocketService` constructor, and the `serve`, `close`, and `connect` methods to get everything to compile. They were just empty functions. When we ran the test, it failed as expected.

Next Jerry wrote the `connect` method as part of the test case class.

```
private void connect(int port) {
  try {
    Socket s = new Socket("localhost", port);
    s.close();
  } catch (IOException e) {
```

```
      fail("could not connect");
    }
  }
```

Running this produced the following error:

```
testOneConnection: could not connect
```

I said: "It's failing because it can't find port 999 anywhere, right?"

"Right!" said Jerry. "But that's easy to fix. Here, why don't you fix it?"

I'd never written socket code before, so I wasn't sure what to do next. Jerry pointed me to the `ServerSocket` entry in the Javadocs. The examples there made it look pretty simple. So I fleshed in the `SocketService` methods as shown below.

```
import java.net.ServerSocket;

public class SocketService {
  private ServerSocket serverSocket = null;

  public void serve(int port) throws Exception {
    serverSocket = new ServerSocket(port);
  }

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

  public int connections() {
    return 0;
  }
}
```

Running this gave us: `testOneConnection: expected:<1> but was:<0>`

"Aha!" I said. "It found port 999. Cool! Now we just need to count the connection!"

So I changed the `SocketService` class as follows:

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

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

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

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

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

But this didn't work. It didn't even fail. When I ran the test, the test hung.

"What's going on?" I asked.

Jerry smiled. "See if you can figure it out, Alphonse. Trace it through."

"OK, let's see. The test program calls `serve`, which creates the socket and calls `accept`. Oh! `accept` doesn't return until it gets a connection! And so `serve` never returns, and we never get to call `connect`."

Jerry nodded. "So how are you going to fix this Alphonse?"

I thought about this for a minute. I needed to call the `connect` function after calling `accept`, but when you call `accept`, it won't return until you call connect. At first this sounded impossible.

"It's not impossible, Alphonse." said Jerry. "You just have to create a thread."

I thought about this a little more. Yes, I could put the call to accept in a separate thread. Then I could invoke that thread and then call connect.

"I see what you mean about socket code being a little interesting." I said. And then I made the following changes.

```
  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();
  }
```

"Nice use of the anonymous inner class, Alphonse." said Jerry.

"Thanks." It felt good to get a compliment from him. "But it does make for a lot of monkey tails at the end of the function."

"We'll refactor that later. First, lets run the test."

The test ran just fine, but Jerry looked pensive, like he'd just been lied to.

"Run the test again, Alphonse."

I happily hit the run button, and it worked again.

"Again." he said.

I looked at him for a second to see if he was joking. Clearly he was not. His eyes were locked on the screen, as though he were hunting a dribin. So I hit the button again and saw:

```
testOneConnection: expected:<1> but was:<0>
```

"Now wait a minute!" I hollered. "That can't be!"

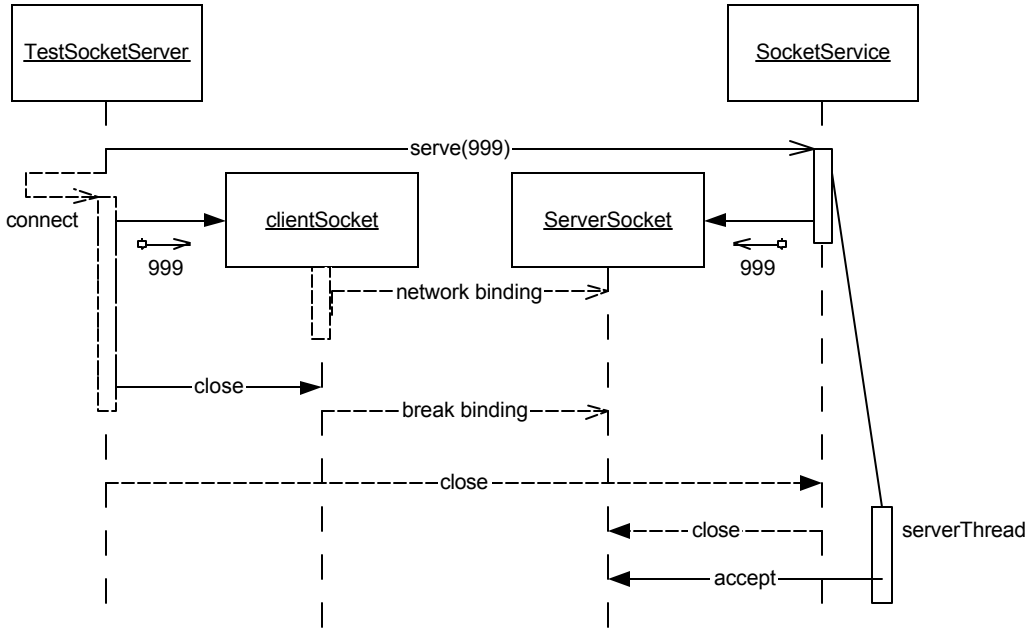
"Oh, yes it can." Jerry said. "I was expecting it."

I repeatedly pushed the button. In ten attempts I saw three failures. Was I losing my mind? How can a program behave like that?

"How did you know, Jerry? Are you related to the oracle of Aldebran?"

"No, I've just written this kind of code before, and I know a little bit about what to expect. Can you work out what's happening? Think it through carefully."

My brain was already hurting, but I started piecing things together. I went to the sketch-wall and began to draw.



When I had worked it out, I recited the scenario to Jerry. "`TestSocketServer` sent the `serve(999)` message to `SocketService`. `SocketService` created the `ServerSocket` and the `serverThread` and then returned. `TestSocketServer` then called `connect` which created the client socket. The two sockets must have found each other because we didn't get a '`could not connect`' error. The `ServerSocket` must have accepted the connection, but perhaps `serverThread` hadn't had a chance to run yet. And while `serverThread` was blocked, the `connect` function closed the client socket. Then `TestSocketServer` sent the close message to the `SocketService`, which closed the `serverSocket`. By the time the `serverThread` got a chance to call the `accept` function, the server socket was closed."

"I think you're right." Jerry said. "The two events – accept and close – are asynchronous, and the system is sensitive to their order. We call that a *race condition*. We have to make sure we always win the race.

We decided to test my hypothesis by putting a print statement in the catch block after the `accept` call. Sure enough, three times in ten, we saw the message.

"So how can we get our unit test to run?" Jerry asked me.

"It seems to me that the test can't just open the client socket and then immediately close it." I replied. "It needs to wait for the accept."

"We could just wait for 100ms real time before closing the client socket." he said.

"Yeah, that would probably work." I replied. "But it's a bit ugly."

"Let's see if we can get it to work, and then we'll consider refactoring it."

So I made the following changes to the `connect` method.

```
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");
    }
  }
```

This made the tests pass 10 for 10.

"This is scary." I said. "The fact that a test passes once doesn't mean the code is working."

"Yeah." Jerry agreed. "When you are dealing with multiple threads, you have to keep an eye out for possible race conditions. Hitting the test button multiple times is a good habit to get into."

"It's a good thing we found this in our test cases." I said. "Finding it after the system was running would have been a *lot* harder."

Jerry just nodded.

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 Objec*t?"

"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();
    }
}
```

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

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) {
           }
         }
```

"`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.

```
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

The Craftsman: 10 SocketService 5 Dangling Threads

Robert C. Martin
14 Jan, 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/SDSocketServiceR4_TestMultiThreaded.zip

---

I have a standing monthly reservation for breakfast on the observation deck. It's a bit of an extravagance on an apprentice's credit, but I love eating under the starbow.

As I ate, I started thinking about the dangling thread problem we solved yesterday. We fixed the `serverThread`, but the serviceRunnable threads were all still left dangling. I was certain Jerry would want to fix those before moving on to other issues.

Sure enough, when I walked into the lab, Jerry was already there with the following test case on the screen:

```
public void testAllServersClosed() throws Exception {
  ss.serve(999, new WaitThenClose());
  Socket s1 = new Socket("localhost", 999);
  Thread.sleep(20);
  assertEquals(1,WaitThenClose.threadsActive);
  ss.close();
  assertEquals(0, WaitThenClose.threadsActive);
}
```

"Ah, I see what you are doing." I said. "You are making sure that all `SocketServers` are closed by the time you return from closing the `SocketService`."

"Right! I want to make sure we don't leave those servers dangling."

"But you are only testing it with one server. Don't we need to test it with many?"

"Right again! Lets get this simple test working first and then we'll add more servers to it."

"OK", I said, "I think I know how to write `WaitThenClose`."

```
class WaitThenClose implements SocketServer {
  public static int threadsActive = 0;
  public void serve(Socket s) {
    threadsActive++;
    delay();
    threadsActive--;
  }

  private void delay() {
```

```
    try {
      Thread.sleep(100);
    } catch (InterruptedException e) {
    }
  }
}
```

As I wrote this code, Jerry nodded in concurrence as it appeared on the screen. This was just what he had thought `WaitThenClose` would be like. Once it was done and compiled we ran the tests, and they failed as expected:

```
1) testAllServersClosed AssertionFailedError: expected:<0> but was:<1>
```

Jerry rubbed his hands together and said: "OK, now lets make it pass the test." He made a move to grab the keyboard but I blocked it saying: "I think I have an idea about that." So while Jerry watched, I made the following changes.

```
  private LinkedList serverThreads = new LinkedList();

  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();
              Thread serverThread = new Thread(new ServiceRunnable(s));
              serverThreads.add(serverThread);
              serverThread.start();
            } catch (IOException e) {
            }
          }
        }
      }
    );
    serverThread.start();
  }

  public void close() throws Exception {
    if (running) {
      running = false;
      serverSocket.close();
      serverThread.join();
      for (Iterator i = serverThreads.iterator(); i.hasNext();) {
        Thread thread = (Thread) i.next();
        serverThreads.remove(thread);
        thread.join();
      }
    } else {
      serverSocket.close();
    }
  }
```

As I compiled this code, Jerry got a funny look on his face.

"Is this OK?" I asked.

"Let's see." He said. "Run the test."

When I ran test it gave me a peculiar error:

```
1) testOneConnection java.util.ConcurrentModificationException
```

"What's that?" I said.

"You've broken a rule, Alphonse. You never add or remove something from a list while you have an iterator active."

"Ouch! Of course! I knew that!" I said sheepishly. "OK, but that's easy to fix because I really don't need to remove the thread from the list, do I?" I quickly removed the `remove` line, and ran the test again.

"Ah! Now it's working just fine."

Jerry nodded, but stared at me expectantly.

"What!?" I exclaimed after enduring his gaze for half a minute.

"You are still modifying the list while the iterator is active."

"I am?" I was quite puzzled by this. "Jerry, there's only one place where the list is modified, and that's where the thread is added in the `running` loop. How can that be called while the iterator is active?"

"Oh, I think it's possible. The call to accept might be on the verge of returning as you enter the iterator loop. When the iterator loop blocks on a join, the accept will return and add a thread to the list."

"OK, I can sort of see that, but can we test for it?"

"We could, but there's no point. It turns out that there's another place where you'll be modifying the list while the iterator is open."

"There is?"

"Yes, you are just about to add it. How many threads are in that list?"

"All of them...Oh! I should be deleting the thread from the list when the thread completes! Otherwise completed threads will just hang around in the list." I grabbed the keyboard and made the following change.

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

  ServiceRunnable(Socket s) {
    itsSocket = s;
  }

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

"Aha! Now it fails again." I said. "You were right! Some of the threads complete before the iterator loop finishes. Boy, it's a good thing the iterator complains about concurrent updates!"

"Yes, it is." Nodded Jerry. "Now let me show you how I'd fix this."

Once again he took the keyboard, and once again I did not protest. He proceeded to make the following change.

```
public void close() throws Exception {
  if (running) {
    running = false;
    serverSocket.close();
    serverThread.join();
    while (serverThreads.size() > 0) {
      Thread t = (Thread)serverThreads.get(0);
      serverThreads.remove(t);
```

```
        t.join();
      }
    } else {
      serverSocket.close();
    }
  }
```

"There!" Jerry said. "Now all the tests pass again."

"I get it!" I said. "Instead of using an iterator, you just pull the first element off the list and keep looping until the list is empty."

"Right. That way there's no iterator open for a long period of time. Joins can take awhile. It's not good to leave iterator open for long periods when other threads might try to modify the list."

"So are we done?"

Jerry shook his head. "No, there's still a danger."

"What do you mean? What's wrong now?"

"Alponse, whenever you have a container being modified by many different threads, there is a chance that the two threads will collide inside the container. One thread might be in the middle of adding an element while another thread is in the middle of deleting one. When that happens the container can get corrupted and bizarre things can start happening."

"Oh, so you mean we should synchronize access to the container?"

"Yes, that's exactly what I mean. We need to make sure that while the container is being modified, no other thread can access it."

"OK, I can do that. It's pretty simple!" So I proceeded to surround the `add` statement and the two `remove` statements with `synchronized(serverThreads){...}`. I ran the tests, and they all still worked.

"That's one way." Jerry said with a smile. "But it's a bit error prone. If someone else modifies the code and puts in another add or remove then they have to remember to put the `synchronized` statements in place. If they forget then bad things could happen."

I thought about that for a minute and decided that he was right. It would be better if we didn't have to surround every statement that manipulated the list with synchronization. "You know a better way then?"

"Yes." He took the keyboard, took out all my `synchronized` statements. Then he changed one more line of code -- the line that created the `LinkedList` in the first place:

```
private List serverThreads = Collections.synchronizedList(new LinkedList());
```

Jerry compiled the code and ran all the tests. Everything worked just fine. Then he looked at me and asked: "What do you know about design patterns, Alphonse? Have you heard of the Decorator pattern?"

"I've heard of them, of course, and I've seen the book on people's bookshelves, but I don't know too much about them."

Jerry looked sternly at me and said: "Then it's time you started learning about them in earnest. You can borrow my book and study it if you like. The first think I want you to read is the chapter on the Decorator pattern. The `synchronizedList` function that we just called wraps the `LinkedList` in a Decorator. All calls to the LinkedList are synchronized by it."

"That sounds like a pretty good solution." I said.

"Yeah, well, you just have to remember to explicitly synchronize any use of an iterator." He grimaced.

"Really? You mean iteration isn't synchronized in a synchronized list?"

"TANSTAAFL!" He replied?

"Huh?"

"TANSTAAFL! There Ain't No Such Thing As A Free Lunch."

"I know." I said as I headed off toward the galley for lunch.

The code that Jerry and Alphonse finished can be retrieved from:
www.objectmentor.com/resources/articles/CraftsmanCode/SDSocketServiceR4_TestMultiThreaded.zip