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