

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 a 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.](http://www.objectmentor.com/resources/articles/CraftsmanCode/Craftsman_11_SMCRemote_1_WhatsMain.zip)

zip

The Craftsman: 12 SMCRemote Part II Three Ugly Lines

Robert C. Martin
18 March, 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/Craftsman_11_SMCRemote_1_WhatsMain.zip

I spent my break up in the observation deck. We had a little bit of excitement as the ice shield passed through a thick patch of particles that made the ice flicker with blue flashes and transient patterns all over it's surface.

As always, Jerry was waiting for me after break. I wondered if he ever actually took any of this breaks. He beamed at me and said:

"OK, let's ship a file over the socket."

"Did you watch the Cherenkov display?" I asked.

"A beauty!" he said. I guess he *does* take his breaks. But where does he go?

"OK, lets ship a file." I said. "I'll write the test case." I grabbed the keyboard and started to type. The first thing I wrote was the code that created the file to be sent through the socket.

```
public void testSendFile() throws Exception {  
    File f = new File("testSendFile");  
    FileOutputStream stream = new FileOutputStream(f);  
    stream.write("I am sending this file.".getBytes());  
    stream.close();  
}
```

"I know you like to create your data files in the test code rather than depending on them to just be there."

"Right." said Jerry. "But do you notice that you have quite a bit of duplication?"

I looked around in the test class and immediately saw that we had written almost this exact same code in the `testCountBytesInFile()` method that we wrote before our break.

"It's just a four lines of code." I said. "Hardly major duplication."

"True." Jerry replied. "But duplication should always be removed as soon as possible. If you allow duplicate code to accumulate in your project you'll find yourself with a huge source of confusion and bugs."

"OK, it's easy enough to fix." So I extracted a new function called `createTestFile()` and changed both `testCountBytesInFile()`, and `testSendFile()` to call it.

```
private File createTestFile(String name, String content)
```

```

throws IOException {
    File f = new File(name);
    FileOutputStream stream = new FileOutputStream(f);
    stream.write(content.getBytes());
    stream.close();
    return f;
}

```

I ran the tests just to make sure I hadn't broken anything, and then continued writing the test. I knew that the test had to simulate `main()`, so I called all the functions that `main()` would have to call. Then I added one final call that would send the file.

```

public void testSendFile() throws Exception {
    File f = createTestFile("testSendFile", "I am sending this file.");
    c.setFilename("testSendFile");
    assertTrue(c.connect());
    assertTrue(c.prepareFile());
    assertTrue(c.sendFile());
}

```

"Good." Jerry said. You are anticipating that we'll need a method in the client named `sendFile()`."

"Right." I said. "This method will send the prepared file."

I turned back to the test and then got stuck. I didn't know how to proceed. How was I going to test that this file that I had created and "sent" actually got sent to the server. We didn't even *have* a server? Did I have to write the whole server before I could test this? What was I testing anyway?

I sat there for about a minute while Jerry watched expectantly. Then I turned to him and explained my dilemma.

"No, you don't need to write the server." Jerry said. "At this point all we are testing is the client's ability to *send* the file, not the servers ability to receive it."

"How can I send the file without having a server to receive it?"

"You can create a stub server that does as little work as possible. It doesn't have to receive the file at first, it just has to acknowledge that you sent the file correctly."

"So...hmmm...something like this?"

```

    assertTrue(server.fileReceived);

```

"Yah, that ought to do." Jerry nodded. "Now why don't you make that test pass?"

I thought about this for a minute and then realized that this shouldn't be very hard. So I wrote a simple mock server that did nothing.

```

class TestSMCRServer implements SocketServer {
    public boolean fileReceived = false;
    public void serve(Socket socket) {
    }
}

```

Jerry said: "Ah, more duplication!"

I looked around and saw that before the break we had implemented a very similar mock server in the `testConnectToSMCRemoteServer()` method, so I eliminated it.

```

public void testConnectToSMCRemoteServer() throws Exception {
    boolean connection = c.connect();
    assertTrue(connection);
}

```

Then I started the server in the `setUp()` method of the test, and closed it in `TearDown()`. Thus, before every test method was called, the server would be started; and then it would be closed when the test method returned.

```
protected void setUp() throws Exception {
    c = new SMCRemoteClient();
    server = new TestSMCRServer();
    smc = new SocketService(SMCPORT, server);
}

protected void tearDown() throws Exception {
    smc.close();
}
```

Finally, I wrote a dummy `sendFile()` method in `SMCRemoteClient`.

```
public boolean sendFile() {
    return false;
}
```

The tests failed as expected. I sighed and looked over at Jerry. "Now what?" I said.

"Send the file." he said.

"You mean just open the file and shove it over the socket?"

Jerry thought for a minute and said: "No, we probably need to tell the server to expect a file. So lets send a simple message and follow it with the contents of the file."

Jerry grabbed the keyboard and made the following changes to `SMCRemoteClient`. "First we have to get the streams out of the socket." He said.

```
public boolean connect() {
    boolean connectionStatus = false;
    try {
        Socket s = new Socket("localhost", 9000);
        is = new BufferedReader(new InputStreamReader(s.getInputStream()));
        os = new PrintWriter(new OutputStreamWriter(s.getOutputStream()));
        connectionStatus = true;
    } catch (IOException e) {
        e.printStackTrace();
        connectionStatus = false;
    }
    return connectionStatus;
}
```

"Then we have to make the file available for reading."

```
public boolean prepareFile() {
    boolean filePrepared = false;
    File f = new File(itsFilename);
    if (f.exists()) {
        try {
            itsFileLength = f.length();
            fileReader = new BufferedReader(
                new InputStreamReader(new FileInputStream(f)));
            filePrepared = true;
        } catch (FileNotFoundException e) {
            filePrepared = false;
            e.printStackTrace();
        }
    }
}
```

```

    }
    return filePrepared;
}

```

"Finally, we can send the file."

```

public boolean sendFile() {
    boolean fileSent = false;
    try {
        writeSendFileCommand();
        fileSent = true;
    } catch (Exception e) {
        fileSent = false;
    }
    return fileSent;
}

private void writeSendFileCommand() throws IOException {
    os.println("Sending");
    os.println(itsFilename);
    os.println(itsFileLength);
    char buffer[] = new char[(int) itsFileLength];
    fileReader.read(buffer);
    os.write(buffer);
    os.flush();
}

```

"Wow!" I said. "That was a lot of typing without testing."

Jerry looked at me sheepishly. "Yes, I'm a little nervous about that."

He hit the test button, and the test failed because `server.fileReceived` returned false.

"Whew!" said Jerry. "We dodged a muon pulse."

"So." I said. "You are going to precede the file with three lines. The first contains the string "Sending". The second contains the file name, and the third contains the length of the file. After that you send the file in character array."

"Right. I told you before the break that we'd need that file length."

"Hmmm. I guess so." I said.

"So now, all we have to do is receive the file in the mock server. Would you like to take a try?"

I was pretty sure I knew what to do. So I made the following changes. First I changed the test to make sure we got the filename, length, and the file contents.

```

public void testSendFile() throws Exception {
    File f = createTestFile("testSendFile", "I am sending this file.");
    c.setFilename("testSendFile");
    assertTrue(c.connect());
    assertTrue(c.prepareFile());
    assertTrue(c.sendFile());
    Thread.sleep(50);
    assertTrue(server.fileReceived);
    assertEquals("testSendFile", server.filename);
    assertEquals(23, server.fileLength);
    assertEquals("I am sending this file.", new String(server.content));
    f.delete();
}

```

Next I changed the mock server to parse the incoming data and make sure it was correct.

```

class TestSMCRServer implements SocketServer {
    public String filename = "noFileName";
    public long fileLength = -1;
    public boolean fileReceived = false;
    private PrintStream os;
    private BufferedReader is;
    public char[] content;
    public String command;

    public void serve(Socket socket) {
        try {
            os = new PrintStream(socket.getOutputStream());
            is = new BufferedReader(new InputStreamReader(socket.getInputStream()));
            os.println("SMCR Test Server");
            os.flush();
            parse(is.readLine());
        } catch (Exception e) {
        }
    }

    private void parse(String cmd) throws Exception {
        if (cmd != null) {
            if (cmd.equals("Sending")) {
                filename = is.readLine();
                fileLength = Long.parseLong(is.readLine());
                content = new char[(int)fileLength];
                is.read(content, 0, (int)fileLength);
                fileReceived = true;
            }
        }
    }
}

```

Finally, I modified `SMCRremoteClient.connect()` to wait for the SMCR message sent by the mock server.

```

public boolean connect() {
    ...
    String headerLine = is.readLine();
    connectionStatus = headerLine != null &&
        headerLine.startsWith("SMCR");
    ...
}

```

I didn't type this all at once. I didn't want to have to dodge another muon pulse. So I made the changes in much smaller steps, running the tests in between each step. I could tell that Jerry was impressed. He was still embarrassed about making such a big change. Eventually, when all the tests passed, I felt just a little superior, so I risked an observation.

"Jerry." I said. "This code is pretty ugly."

"What do you mean?"

"Well, sending those three lines, the filename, the length, and the string "Sending". That's ugly."

Jerry looked at me skeptically. He sat up as straight as he could and gave me a condescending look. "I suppose you know a better way."

"I think I do." I said, and I started to type...

To be continued.

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

www.objectmentor.com/resources/articles/CraftsmanCode/Craftsman_12_SMCRemote_11_ThreeUglyLines.zip

The Craftsman: 13

SMCRemote Part III

Objects

Robert C. Martin
21 April, 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/Craftsman_12_SMCRemote_II_ThreeUglyLines.zip

At our current τ of .045 *Destination* was still many lifetimes in the future. Each generation since *Departure* felt those lifetimes stretch inexhaustibly before them. Sometimes the feeling was depressing -- but not today. Today I used a bit of that inexhaustible time to taunt Jerry.

I stretched my hands out in front of the keyboard and cracked my knuckles. I rocked my head back and forth, pretending to work out the kinks. I paused, gazing off into the distance, putting on an air of deep contemplation. "Oh Yes!" I said, "I think I know a better way!" Jerry rolled his eyes and sighed, waiting for me to get on with it. I decided I had better not push my luck, so I started to work.

In order to write a file across the socket, Jerry had sent three lines of text first. One had the string "Sending", the next had the name of the file, and the last was length of the file. Then Jerry sent the file itself as an array of chars. The code looked like this:

```
private void writeSendFileCommand() throws IOException {
    os.println("Sending");
    os.println(itsFilename);
    os.println(itsFileLength);
    char buffer[] = new char[(int) itsFileLength];
    fileReader.read(buffer);
    os.write(buffer);
    os.flush();
}
```

When reading the file back in from the socket, he called `readLine` three times, once for each of the three lines he sent. He used the "Sending" string as a transaction identifier. He saved the second as the file name. The third was the length of the file. He used it to allocate an array of chars to use as a buffer. Then he used the length to read the appropriate number of chars from the socket.

```
private void parse(String cmd) throws Exception {
    if (cmd != null) {
        if (cmd.equals("Sending")) {
            filename = is.readLine();
            fileLength = Long.parseLong(is.readLine());
            content = new char[(int)fileLength];
        }
    }
}
```

```

        is.read(content, 0, (int) fileLength);
        fileReceived = true;
    }
}

```

This all worked just fine, but I thought it was ugly. I was convinced there was a better way. So I started to make some simple changes. First I changed the test to read objects instead of lines:

```

public void serve(Socket socket) {
    try {
        os = new PrintStream(socket.getOutputStream());
        is = new ObjectInputStream(socket.getInputStream());
        os.println("SMCR Test Server");
        os.flush();
        parse((String) is.readObject());
    } catch (Exception e) {
    }
}

private void parse(String cmd) throws Exception {
    if (cmd != null) {
        if (cmd.equals("Sending")) {
            filename = (String) is.readObject();
            fileLength = is.readLong();
            content = (char[]) is.readObject();
            fileReceived = true;
        }
    }
}

```

Next I changed the SMCRRemoteClient to write objects instead of strings.

```

public boolean connect() {
    ...
    os = new ObjectOutputStream(smcrSocket.getOutputStream());
    ...
}

private void writeSendFileCommand() throws IOException {
    os.writeObject("Sending");
    os.writeObject(itsFilename);
    os.writeLong(itsFileLength);
    char buffer[] = new char[(int) itsFileLength];
    fileReader.read(buffer);
    os.writeObject(buffer);
    os.flush();
}

```

I ran all the tests, and they worked just fine. "See?" I said. "I figured it would be better to write objects than it would be to write strings."

I looked at Jerry, but something had changed. Jerry's eyes weren't focusing. He got up and started to pace. Occasionally he would stop, look at the screen, look at me, shake his head and start pacing again. He kept mumbling something about years, experience, and stupidity. It was a little scary.

Finally he stopped, looked me square in the eye, and said: "Well, Alphonse, you've gone and done it."

"What did I *do*, Jerry?"

He stared at me for another couple of seconds. Then turned towards the turbo and said: "Follow me."

The ride down the turbo was silent. Jerry's mood was hard to read. He wasn't exactly angry, but he was certainly annoyed, and I was somehow involved with his annoyance. As we rode, silently shifting our

weight to adjust for the coriolis deflection, I tried to figure out why my simple code change would have such a profound effect upon him.

I followed Jerry into a lounge on one of the low-g levels. Apprentices weren't normally allowed below .49g. I hadn't been reading the wall markers on the way down, but this one felt to be less than .4g. Inside the lounge were five other journeymen programmers. Jerry introduced me to the group. I made sure to remember everyone's name: Johnson, Jasmine, Jason, Jasper, and Jennifer. Jerry told me to stand in the center of the lounge, while he and everyone else sat on couches around me. Then Jerry turned to the group and with a grimace he said:

"Well, it's happened. I believe Alphonse is the first apprentice this year to *Micah* his Journeyman." I felt my heart skip a beat, and my eyes got wide. It was such a simple thing! I hadn't expected *this*!

"Have any of you been Micahed so far this year?" asked Jerry. There was a murmur around the room, but everyone shook their heads. Apparently nobody had.

Jasmine looked at me long and hard. She locked her eyes on mine and said to Jerry: "OK, Jer (She pronounced it Jair). Tell us your story."

Jerry sighed and shrugged. He made a visible effort to gather himself together, and then began to speak.

"As you know, Mr. C. asked me to get the SMCRemote stuff working." They all grunted and nodded. Apparently they all knew about it. "Alphonse and I had spent a day on the SocketServer exercise; and he had done very well."

Another shock: SocketServer had been an *exercise*???

"Once we got it working we started putting together the client portion of SMCRemote. One of our first test cases was to ship a file from the client to the server over the socket." There were more nods and grunts around the room.

Jerry was getting visibly more nervous. He squirmed in his seat and avoided eye contact. "I set up a file-send transaction by shipping three text lines followed by an array of chars. The first line was the transaction id, the second was the file name, and the third was the file length." More nods. This wasn't surprising to any of them. "Clearly this was just a simple way to get the tests to pass so we could refactor into a better form." More nods, more agreeing mumbles.

"And then..." Jerry paused. "And then, Alphonse said he thought he had a -- er -- a better idea."

There was silence in the room. Jasmine's eyes were still locked on mine, but her look shifted from appraisal to speculation. One by one I could sense the gaze of the other journeymen land on me. *What was the big deal?* Why were they claiming a Micah for me?

It was Johnson who broke the spell.

"You don't me to tell us..."

Out of the corner of my eye I could see that Jerry was nodding. Nodding about *what*?

I couldn't stand it any more. I broke away from Jasmine's gaze, looked each of the other journeymen briefly in the eye, and then said: "All I did was suggest that we ship objects instead of strings! I don't see that as a Micah!"

Jennifer stepped up to me and said: "Yes, that's *all* you did. And, no, I don't suppose *you* think all that much of it. But to us, it's a big deal."

"But why?"

"Because", said Jeremy, "the most important trait of a good programmer is the ability to think abstractly. Very few programmers can actually do that. You have just proven that you can."

I was incredulous. "It was just an object." I said.

"Exactly." said Jennifer. They all nodded earnestly.

I shook my head. "Well then if this is such a good thing -- a Micah, and all -- why is Jerry so annoyed?"

"Oh that!" laughed Jasmine. "Jerry came down here at his last break and told us all about the file length issue you had with him. He was sure you were going to be impressed with him when you saw how that file length just fell into position in the file-send transaction. He was anticipating how awed you would

be."

"Yeah", said Jasper, "and you went and showed him how the length was irrelevant."

"I did?"

Jerry stood up and said: "Think about it Alphonse, if you are sending the character array as an object, why do you need to send the length separately? These guys are going to be ribbing me about this for the next six months."

"We sure are!" said Jennifer enthusiastically. "Every time we review any of his code we'll be asking him where the file length parameter is!" She giggled as Jerry grimaced and hung his head.

"You see, Alphonse", said Jasmine, "not only did you make a worthwhile leap of abstraction, your solution was *simpler* than Jerry's. What's more, it was simple in a way that invalidated Jerry's anticipated need for the file length. You *Micahed* him!"

I was beginning to understand. At least I wasn't in any kind of trouble...

"I think", said Jasmine, "that an event like this calls for a change of partners. Jerry, I'll swap apprentices with you. You take Andy, and I'll work with Alphonse for a few days."

...or was I?

To be continued.

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

www.objectmentor.com/resources/articles/CraftsmanCode/Craftsman_13_SMCRemote_III_Objects.zi

p

The Craftsman: 14 SMCRemote Part IV Transactions

Robert C. Martin
19 May, 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/Craftsman_14_SMCRemote_IV_Transactions.zip

"OK Hotshot, let's see what you've got."

The intensity of Jasmine's stare nailed me to my chair. "W-What do you mean?" I stammered.

"Look, Hotshot, you aren't going to embarrass *me* the way you did Jerry. I mean Jerry's good and all, but I'm a *lot* better."

"I wasn't trying to embarrass anyone, I..."

"Yah, sure. Let's just get on with this, shall we? What's the next change you expect to make?"

We were sitting in the lab, looking at the code that Jerry and I had just written. I had showed Jasmine how I had changed Jerry's code that sent strings across the socket to send objects instead.

"I -- er -- I don't know. I just thought sending objects would be --uh -- better than strings." God, she made me nervous. She is stunningly beautiful, in an austere kind of way. Intensity spills out of her every look and action. And those deep black eyes!

Jasmine rolled those eyes now. "THINK! Hotshot, THINK! You aren't just going to wrap a couple of strings and integers into an object are you? What does that wrapping imply? What can you *do* with it?"

"I, uh..." If she'd stop pinning me with those eyes I might be able to THINK. I simultaneously wanted to be anywhere else, and nowhere else. It felt like being torn in two.

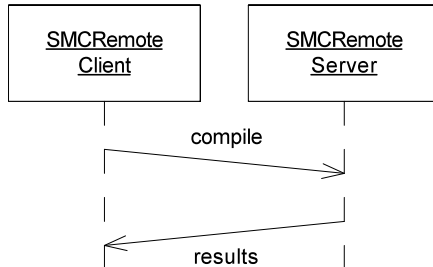
I closed my eyes and mentally recited a quick calming mantra. I forced the tension out of my mind and body. Within a few seconds I was able to consider the question she had provoked me with.

The test case we were working on verified that we could send a file over the socket. The code that sent the file looked like this:

```
private void writeSendFileCommand() throws IOException {
    os.writeObject("Sending");
    os.writeObject(itsFilename);
    os.writeLong(itsFileLength);
    char buffer[] = new char[(int) itsFileLength];
    fileReader.read(buffer);
    os.writeObject(buffer);
    os.flush();
}
```

But *why* were we sending a file? We were sending it to the `SMCRemoteServer` to be compiled. Then the server would send back the compiled file. Why had Jerry sent the "Sending" string first? He said that it was to alert the server that a file was coming. But we don't want to tell the server that a file is coming, we want to tell the server to compile a file, and send back the results.

I was thinking hard, but some part of my brain continued to recite the calming mantra. Almost as if in a trance, I walked to the wall and drew the following diagram:



I looked over at Jasmine, and noticed a smile flicker across her severe expression. "I like the way you are thinking Hotshot, but don't stop there."

There were four pieces of data being sent to the server. The file name, the file length, the file contents, and the "Sending" string. Why were these being sent separately? They were all part of the same packet of information, the same -- Transaction! That was it!

I shook myself out of my calming trance and made the following change to the test:

```
public void testCompileFile() throws Exception {
    File f = createTestFile("testSendFile", "I am sending this file.");
    c.setFilename("testSendFile");
    assertTrue(c.connect());
    assertTrue(c.prepareFile());
    assertTrue(c.compileFile());
    Thread.sleep(50);
    assertTrue(server.fileReceived);
    assertEquals("testSendFile", server.filename);
    assertEquals(23, server.fileLength);
    assertEquals("I am sending this file.", new String(server.content));
    f.delete();
}
```

Then I modified the old `sendFile` function as follows:

```
public boolean compileFile() {
    boolean fileSent = false;
    char buffer[] = new char[(int) itsFileLength];
    try {
        fileReader.read(buffer);
        CompileFileTransaction cft =
            new CompileFileTransaction(itsFilename, buffer);
        os.writeObject(cft);
        os.flush();
        fileSent = true;
    } catch (Exception e) {
        fileSent = false;
    }
    return fileSent;
}
```

I looked over at Jasmine, and saw that she was following along *very* closely. I couldn't read her expression through her raw intensity, but I was pretty sure I was on a good track.

Next I wrote the `CompileFileTransaction` class.

```
public class CompileFileTransaction implements Serializable {
    private String filename;
    private char contents[];
    public CompileFileTransaction(String filename, char buffer[]) {
        this.filename = filename;
        this.contents=buffer;
    }
    public String getFilename() {
        return filename;
    }
    public char[] getContents() {
        return contents;
    }
}
```

This allowed the project to compile. Of course the tests failed big time. So I made the following changes to the mock server in the test.

```
public void serve(Socket socket) {
    try {
        os = new PrintStream(socket.getOutputStream());
        is = new ObjectInputStream(socket.getInputStream());
        os.println("SMCR Test Server");
        os.flush();
        parse(is.readObject());
    } catch (Exception e) {
    }
}

private void parse(Object cmd) throws Exception {
    if (cmd != null) {
        if (cmd instanceof CompileFileTransaction) {
            CompileFileTransaction cft = (CompileFileTransaction) cmd;
            filename = cft.getFilename();
            content = cft.getContents();
            fileLength = content.length;
            fileReceived = true;
        }
    }
}
```

These changes made all the tests pass. I looked over at Jasmine and asked: "Is this what you had in mind?"

"Well, Hotshot, it's a start. It's certainly a lot better than converting all the data elements to strings the way Jerry did it. And it's also a lot better than shipping each data element as its own individual object."

"How would you make it better?" I asked, hoping to get her focus off me, and onto the project.

"Later." she said. "Right now, let's complete the transaction. You've got to get the client to accept server's response."

"That shouldn't be hard." I said. So I added the following three lines to the end of the `testCompileFile` test case:

```
File resultFile = new File("resultFile.java");
assertTrue("Result file does not exist", resultFile.exists());
resultFile.delete();
```

I ran the test and verified that it failed.

"After we call `compileFile` the results should be written into a file." I explained to Jasmine. "At the moment I don't care what's in that file I just want to make sure that it gets created."

"So how are you going to create it?" she challenged.

"I'll show you." I said, and I made the following change to the mock server in the test.

```
private void parse(Object cmd) throws Exception {
    if (cmd != null) {
        if (cmd instanceof CompileFileTransaction) {
            CompileFileTransaction cft = (CompileFileTransaction) cmd;
            filename = cft.getFilename();
            content = cft.getContents();
            fileLength = content.length;
            fileReceived = true;
            CompilerResultsTransaction crt =
                new CompilerResultsTransaction("resultFile.java");
            os.writeObject(crt);
            os.flush();
        }
    }
}
```

Then I made this compile by creating a skeleton of the `CompilerResultsTransaction` class.

```
public class CompilerResultsTransaction implements Serializable {
    public CompilerResultsTransaction(String filename) {

    }

    public void write() {

    }
}
```

Of course the test still failed. So I made the following change to `compileFile`.

```
public boolean compileFile() {
    boolean fileCompiled = false;
    char buffer[] = new char[(int) itsFileLength];
    try {
        fileReader.read(buffer);
        CompileFileTransaction cft =
            new CompileFileTransaction(itsFilename, buffer);
        os.writeObject(cft);
        os.flush();
        Object response = is.readObject();
        CompilerResultsTransaction crt = (CompilerResultsTransaction) response;
        crt.write();
        fileCompiled = true;
    } catch (Exception e) {
        fileCompiled = false;
    }
    return fileCompiled;
}
```

And then finally I fleshed out the transaction.

```
public class CompilerResultsTransaction implements Serializable {
    private String filename;
    public CompilerResultsTransaction(String filename) {
```

```

        this.filename = filename;
    }

    public void write() throws Exception {
        File resultFile = new File(filename);
        resultFile.createNewFile();
    }
}

```

"Good enough for now." Jasmine said, and leaned back in her chair. "I'm going to go on break in a few minutes while you get the `CompilerResultsTransaction` to actually write the file instead of just create it. Also, I'd like you to do a bit of cleanup of the code. There's a lot of cruft left behind from when you and Jerry were messing around with sending strings and integers. But before I go, I'd like your thoughts on that `instanceof` you used in the mock server."

"That was the simplest thing I could think of to check whether or not the incoming object was a `CompileFileTransaction`." I said. "Is there something wrong with it?"

She stood up, preparing to leave. She looked down at me and said: "No, nothing horribly wrong. But do you think that's what the real server is going to do? Do you think the real server will have a long `if/else` chain of `instanceof` expressions to parse the incoming transactions?"

"I hadn't really thought that far ahead." I said.

"No." she said flatly. "I imagine you hadn't." And she strode out of the room.

The room felt distinctly empty without her in it, as though my presence didn't count at all. I let out a long sigh, and shook my head. I could see that the working with Jasmine was going to be both exhausting and educational in many different ways.

One thing I was sure of, I *really* hated being called Hotshot. I sighed again, turned back to the computer, and started working on the tasks he gave me.

To be continued.

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

www.objectmentor.com/resources/articles/CraftsmanCode/Craftsman_13_SMCRemote_III_Objects.zip

p

The Craftsman: 15

SMCRemote Part V

Ess Are Pee

Robert C. Martin
18 June, 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/Craftsman_IV_SMCRemote_IV_Transactions.zip

The last thing I saw, before the door slid shut, was Jasmine's long black hair swaying and bouncing to the rhythm of her purposeful stride. As her saturating presence drained from the room, I felt my lungs release the breath I hadn't known they were holding. My eyes lost their focus, and for several quasi-conscious minutes I just sat and blindly gazed at the blur that was the door.

"Whew!" I said to myself. "Working with Jasmine is *not* going to be easy."

"Easy or not, " I replied to myself, "*she asked you to get that file written and to clean up the code. So you'd better get cracking.*"

I sighed. "Agreed, agreed." I agreed with myself. "I can just imagine what Jasmine would say if I didn't have something to show her. She'd say: '*Some Hotshot you turned out to be! What did you do, just sit here the whole time doing nothing?*'"

"Okay, Jasmine, okay. So, what should we do first?"

"Well, Hotshot, we have to get the `CompilerResultsTransaction` to carry the contents of a file from the server to the client; and then write that file on the client."

"Oh right." I said. "The compiler will create an output file on the server, and we have to move it to the client. This is just what we did in the `CompileFileTransaction`."

```
public class CompileFileTransaction implements Serializable {
    private String filename;
    private char contents[];
    public CompileFileTransaction(String filename, char buffer[]) {
        this.filename = filename;
        this.contents=buffer;
    }
    public String getFilename() {
        return filename;
    }
    public char[] getContents() {
        return contents;
    }
}
```

"We opened and read the file in the `compileFile` method, and then constructed and passed the file name and the array of chars into the constructor to the `CompileFileTransaction`."

```
public boolean compileFile() {
    char buffer[] = new char[(int) itsFileLength];
    try {
        fileReader.read(buffer);
        CompileFileTransaction cft =
            new CompileFileTransaction(itsFilename, buffer);
```

"Right, Hotshot, that's just what we did. Is there something you don't like about that?"

"I don't like having to write the same code twice. Jerry made a big point out of not duplicating code."

"Jerry's not exactly the sharpest knife in the drawer, Hotshot".

"Maybe not, but I think he was right about that. So I think I want to write a class that carries a file across the socket."

"Something like a `FileCarrier`?"

"Yeah, that's a good name for it!"

"OK, Hottie, write a test for it."

"Hottie? -- Er, OK. How about this?"

```
public class FileCarrierTest extends TestCase {
    public void testAFile() throws Exception {
        final String TESTFILE = "testFile.txt";
        final String TESTSTRING = "test";
        createFile(TESTFILE, TESTSTRING);
        FileCarrier fc = new FileCarrier(TESTFILE);
        fc.write();
        assertTrue(new File(TESTFILE).exists());
        String contents = readFile(TESTFILE);
        assertEquals(TESTSTRING, contents);
    }
}
```

"Yes, very good, Hot Stuff, keep going."

"OK, beautiful. Here are the two utility functions..."

```
private String readFile(final String TESTFILE) throws IOException {
    BufferedReader reader = new BufferedReader(new FileReader(TESTFILE));
    String line = reader.readLine();
    return line;
}

private void createFile(final String TESTFILE,
                        final String TESTSTRING) throws IOException {
    PrintWriter writer = new PrintWriter(new FileWriter(TESTFILE));
    writer.println(TESTSTRING);
    writer.close();
}
```

"...and here is the stubbed implementation of `FileCarrier` that will make this test compile and fail.

```
public class FileCarrier {
    public FileCarrier(String fileName) {
    }

    public void write() {
    }
}
```

```
}
```

"So now, Jazzy-wazzy, all we have to do to make the test pass is read the file in the constructor and write it in the `write()` function."

"Not yet, over-temp, first run the test to make sure it will fail."

"Jay-girl, there's no implementation in `FileCarrier`. Of course it's going to fail.

"Well, exotherm, you and I both know that. But does the program?"

"I love it when you make me run a test! OK, here goes."

*** the test passes ***

"Huh? What? How can that be? Do you understand that Jazz?"

"Gosh, boiling point, I sure don't. How can the test pass when there's nothing in FileCarrier to..."

"Oh. Egad. Are we dumb or what. We never deleted the test file."

"Ah, yes, I see that now, Mr. tepid breath. So why don't you fix that?"

"OK, here."

```
public void testAFile() throws Exception {
    final String TESTFILE = "testFile.txt";
    final String TESTSTRING = "test";
    createFile(TESTFILE, TESTSTRING);
    FileCarrier fc = new FileCarrier(TESTFILE);
    new File(TESTFILE).delete();
    fc.write();
    assertTrue(new File(TESTFILE).exists());
    String contents = readFile(TESTFILE);
    assertEquals(TESTSTRING, contents);
}
```

"OK, great, now it fails. But -- latent-heat -- can you make it pass?"

"Sure I can, JJ, just watch me!"

```
public class FileCarrier implements Serializable {
    private String fileName;
    private char[] contents;

    public FileCarrier(String fileName) throws Exception {
        File f = new File(fileName);
        this.fileName = fileName;
        int fileSize = (int)f.length();
        contents = new char[fileSize];
        FileReader reader = new FileReader(f);
        reader.read(contents);
        reader.close();
    }

    public void write() throws Exception {
        FileWriter writer = new FileWriter(fileName);
        writer.write(contents);
        writer.close();
    }
}
```

"Yee Ha! electron-volt, you are cooking now!"

"Why, thanks! Gorgeous! But you haven't seen anything yet. Watch how I integrate the `FileCarrier` into the `CompileFileTransaction`! That should turn your head."

```
public class CompileFileTransaction implements Serializable {
```

```

FileCarrier sourceFile;
public CompileFileTransaction(String filename) throws Exception {
    sourceFile = new FileCarrier(filename);
}
public String getFilename() {
    return sourceFile.getFileName();
}
public char[] getContents() {
    return sourceFile.getContents();
}
}

```

"And now I'll change the compileFile function to use the new CompileFileTransaction!"

```

CompileFileTransaction cft = new CompileFileTransaction(itsFilename);
os.writeObject(cft);
os.flush();
Object response = is.readObject();
CompilerResultsTransaction crt = (CompilerResultsTransaction)response;
crt.write();

```

"And now I'll run all the tests and ... see? They all pass!"

"Oh, fever man, stop it, stop it, you're making me crazy!"

"It's all part of my plan, sweet-eyes, all part of my plan. Now, watch with bated breath as I put the FileCarrier into the CompilerResultsTransaction!"

"Ooooh!"

```

public class CompilerResultsTransaction implements Serializable {
    private FileCarrier resultFile;
    public CompilerResultsTransaction(String filename) throws Exception {
        resultFile = new FileCarrier(filename);
    }

    public void write() throws Exception {
        resultFile.write();
    }
}

```

"Gasp!"

"That's right! And look how masterfully I change the test to use the new transaction!"

```

private void parse(Object cmd) throws Exception {
    if (cmd != null) {
        if (cmd instanceof CompileFileTransaction) {
            CompileFileTransaction cft = (CompileFileTransaction) cmd;
            filename = cft.getFilename();
            content = cft.getContents();
            fileLength = content.length;
            fileReceived = true;
            TestSMCRemoteClient.createTestFile("resultFile.java", "Some content.");
            CompilerResultsTransaction crt =
                new CompilerResultsTransaction("resultFile.java");
            os.writeObject(crt);
            os.flush();
        }
    }
}

```

"More, Alphonse, Please, More!"

You want more Jasmine? I can give you more. Lots more. You see Jasmine, I *know* about objects. Oh yes, I know about them. `FileCarrier` is an *Object* Jasmine. Do you see how it can be used in more than one place, Jasmine? Do you see how it encapsulates a single responsibility, Jasmine? Do you? Do you know the **Single Responsibility Principle** Jasmine? Have you heard of it? Have you studied it? Well I have. Do you know what it says, Jasmine? It says that a class should have one and only one reason to change, Jasmine. It says that all the functions and variables of a class should work together towards a single goal, Jasmine. It says that a class should not try to accomplish more than one goal...Are you *listening* Jasmine?

"Yes, Alphonse. I'm listening. Please, don't stop!"

"You'd better not ask me to stop! You see, Jasmine, those of us in the know call this principle the SRP. That's ESS ARE PEE Jasmine.

"ESS ARE PEE, Alphonse. ess are pee."

"Remember the `compileFile` function Jasmine, remember how it used to read the file and pass a `char` array into the `CompileFileTransaction`? You asked me what I didn't LIKE about that function. Well I'll TELL you what I didn't LIKE about it, Jasmine; it was VIOLATING the SRP! It had TWO reasons to change, instead of ONE. It depended BOTH on the details of reading a file, AND on the policy of building and sending transactions. That's TOO MUCH RESPONSIBILITY Jasmine."

"You're scaring me, Alphonse."

"You SHOULD be scared, Jasmine. I'm..."

"Oh! Alphonse!"

~ ~ ~

Then several things happened at once:

1. I noticed that the door was open.
2. I noticed that the chair next to me was empty.
3. I heard the echoes of my falsetto imitation of Jasmine still reverberating off the walls.
4. I saw Jasmine standing in the doorway. Her eyes were cold steel.

To be continued...maybe.

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

www.objectmentor.com/resources/articles/CraftsmanCode/Craftsman_15_SMCRemote_V_EssArePee.zip

The Craftsman: 16

SMCRemote Part VI

Wham Bam

Robert C. Martin
18 July, 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/Craftsman_15_SMCRemote_V_EssArePee.zip

Jasmine just stood there staring daggers at me. I tried to lower my head, but I couldn't break contact with her eyes. After about half a minute she rolled those eyes, and stared at the ceiling, tapping her foot. Finally, she shook her head, squared her shoulders and strode over to me. She had an odd, sad, look on her face.

"Alphonse," she said sternly, "that is territory that you are never to tread again. Do you understand me?"

Thoroughly ashamed of myself, I nodded and said: "Yes Jasmine. I'm -- uh -- I'm sorry."

"And from now on, call me Ma'am."

"Yes,... Ma'am." Oh god this was excruciating.

She gave an wry little snort that made her hair bounce and shimmer and then she said: "OK, let's see what you've done."

I showed her the `FileCarrier` code, and the tests I had written. She seemed satisfied at first but then said: "Look at the way `FileCarrier` reads and writes the file.

```
public class FileCarrier implements Serializable {
    private String fileName;
    private char[] contents;

    public FileCarrier(String fileName) throws Exception {
        File f = new File(fileName);
        this.fileName = fileName;
        int fileSize = (int)f.length();
        contents = new char[fileSize];
        FileReader reader = new FileReader(f);
        reader.read(contents);
        reader.close();
    }

    public void write() throws Exception {
        FileWriter writer = new FileWriter(fileName);
        writer.write(contents);
        writer.close();
    }
}
```

```

    }

    public String getFileName() {
        return fileName;
    }
    public char[] getContents() {
        return contents;
    }
}

```

"FileCarrier reads the file with a single read, and writes it with a single write. This works for small examples, but there are a few problems with it. First of all, I'm not convinced that the read won't abort early, filling only part of the array. Secondly, the carried file is going to be transmitted over a socket to another system. That system may use a different line-end character. So I don't think FileCarrier will work well across foreign systems."

I was listening. I was listening *hard*. But some disconnected part of me watched raptly as she focused those black eyes on the screen. Her lashes, nose, and lips danced as she spoke.

"What do you think we should do about that Alphonse?"

I didn't miss a beat. "Well, -- er -- *Ma'am*, we should probably read and write the files a line at a time, and carry the files in a list of lines."

"All right Alphonse, why don't you make that change?"

I directed my attention to the keyboard and, bit-by-bit, I made the appropriate changes to the FileCarrier. I took special care to keep the tests running. Once everything was working I refactored the class so that it read as clearly and cleanly as I could make it. I was on my very best behavior.

```

public class FileCarrier implements Serializable {
    private String fileName;
    private LinkedList lines = new LinkedList();

    public FileCarrier(String fileName) throws Exception {
        this.fileName = fileName;
        loadLines();
    }

    private void loadLines() throws IOException {
        BufferedReader br = makeBufferedReader();
        String line;
        while ((line = br.readLine()) != null)
            lines.add(line);
        br.close();
    }

    private BufferedReader makeBufferedReader()
        throws FileNotFoundException {
        return new BufferedReader(
            new InputStreamReader(
                new FileInputStream(fileName)));
    }

    public void write() throws Exception {
        PrintStream ps = makePrintStream();
        for (Iterator i = lines.iterator(); i.hasNext();)
            ps.println((String) i.next());
        ps.close();
    }

    private PrintStream makePrintStream() throws FileNotFoundException {
        return new PrintStream(
            new FileOutputStream(fileName));
    }
}

```

```

    public String getFileName() {
        return fileName;
    }
}

```

"That's very good, Alphonse."

"Thank you Ma'am." I suppressed a cringe.

"However, I don't think FileCarrierTest really ensures that FileCarrier faithfully reproduces the file. I'd like to see some more tests."

She was being too polite. I was starting to wish she'd call me Hot Shot again.

"I agree, Ma'am. I'll improve the test."

Once again I build the code bit-by-bit, keeping the tests running as I made the changes. Once again I refactored with great care until I was very sure that the code was as clean and expressive as I could make it. I didn't want to give Ma'am any more reason to be angry or disappointed.

```

public class FileCarrierTest extends TestCase {
    public void testFileCarrier() throws Exception {
        final String ORIGINAL_FILENAME = "testFileCarrier.txt";
        final String RENAMED_FILENAME = "testFileCarrierRenamed.txt";
        File originalFile = new File(ORIGINAL_FILENAME);
        File renamedOriginal = new File(RENAMED_FILENAME);

        ensureFileIsRemoved(originalFile);
        ensureFileIsRemoved(renamedOriginal);

        createTestFile(originalFile);
        FileCarrier fc = new FileCarrier(ORIGINAL_FILENAME);
        rename(originalFile, renamedOriginal);
        fc.write();

        assertTrue(originalFile.exists());
        assertTrue(filesAreTheSame(originalFile, renamedOriginal));

        originalFile.delete();
        renamedOriginal.delete();
    }

    private void rename(File oldFile, File newFile) {
        oldFile.renameTo(newFile);
        assertTrue(oldFile.exists() == false);
        assertTrue(newFile.exists());
    }

    private void createTestFile(File file) throws IOException {
        PrintWriter w = new PrintWriter(new FileWriter(file));
        w.println("line one");
        w.println("line two");
        w.println("line three");
        w.close();
    }

    private void ensureFileIsRemoved(File file) {
        if (file.exists()) file.delete();
        assertTrue(file.exists() == false);
    }

    private boolean filesAreTheSame(File f1, File f2) throws Exception {
        FileInputStream r1 = new FileInputStream(f1);
        FileInputStream r2 = new FileInputStream(f2);
    }
}

```

```

    try {
        int c;
        while ((c = r1.read()) != -1) {
            if (r2.read() != c) {
                return false;
            }
        }
        if (r2.read() != -1)
            return false;
        else
            return true;
    } finally {
        r1.close();
        r2.close();
    }
}
}
}

```

Ma'am studied the code as I wrote it. She never looked at me -- never once. As uncomfortable as they had made me, her penetrating glares and snide remarks were better than this formal etiquette. The tension between us was palpable.

"That's very nice, Alphonse. Good clean code. I especially like the way you made sure that the original file was renamed, and that the new file was created. The way you've written it, there can't be any doubt that the `FileCarrier` is creating the file. There's no way the old file could have been left behind.

"The one problem I have with it is that I haven't seen the `filesAreTheSame` method fail. Do you think it really works properly?"

I carefully examined the code. I couldn't see any flaw. I was pretty sure it had to work. But I wasn't going to make any assertions without evidence. So I started writing some tests for the `filesAreTheSame` method.

I began by writing a test that showed that the method worked for two files that were equal. Then I wrote a test that showed that two different files did not compare the same. I wrote a test that showed that if one file were a prefix of the other, they would not compare. In the end I wrote five different test cases.

These five test cases had a *lot* of duplicate code. Each wrote two files. Each compared the two files. Each deleted the two files. To get rid of this duplication I used the Template Method pattern. I moved all the common code into an abstract base class called `FileComparator`. I moved all the differentiating code into simple anonymous derivatives. Thus, each test case created a new anonymous derivative that specified nothing more than the contents of the two files, and the sense of the comparison.

As always, I wrote this code one tiny step at a time, running the tests between each step, and carefully refactoring whenever I could. Ma'am never took her eyes off the screen. She was purposefully avoiding any informal contact. Once, our elbows accidentally touched. Shivers ran through me; but she didn't show any reaction at all. A few minutes later she moved her chair a few centimeters further from mine.

```

public class FileCarrierTest extends TestCase {
    private abstract class FileComparator {
        abstract void writeFirstFile(PrintWriter w);
        abstract void writeSecondFile(PrintWriter w);

        void compare(boolean expected) throws Exception {
            File f1 = new File("f1");
            File f2 = new File("f2");
            PrintWriter w1 = new PrintWriter(new FileWriter(f1));
            PrintWriter w2 = new PrintWriter(new FileWriter(f2));
            writeFirstFile(w1);
            writeSecondFile(w2);
            w1.close();
            w2.close();
            assertEquals("(f1,f2)", expected, filesAreTheSame(f1, f2));
        }
    }
}

```

```

        assertEquals("(f2,f1)", expected, filesAreTheSame(f2, f1));
        f1.delete();
        f2.delete();
    }
}

public void testOneFileLongerThanTheOther() throws Exception {
    FileComparator c = new FileComparator() {
        void writeFirstFile(PrintWriter w) {
            w.println("hi there");
        }

        void writeSecondFile(PrintWriter w) {
            w.println("hi there you");
        }
    };
    c.compare(false);
}

public void testFilesAreDifferentInTheMiddle() throws Exception {
    FileComparator c = new FileComparator() {
        void writeFirstFile(PrintWriter w) {
            w.println("hi there");
        }

        void writeSecondFile(PrintWriter w) {
            w.println("hi their");
        }
    };
    c.compare(false);
}

public void testSecondLineDifferent() throws Exception {
    FileComparator c = new FileComparator() {
        void writeFirstFile(PrintWriter w) {
            w.println("hi there");
            w.println("This is fun");
        }

        void writeSecondFile(PrintWriter w) {
            w.println("hi there");
            w.println("This isn't fun");
        }
    };
    c.compare(false);
}

public void testFilesSame() throws Exception {
    FileComparator c = new FileComparator() {
        void writeFirstFile(PrintWriter w) {
            w.println("hi there");
        }

        void writeSecondFile(PrintWriter w) {
            w.println("hi there");
        }
    };
    c.compare(true);
}

public void testMultipleLinesSame() throws Exception {
    FileComparator c = new FileComparator() {
        void writeFirstFile(PrintWriter w) {

```

```
        w.println("hi there");
        w.println("this is fun");
        w.println("Lots of fun");
    }

    void writeSecondFile(PrintWriter w) {
        w.println("hi there");
        w.println("this is fun");
        w.println("Lots of fun");
    }
};
c.compare(true);
}
```

"Alphonse, this is very nice."

I wanted to scream.

"I love the way you used the Template Method pattern to eliminate duplication. I'm glad you are taking your studies so seriously. Many apprentices don't learn their patterns until their journeymen force them to.

"I also like the way you tested the commutivity of equality. Every comparison happens in both directions. Splendid!"

Egad! When was she going to complain about something? Where had Jasmine gone?

"Thank you, Ma'am."

To be continued...

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

www.objectmentor.com/resources/articles/CraftsmanCode/Craftsman_16_SMCRemote_VI_Wham_Bam.zip

The Craftsman: 17

SMCRemote Part VII

Call the Guards

Robert C. Martin
17 Aug, 2003

...Continued from last month. You can download last month's code from:
www.objectmentor.com/resources/articles/CraftsmanCode/Craftsman_16_SMCRemote_VI_Wham_Bam.zip

Dear Diary, my first week as an apprentice for Mr. C. is over. I've learned a lot in that week, and I've also made a real mess of things. Here's a recap: On Monday Jerry had me write a program to generate prime numbers. On Tuesday he had me write a program to generate prime factors. Wednesday was spent getting the SocketServer working. Thursday we started working on SMCRemoteClient, and I Micahed Jerry. I met Jasmine that afternoon in the Journeymen's lounge. Friday was the most embarrassing day of my life. I haven't seen Jasmine (Ms J) since we finished working Friday afternoon. What a roller coaster of a week. (Diary, just what is a roller coaster anyway?)

I spent the weekend pretty depressed. I was sure that I was going to be transferred to sanitation & recycling, reactor cleanup, or mildew scrubbing. I barely bothered to get dressed at eat.

Anyway, now it's Monday evening, and I'm writing to tell you about the first day of my second week. It didn't turn out badly at all. In fact, I'm starting to feel optimistic again.

I woke up on Monday full of dread. After breakfast I shambled over to the workroom expecting to see Ms. J waiting for me at my workstation. Instead I saw a plumpish middle-aged woman with shoulder length blonde hair, and a grandmother smile that spoke of younger years. When she looked at me her eyes twinkled and she gave me a wide grin. "You must be Alphonse." She said, holding out her hand. "My name is Jean. I've heard a great deal about you. Are you hungry? Young men like you are always hungry. I've got a nice sandwich here in my basket if you'd like it, or perhaps you'd like an apple or a banana.

There was, indeed, a basket on the floor next to her. It looked like it contained quite a bit more than just sandwiches and bananas. I shook her hand and said: "Yes Ma'am -- I mean, no Ma'am -- I mean -- Yes I am Alphonse, and no thank you for the sandwich, and I'm -- er -- pleased to meet you."

She looked at me sternly, yet with a big motherly grin. "Now we'll have none of that Ma'am nonsense. I'm certainly not anybody's Ma'am. The very idea! You just call me Jean, dear."

"Uh, well, thank you Jean. Can you tell me where Ms. J is?"

"Who dear?"

"er -- Jasmine. She and I are supposed to be working together."

"Goodness me, I've never heard her called Ms. J before. Did she ask you to call her that? I don't think anyone else calls her anything but Jasmine. Such a lovely girl, isn't she? Anyway, dear, Mr. C. asked her to work on a different project for the time being, and so I'll be working with you from now on."

"You?" I was caught completely unprepared for this. "Oh." I said intelligently. "Uh..."

"Now you sit right down here, dear, and let me tell you what I've been thinking."

"Thinking?"

"Yes dear! I've been looking over this program of yours for the last half-hour -- I like to get to work early you know -- not that you need to get here any earlier than usual -- I know a growing boy needs his sleep and breakfast. Anyway I've been quite pleased with this program. You've got a very pleasant suite of tests, and the code is quite easy to read and, really quite nicely structured, I'd say. But there's one thing that puzzles me dear."

"Uh -- puzzles?"

"Yes, dear! I've been looking all through this program and I can't find the main function. When were you going to write it dear?"

"Uh, well, Jerry said..."

"Oh let me just guess what Jerry said. Jerry's a nice lad, dear, but sometimes I think he could use a few more clues about things. But I shouldn't be saying anything bad about anyone. Jerry is a fine programmer, dear, and you just never mind about what I said. Now, let's write that main function. Would you like to start? My fingers are feeling a bit stiff this morning. Take my advice and don't get old, dear."

Dazed by the sheer inundation of words I took the keyboard and began typing:

```
public void main(
```

"Oh, now, dear! How long have you been working here? You've got to write a test first, dear. You can't just go off writing the main function! Where would we be if everyone just wrote the functions without writing the tests first? I can tell you where -- In a pickle, that's where. No, dear, you delete that and write a test first.

She took a pair of knitting needles out of her basket and began working on a project of some kind. It looked vaguely like a shawl. She started softly humming a simple tune while I deleted the characters I had written and started over.

How do you test the main function? I guess the best answer is simply to call it and then make sure it did what it was supposed to do. The main function interprets the command line options, so calling it is just a matter of passing in the right options. So I started typing again:

```
public void testMain() throws Exception {  
    SMCRemoteClient.main(new String[]{"myFile.sm"});  
}
```

Jean looked up from her knitting and said: "That's nice dear, but where does `myFile.sm` come from? We can't just have that laying around, can we? No, we'll have to create it right here, won't we dear? And don't forget to delete it when you are done, dear. Nothing worse than a bunch of old files hanging around, I always say."

So I continued typing:

```
public void testMain() throws Exception {  
    File f = createTestFile("myFile.sm", "the content");  
    SMCRemoteClient.main(new String[]{"myFile.sm"});  
    f.delete();  
    File resultFile = new File("resultFile.java");  
    assertTrue(resultFile.exists());  
    resultFile.delete();  
}
```

Jean had finished another row of her shawl while I typed. She looked up just as I finished and said: "Now, dear, that's just fine, but I do think that `main` might not have enough time to finish executing before that `delete` takes effect. Remember, dear, you've got all those socket threads running, and one of them could easily still be going by the time `main` returns. No, dear, don't do anything about it now; just keep it in mind. Now I think you'd better write `main`, don't you?"

I thought to myself that this was a pretty sharp old lady, and I started writing the main function.

```

public static void main(String[] args) {
    SMCRremoteClient client = new SMCRremoteClient();
    client.setFilename(args[0]);
    if (client.prepareFile())
        if (client.connect())
            if (client.compileFile())
                client.close();
            else { // compileFile
                System.out.println("failed to compile");
            }
        else { // connect
            System.out.println("failed to connect");
        }
    else { // prepareFile
        System.out.println("failed to prepare");
    }
}

```

"Oh my, now nice that looks! I think it's very clever how you commented those `else` statements. Though I wonder if it wouldn't be more readable if you inverted the sense of the `if` statements and used them as guards. No, dear, don't change it yet. Let's see if it works first. It's never worth making lots of changes until you know whether the program work or not, don't you agree? First make it work, *then* make it right."

So I ran the test, and it worked first time.

"Oh, that's just splendid!" She said while glancing up from her knitting. "Now, let's change that `if` statement."

So I changed the function so that the `if` statements were guards.

```

public static void main(String[] args) {
    SMCRremoteClient client = new SMCRremoteClient();
    client.setFilename(args[0]);
    if (!client.prepareFile()) {
        System.out.println("failed to prepare");
        return;
    }
    if (!client.connect()) {
        System.out.println("failed to connect");
        return;
    }
    if (!client.compileFile()) {
        System.out.println("failed to compile");
        client.close();
        return;
    }
    client.close();
}

```

Jean put her knitting down into her basket and looked carefully at the code. "Well I do think that looks a little better, though I can't say I care for that duplicated `close` statement. And the violation of single/entry, single/exit is a bit bothersome. Still, it's better than all that indentation, don't you agree? Of course we could change those three functions to throw exceptions, but then we'd have to catch them, and that would be a bother. No, let's leave it like that for the time being. Now, I think it's time for a break, don't you? Would you like to carry my basket to the break room for me? I'm always over-packing it and it gets so heavy after awhile you know."

To be continued...

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

www.objectmentor.com/resources/articles/CraftsmanCode/Craftsman_17_Call_the_Guards.zip

The Craftsman: 18

SMCRemote Part VIII

Slow and Steady

Robert C. Martin
24 Sept, 2003

*...Continued from last month. You can download last month's code from:
www.objectmentor.com/resources/articles/CraftsmanCode/Craftsman_17_Call_the_Guards.zip*

"All right then dear, I think it's time we started to work on the server part of this application, don't you? After all the client portion seems to be working as well as we can tell, and a client without a server just seems so lonely. Don't you agree?"

Jean had just settled her ample behind into her seat. As she spoke she reached into her bag and pulled out her knitting. The shawl (or whatever it was) had grown considerably during our break (during which I had learned all about Jean's children, grandchildren, nephews, and nieces)

"Er, server?" Two words were all I could ever seem to muster in response to one of Jean's vocal deluges.

"Yes dear, the server. The server is going to have to receive that file you've been so diligently sending, save it to disk, and then compile it with SMC. Remember dear, we are building a remote compiler for SMC, the State Machine Compiler. Now, dear, what's the first test case we should write?"

What test case indeed? I thought furiously about this. The server side of the pair of programs had to listen at a socket and receive `CompileFileTransaction` objects. These objects contain encapsulated files in `FileCarrier` sub-objects. The server needs to write these encapsulated files to a safe place on the disk, compile them, and then send the results back to the client in a `CompilerResultsTransaction` object.

My first worry was how to compile the files. Somehow we needed to invoke the compiler and get it to compile the saved file.

"Uh, could we write a test case for invoking the compiler?"

Jean smiled the way a grandmother smiles at an infant taking its first steps. "Why of course, dear, that would be a lovely place to start. Do you know how to invoke the compiler? Let's see, I think we just have to build the right command line and then invoke it. Now what was that command line syntax? It's been so long since I used SMC, I don't quite remember. Hear, dear, type this command: `java smc.Smc`. That's right dear, now what does that error message say? Class def not found? Oh yes, we forgot the classpath. Don't get old dear; you start forgetting things. Type this: `java -cp C:/SMC/smc.jar smc.Smc`.¹ That's better, now what does that message say?"

The message on the screen was usage error that described all the command line arguments for SMC. "Er, it's a usage message."

¹ You can download `smc.jar` from:
http://www.objectmentor.com/resources/downloads/smc_java

"So it is, dear, so it is. And it looks to me as though we want to invoke SMC with the following command: `java -cp C:/SMC/smc.jar smc.Smc -f myFile.sm`. Don't you agree dear? Why don't you write a test case that will generate that command line?"

So I created a new unit test class named `SMCRemoteServerTest`.

```
public class SMCRemoteServerTest extends TestCase {
    public void testBuildCommandLine() throws Exception {
        assertEquals("java -cp C:/SMC/smc.jar -f myFile.sm",
            SMCRemoteServer.buildCommandLine("myFile.sm")) ;
    }
}
```

And I wrote the `SMCRemoteServer` class too.

```
public class SMCRemoteServer {
    public static String buildCommandLine(String file) {
        return null;
    }
}
```

"Very good, dear. The test fails, as it should. You've been learning your lessons well. Now it should be a simple matter to make that test pass, don't you think?"

"Er -- sure."

```
public static String buildCommandLine(String file) {
    return "java -cp C:/SMC/smc.jar smc.Smc -f " + file;
}
```

"That's fine dear, now run it for me, won't you? Good. Oh! Why, the test failed dear, what could be the matter?"

The message on the screen was: `expected:<.....> but was:<...smc.Smc ...>`. I looked carefully at the code and realized that I had forgotten the `smc.Smc` in the test case. "Er, I guess I wrote the test wrong."

"So you did, so you did. Yes, sometimes we do write the tests wrong. Bugs can occur anywhere. That's why it's always a good idea to write both tests and code. That way you are writing everything twice. I have a nephew who does accounting in ship stores, and he always enters every transaction into two distinct ledgers. What did he call that? Oh, yes, *dual entry bookkeeping*. He says it helps him prevent and track down errors. And that's just what we're doing, isn't it dear. We write every bit of functionality twice. Once in a test, and once in code. And it *does* help us to prevent and track down errors doesn't it dear?"

While she was droning, I fixed the test case. Jean is a nice old lady, and pretty smart too, but my ears were ringing from the continual chatter.

```
public void testBuildCommandLine() throws Exception {
    assertEquals("java -cp C:/SMC/smc.jar smc.Smc -f myFile.sm",
        SMCRemoteServer.buildCommandLine("myFile.sm")) ;
}
```

This made the test pass.

"Wonderful, dear, wonderful; but the code is a bit messy, don't you think? Let's clean it up before we go any further. Clean your messes before they start, I always say."

I looked at the code and didn't see anything particularly messy. So I looked back at Jean and said: "Er, where would you like to start?"

"My goodness dear; why it's as plain as the nose on your face! That `buildCommandLine` function needs a bit of straightening. That string in the return statement is just full of unexplained assumptions that are sure to confuse the next person to read this code. We don't want to confuse anyone, do we? I should

say we don't. Here, dear, give me the keyboard."

She set down her knitting (is that the beginnings of a sleeve) and with considerable effort took control of the keyboard. Her typing was slow but deliberate, as if each keystroke caused her pain. Eventually she changed the return statement as follows:

```
return "java -cp " + "C:/SMC/smc.jar" + " " + "smc.Smc" + " -f " + file;
```

Then she ran the test, and it passed.

"There, dear, do you see now? We need to replace some of those strings with constants that explain how the command line is built. Do you want to do it dear, or shall I?"

Her pained expression was answer enough. I took the keyboard and added the constants I thought she was after.

```
public class SMCRemoteServer {  
    private static final String SMC_CLASSPATH = "C:/SMC/smc.jar";  
    private static final String SMC_CLASSNAME = "smc.Smc";  
  
    public static String buildCommandLine(String file) {  
        return "java -cp " +  
            SMC_CLASSPATH + " " +  
            SMC_CLASSNAME +  
            " -f " + file;  
    }  
}
```

"Yes, dear, that's just what I was after. Don't you think that others will make a bit more sense out of it now? I know I would if I were them. Now dear, why don't you and I go to the break room and settle our brains?"

Something inside me snapped.

"Jean, we've barely gotten anything done!"

"Why dear, whatever do you mean, we've gotten quite a bit done."

I wasn't thinking about whom I was talking to. I kept right on blathering. "We've only gotten one stupid function done. If we keep moving at this snail's pace, Mr. C is going to transfer us to the husbandry deck to clean out the Dribin cages!"

"Really dear, why ever would you think such a thing? I've been working in this department for well over thirty years, dear, and nobody has ever suggested that I should be cleaning up after Dribins."

She stopped for a moment as if composing her thoughts. Then she looked at me with a kindly, but stern, expression on her face.

"Alphonse dear, the only way to get done with a program quickly, is to do the very best job you can with it. If you rush, or if you take shortcuts, you will pay for it later in debugging time. Mr. C. is paying you for your time, dear, and he wants the very best work you can do. The code is your product, dear, and Mr. C. doesn't pay for poor products. He doesn't want to hear that you saved a day by being sloppy because he knows that he'll have to pay dearly for that sloppiness later. What he wants is the very best code you can write. He wants that code to be as clean, expressive, and well tested as you can make it. And, dear, Mr. C. knows that if you do that, you'll be done with it quicker than all those young fools who think they can go faster by cutting corners. No, dear, we aren't going too slow. We're moving along at just the right pace to get done as quickly as possible. Now, let's go get a cup of tea, shall we?"

To be continued...

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

www.objectmentor.com/resources/articles/CraftsmanCode/Craftsman_17_Slow_And_Steady.zip

The Craftsman: 19

SMCRemote Part IX

Tolerance

Robert C. Martin
16 Oct, 2003

*...Continued from last month. You can download last month's code from:
www.objectmentor.com/resources/articles/CraftsmanCode/Craftsman_18_Slow_And_Steady.zip*

Jean took me down the turbo to the journeyman's lounge at .36g in the gamma arm. She said the low-g did wonders for her joints. When we got there I noticed Jerry, Jasmine, and a few other journeymen clustered at a table. Jean noticed them too and started walking over to them. I followed reluctantly.

"Hello my dears!" she exclaimed as though she hadn't seen them in weeks. They all greeted her with similar warmth. Then she excused herself and trotted over to a vacant head, leaving me to face my two previous mentors alone.

"Hello Jerry; Ms. J. How are you?"

Jerry looked perplexed. "What's this Ms. J. stuff about?"

Jasmine impatiently brushed this aside and said: "Drop the Ms. J stuff hotshot. We've had enough of that." Her eyes were as disarming as ever, and I found myself unable to answer her intelligently.

A coffee servitor rolled by. Grateful for the distraction I grabbed a cup and then faced my mentors again.

"How's Jean treating you hotshot?" Jasmine asked.

Jerry leaped in with: "Yeah, how'd you score a gig with *her*?"

After barely being able to say more than three words at a time for the whole morning with Jean, my frustrations, came pouring out.

"I didn't score anything, she just showed up this morning. Frankly working with her is a little frustrating. She's always calling me "Dear"; and she talks a *lot* and takes a *lot* of breaks. We barely get anything done. I'm not sure I want to keep working with her."

Jasmine and Jerry looked at me, then at each other, then starting laughing. Jerry settled down quickly; but Jasmine couldn't seem to gain control of her self. Every time she looked at me she spurted out new laughs that sounded like a sea lion calling to its mate.

"What?" I said.

Jerry took me aside while Jasmine continued to emit barks of laughter like shrill hiccoughs. "Alphonse, you don't understand how lucky you are. Any of us here would give our eyeteeth to work with Jean. I know she's a little quirky; but don't let the grandmother facade fool you. She's one of the best there is, and you'll learn a lot from her."

I was dumbfounded; but I couldn't respond because Jean chose that moment to toddle back from the head.

"My Goodness, I feel much better now."

This was more information than I wanted, so I changed the subject by offering to call the coffee servitor.

"Oh, my no! Dear. Why that'd just bring me right back here in ten or twenty minutes. No, I think we ought to go back up to our uncomfortable .6g lab and keep working on SMCRemote, don't you? We've got a lot of ground yet to cover today, and we certainly aren't getting it done down here, are we? ...What is Jasmine making that terrible sound for? Sounds like some animal dying. Jasmine, drink some water dear...

We took the turbo back out to the rim and walked back to the lab. The nearly doubled g slowed Jean down quite a bit. She was much spryer down in the lounge.

After we got situated back at our workstation, she said: "Now then Alphonse dear, I think we'd better see if we can execute that command we just built. What do you think?"

I had been wondering about how we were going to do this, so I agreed. "Er, yes."

"Good dear. Now, since the command we are about to execute invokes the SMC compiler, I think the first thing we need to do is create a simple source file for that compiler to read. So let's write a test case that creates this file, then invokes the compiler, and then checks to see that the compiler has created the proper output files."

She had pulled out her knitting again, and showed no sign that she wanted to touch the keyboard, so I took it and started to type:

```
public void testExecuteCommand() throws Exception {
    File sourceFile = new File("simpleSourceFile.sm");
    PrintWriter pw = new PrintWriter(new FileWriter(sourceFile));
    pw.println("
}
```

I didn't know how to continue, so I looked expectantly at her.

"That's fine dear. Here, I'll type in the SMC syntax for you."

```
public void testExecuteCommand() throws Exception {
    File sourceFile = new File("simpleSourceFile.sm");
    PrintWriter pw = new PrintWriter(new FileWriter(sourceFile));
    pw.println("Context C");
    pw.println("FSMName F");
    pw.println("Initial I");
    pw.println("{I{E I A}}");
    pw.close();
}
}
```

"What does that mean, Jean?"

"Well, dear, for our purposes it means that the compiler will create a file named F.java. That's all you really need to know about it for now. However, once you get back to your quarters it would behoove you to look up the SMC document and read it. I wrote it quite a few years ago dear, and I still think it's one of my better documents. It's called: 'Care and Feeding of the State Map Compiler'¹. Now then, let's see if we can execute that compile command."

I wasn't quite sure how to execute the command; but I'd learned from Jerry and Jasmine that it's often better just to write the calls that express my intention. So I continued to type:

```
public void testExecuteCommand() throws Exception {
    File sourceFile = new File("simpleSourceFile.sm");
    PrintWriter pw = new PrintWriter(new FileWriter(sourceFile));
    pw.println("Context C");
    pw.println("FSMName F");
    pw.println("Initial I");
}
```

¹ You can get this document from: <http://www.objectmentor.com/resources/downloads/bin/smcJava.zip>

```

pw.println("{I{E I A}}");
pw.close();

String command = SMCRemoteServer.buildCommandLine("simpleSourceFile.sm");
assertEquals(true, SMCRemoteServer.executeCommand(command));

File outputFile = new File("F.java");
assertTrue(outputFile.exists());
assertTrue(outputFile.delete());
assertTrue(sourceFile.delete());
}

```

"Very nicely done, Alphonse. I think that captures the test case quite admirably. We haven't written `executeCommand` yet, but we can certainly express how we want it to be called, can't we. Now, let's stub that function out and watch this test case fail. It's always fun to watch them fail, don't you think?"

So I clicked on `executeCommand` and selected "Create Method", and my IDE created just the method stub I wanted. And, sure enough, the test failed.

```

public class SMCRemoteServer {
    ...
    public static boolean executeCommand(String command) {
        return false;
    }
}

```

"All right dear, now let's make that test pass."

Jean was sounding tired again. I knew she'd want another break soon. It was almost lunchtime, so I was hoping she could hold out until then. I quickly searched the javadocs to find out how to execute a command. Jean saw what I was doing and said:

"It's in the `Runtime` class, dear. There's a method named `exec`."

Sure enough, it was right where she said. So I typed the code that I thought would work.

```

public static boolean executeCommand(String command) {
    Runtime rt = Runtime.getRuntime();
    try {
        rt.exec(command);
        return true;
    } catch (IOException e) {
        return false;
    }
}

```

But when I ran the test it failed to find the output file. I looked in the directory and, sure enough, `F.java` was not there.

"Why is this failing, Jean?"

She looked up from her knitting and said: "You didn't wait for the process to finish, dear. When you execute a command it creates a new process that runs concurrently with yours. You have to wait for it to complete before you exit from `executeCommand`."

I examined the JavaDoc for `Runtime.exec`, and found that it returned a `Process` object. I also found that you could wait for the `Process` object to complete, and that you could query it for its exit status. So I made the following changes.

```

public static boolean executeCommand(String command) {
    Runtime rt = Runtime.getRuntime();
    try {
        Process p = rt.exec(command);

```

```
        p.waitFor();  
        return p.exitValue() == 0;  
    } catch (Exception e) {  
        return false;  
    }  
}
```

This time the test passed.

"That wasn't that hard." I said.

"Of course not dear. All we are doing is running a command. Of course it'll get a bit more complicated when we have to capture the messages that the compiler usually prints on the console. We'll have to bind to the standard output, and standard error of the `Process`. But let's do that after lunch dear, I'm starting to get hungry, aren't you?"

I sighed and stood up. It still seemed to me that we were going slowly. Though in hindsight I see that in my first half day with Jean we had finished up the client, and gotten the server to run a compile. We hadn't spent any time debugging. Perhaps we were moving faster than I thought.

Jean put the knitterbot back in her bag and held out a sweater. "Hear, dear, try this on."

If nothing else, this "gig" with Jean was going to teach me a new kind of tolerance.

To be continued...

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

www.objectmentor.com/resources/articles/CraftsmanCode/Craftsman_19_Tolerance.zip

The Craftsman: 20

SMCRemote Part X

Backslide.

Robert C. Martin
28, Nov, 2003

*...Continued from last month. You can download last month's code from:
www.objectmentor.com/resources/articles/CraftsmanCode/Craftsman_19_Tolerance.zip*

I returned to the lab after lunch, but Jean was not there. There was neither note, nor email from her; and her knitting basket was nowhere to be seen. After waiting a few minutes, I decided to sit down and continue working on the `SMCRemoteServer`.

So far the server doesn't serve anything. It's just a couple of functions that build and execute the SMC command line. It seemed reasonable to me that the server code should open a socket and accept connections from the `SMCRemoteClient` that we wrote earlier.

I was pretty frustrated with our pace today. We'd worked the whole morning and only gotten two small functions, and their attendant unit tests written. I felt like making *progress*. So I grabbed the keyboard and began to type.

"First," I thought, "this server needs to serve something. So let's use the `SocketServer` class that Jerry and I built last week."

Getting the server to serve was pretty simple. I just needed to write a constructor that created a `SocketService` object and passed in a `SocketServer`. Then the `SocketServer.serve` method would automatically be called as soon as `SMCRemoteClient` tried to connect.

```
public SMCRemoteServer() throws Exception {
    service = new SocketService(9000, new SocketServer() {
        public void serve(Socket socket) {
            // SMCRemoteClient has connected.
        }
    });
}
```

I looked in the `SMCRemoteClient` code and saw that the client expects a string to be sent upon connection. That string should begin with "SMCR". So I wrote that string in the `serve()` method.

```
public void serve(Socket socket) {
    // SMCRemoteClient has connected.
    try {
        ObjectOutputStream os =
            new ObjectOutputStream(socket.getOutputStream());
        os.writeObject("SMCR");
    } catch (IOException e) {
```

```

    }
}

```

Next the server needs to read a `CompileFileTransaction` from the client. It needs so write the files contained in that transaction, invoke the compiler, and then return the resulting files in a `CompilerResultsTransaction`. This didn't seem like a hard thing to write so...

```

public void serve(Socket socket) {
    // SMCRemoteClient has connected.
    try {
        ObjectOutputStream os =
            new ObjectOutputStream(socket.getOutputStream());
        os.writeObject("SMCR");
        ObjectInputStream is =
            new ObjectInputStream(socket.getInputStream());
        CompileFileTransaction cft =
            (CompileFileTransaction)is.readObject();
        String filename = cft.getFilename();
        cft.sourceFile.write();
        String command = buildCommandLine(filename);
        executeCommand(command);
        //OK, what file do I put into the Result?
    } catch (Exception e) {
    }
}

```

Hmmm. Compiling the file didn't seem very difficult, but what output file should I put into the result transaction? What is its name? I remember from this morning that Jean had coached me in writing a test for invoking a compile. I looked at that test and found that the input file has a `.sm` suffix, and the output file as a `.java` suffix. So all I had to do is replace `".sm"` with `".java"` in the filename.

```

public void serve(Socket socket) {
    // SMCRemoteClient has connected.
    try {
        ObjectOutputStream os =
            new ObjectOutputStream(socket.getOutputStream());
        os.writeObject("SMCR");
        ObjectInputStream is =
            new ObjectInputStream(socket.getInputStream());
        CompileFileTransaction cft =
            (CompileFileTransaction)is.readObject();
        String filename = cft.getFilename();
        cft.sourceFile.write();
        String command = buildCommandLine(filename);
        executeCommand(command);
        //Figure out the file name.
        String compiledFile = filename.replaceAll("\\.sm", ".java");
        CompilerResultsTransaction crt =
            new CompilerResultsTransaction(compiledFile);
        os.writeObject(crt);
        socket.close();
    } catch (Exception e) {
    }
}

```

This looked like it should work. All I needed to do was to start up the server and run the client. That should be pretty simple. So I created a source file in my directory named `F.sm`, just like the one that Jean

had me create this morning.

```
Context C
FSMName F
Initial I
{I{E I A}}
```

And then I wrote a simple main function in `SMCRemoteServer`.

```
public static void main(String[] args) throws Exception {
    SMCRemoteServer server = new SMCRemoteServer();
}
```

And then I ran it. And it just hung there, waiting for a client to connect. Now *this* was exciting! I was getting something *done*! So next I ran the client with "`F.sm`" as its argument. And it *RAN*! Or rather it exited after several seconds with a normal exit code, and without any error messages either from the client or the server. Cool!

But what had it done? I couldn't immediately tell. So I checked the directory and found an `F.java` file sitting there! It had the right date on it, so it must have been created by my run of the client. Way cool! I opened the file and it looked like generated Java code. It even said it was generated by SMC. My code worked!

It had been a couple of weeks since I had felt this good; since before working with Jerry as a matter of fact. *This* was what programming was all about! I was filled with the rush of getting code working. I was *invincible*! I got up and did a little jig around my seat chanting "Oh yes! Oh yes! I'm a programmer. Oh yes!"

Jerry must have been nearby, because he came into the lab at that point. "Hay, Alphonse, what's are you celebrating?"

"Oh, hi Jerry! Look at this! I just got the `SMCRemoteServer` working!"

"Really? That's great Alphonse." The look on his face was funny -- as if he didn't believe me. So I showed him. I showed him how the server was sitting there running. I ran the client again. I showed him that an `F.java` file with the right date was created. I even showed him that it contained generated java code.

Jerry looked at me almost fearfully, and then nervously glanced at the door. "Alphonse, where are your tests?"

"Jerry, you don't need unit tests for something this simple. Look, it was just a dozen lines of code or so. Jerry, I made *progress* here. I got something *done*. And it didn't have to take all day! I think you guys waste too much time on all those unit tests!"

Jerry just stared at me for a few seconds, as if he couldn't quite process what I was telling him. Then he shut the door to the lab and sat down next to me.

"Alphonse, has Jean seen this?"

"No, she hasn't come back from lunch yet."

"Delete it, Alphonse."

"Delete what?"

"Delete the code you just wrote. It's worthless."

I had half expected this kind of reaction from someone. I drew my mouth into a sneer and said: "Oh, come on Jerry! It works! How could it be worthless?"

"Are you sure it works, Alphonse?"

"You saw it for yourself!"

"Are YOU sure it works?"

"Of course it works. The `F.java` file is proof!"

"Alphonse, why is the date of the `F.sm` file exactly the same as the date on the `F.java` file?"

"What?" I looked at the directory, and sure enough they were identical to the second. But that didn't make any sense. I had written the `F.sm` file by hand several minutes ago. Why did it have the same date as the `F.java` file that was created seconds ago when I demonstrated the system to Jerry? I stared at it for awhile, and then admitted that I didn't know.

"Delete the code, Alphonse. You don't understand it."

"Aw, come on Jerry, I understand it. I just can't figure out why the dates don't look right. It works Jerry...it works." But I wasn't as sure any more. *Why was that date different?*

"Alphonse, by any chance did you happen to run the client and server in the same directory?"

"Uh..." I hadn't specified a different directory for them so I guessed they must have been running in the same place. "...I guess so."

"So the client and the server were both reading and writing the same files in the same directory?"

"...oh..."

Jerry nodded grimly. "Yeah."

I shook my head. "OK, Jerry, you've got a point. I don't understand everything that's going on. But look, there's an `F.java` file there. *Something* had to be working!"

"So what? You don't know what is and what isn't working. You don't understand what you've done. The things that appear to be working may be working by accident. Is it possible, for example, that that `F.java` file was left over from a previous unit test?"

It *was* possible! Yikes, Jean and I had caused the system to write an `F.java` file this morning! "Damn! Yes, it's *possible* -- but it's not very likely!"

"Delete the code, Alphonse. It's crap."

I just stared at him. I had made *progress* damn it! Now he wanted me to delete all that progress. But he was right. I didn't understand this code. And I had no tests that would demonstrate, step by step, that everything was working as it should. If my code was working (and now I was really starting to wonder just how much of it was working) it was working more by accident than by design.

"Delete the code, Alphonse. You don't want Jean to see this. Right now she thinks the world of you, and this would be very disappointing for her."

My resolve finally failed. My shoulders fell, my head hung, and I reached over and deleted the code.

Jerry walked out of the room, shaking his head.

A few minutes later, Jean walked in. "Hello Alphonse dear. I'm sorry I'm late, but I got into a conversation with an old friend of mine and we started comparing pictures of our grandchildren. I love showing the pictures of my grandchildren. Have you seen them dear? Oh, never mind dear, we've got work to do you and I. What have you been up to while I was detained?"

"Nothing, Jean, I was just waiting for you."

To be continued...

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

www.objectmentor.com/resources/articles/CraftsmanCode/Craftsman_20_Backslide.zip

The Craftsman: 21

SMCRemote Part XI

Patchwork.

Robert C. Martin
14 December 2003

*...Continued from last month. You can download last month's code from:
www.objectmentor.com/resources/articles/CraftsmanCode/Craftsman_20_Backslide.zip*

"Well now, Alphonse Dear, where do you think we should run the compile?"

I was still reeling from having just deleted all that code I wrote, so I missed the question.

"I'm sorry Jean, what did you say?"

"Well, we can't just run the SMC compiler any old place, can we dear? No, we've got to run it in a new directory. We have to copy all the incoming files into that directory, run the compile, and then send all the resulting files back to the caller."

"How many resulting files are there?" I asked. "I thought that SMC just produced a .java file."

"The Java code generator does dear. But the C++ code generator creates a .h and a .cc file. Other generators might produce other kinds of files. I'm afraid that it's a bit unpredictable. Anyway, dear, we want the compiler to run in a clean environment, and that means an empty directory with just the input .sm file present."

I thought about this for a second, and realized that she was right. I was starting to be very glad that I had deleted that old code. "So we should have a special directory to run the compiles in?"

"Oh, more than that dear. I think we should create a working directory as soon as we receive a `CompileFileTransaction`. Then we should write the input file into that directory. It'll be the only file there, and so things will be nice and clean, don't you agree? And then we can run the compiler, delete the input file, and gather up any remaining files and put them into the `CompilerResultsTransaction` for shipment back to the client. Oh my, this sounds like such *fun*. Why don't you write a little unit test that makes sure we can create a new directory, dear?"

I took the keyboard and started to type. While I was typing, Jean pulled a stack of brightly colored 6" cloth squares from her bag and started to sew them together.

```
public void testMakeWorkingDirectory() throws Exception {
    File workingDirectory = SMCRemoteServer.makeWorkingDirectory();
    assertTrue(workingDirectory.exists());
    assertTrue(workingDirectory.isDirectory());
}
```

"Oh, that's fine dear. First you create the directory, and then you make sure that it exists, and that it is, in fact, a directory. Now make that fine test pass."

I know Jean doesn't mean to be condescending, but every word grates on my nerves. I squared my

shoulders and wrote the code to make the test pass. First I clicked on the missing function and had my IDE add it for me. Then I filled it in with the following code.

```
public static File makeWorkingDirectory() {
    File workingDirectory = new File("workingDirectory");
    workingDirectory.mkdir();
    return workingDirectory;
}
```

This code passed the test on the first try, of course, so I sat back and waited for Jean's next instruction.

"What are you waiting for dear?"

"I'm waiting for you to tell me what to do next."

"Are you? My goodness, I'm just a foolish, talkative old lady. Don't you have any ideas of your own? A bright young man such as yourself, you must be full of ideas. What do *you* think we should do next?"

This was even more condescending than before. Jasmine and Jerry seemed to think that it was some kind of honor working with Jean; but I was finding it pretty painful.

"We should probably delete the directory when we are done with it." I said grumpily.

"I think that's a splendid idea. We'll have to make sure that the entire directory is deleted along with any files inside it."

So I wrote the following unit test while Jean continued sewing squares together.

```
public void testDeleteWorkingDirectory() throws Exception {
    File workingDirectory = SMCRemoteServer.makeWorkingDirectory();
    File someFile = new File(workingDirectory, "someFile");
    someFile.createNewFile();
    assertTrue(someFile.exists());
    SMCRemoteServer.deleteWorkingDirectory(workingDirectory);
    assertFalse(someFile.exists());
    assertFalse(workingDirectory.exists());
}
```

Jean looked up from her sewing and said: "Very nice, Alphonse. That's just what we want."

So then I wrote the following code:

```
public static void deleteWorkingDirectory(File workingDirectory) {
    workingDirectory.delete();
}
```

But this didn't pass the test. The test complained that `someFile` still existed.

"I think you have to delete all the files in the directory before you can delete the directory itself, Alphonse. I've never understood why they made that rule. I think it would be better if they believed you when you said you wanted the directory deleted. Oh well, I guess you'll have to loop through all the files and delete them one by one, dear."

So I wrote the following:

```
public static void deleteWorkingDirectory(File workingDirectory) {
    File[] files = workingDirectory.listFiles();
    for (int i = 0; i < files.length; i++) {
        files[i].delete();
    }
    workingDirectory.delete();
}
```

This made the tests pass. However I didn't like it. "Jean, this works, but what if there are

subdirectories?"

"Yes, dear, we can't be sure that one of the SMC code generators won't create a subdirectory or two. I think you'd better augment the test to handle that case."

So I changed the test as follows:

```
public void testDeleteWorkingDirectory() throws Exception {
    File workingDirectory = SMCRemoteServer.makeWorkingDirectory();
    File someFile = new File(workingDirectory, "someFile");
    someFile.createNewFile();
    assertTrue(someFile.exists());

    File subdirectory = new File(workingDirectory, "subdirectory");
    subdirectory.mkdir();
    File subFile = new File(subdirectory, "subfile");
    subFile.createNewFile();
    assertTrue(subFile.exists());

    SMCRemoteServer.deleteWorkingDirectory(workingDirectory);
    assertFalse(someFile.exists());
    assertFalse(subFile.exists());
    assertFalse(subdirectory.exists());
    assertFalse(workingDirectory.exists());
}
```

Now the test failed as expected. The subFile was not getting deleted. So next I changed the code to make the test pass.

```
public static void deleteWorkingDirectory(File workingDirectory) {
    File[] files = workingDirectory.listFiles();
    for (int i = 0; i < files.length; i++) {
        if (files[i].isDirectory())
            deleteWorkingDirectory(files[i]);
        files[i].delete();
    }
    workingDirectory.delete();
}
```

"Oh that's just fine, dear, just fine. Very nicely done. Now, what next Alphonse?"

Something had been nagging at me, so I said: "Jean, this is a server isn't it?"

"Yes, dear. It waits on a socket for our client code to connect to it."

"How many clients might there be?"

"Oh, there could be dozens, perhaps more."

"Then if we get two clients connecting at the same time, we'll try to create the subdirectory twice, won't we?"

"Why yes, dear, I suppose that's true. That could cause quite a bit of trouble for us couldn't it. We'd have more than one session copying files into the directory, running compiles in the directory, copying files out of the directory, and deleting the directory too. That would be terrible wouldn't it? How do you think we should solve this Alphonse?"

"What if each working directory had a unique name? That way each of the incoming connections would have it's own directory to work in."

"That's just a superb idea, Alphonse. Why don't you write the test cases for that?"

I thought it was a superb idea too, so I wrote the following test.

```
public void testWorkingDirectoriesAreUnique() throws Exception {
    File wd1 = SMCRemoteServer.makeWorkingDirectory();
```

```

File wd2 = SMCRemoteServer.makeWorkingDirectory();
assertFalse(wd1.getName().equals(wd2.getName()));
}

```

This test failed as expected. To make it pass I needed some way to create unique names. "I suppose I could use the time of day to generate a unique name." I said more to myself than to Jean.

"Yes, why don't you try that dear?"

So I wrote the following code.

```

public static File makeWorkingDirectory() {
    File workingDirectory = new File(makeUniqueWorkingDirectoryName());
    workingDirectory.mkdir();
    return workingDirectory;
}

private static String makeUniqueWorkingDirectoryName() {
    return "workingDirectory" + System.currentTimeMillis();
}

```

The test passed, but I saw a funny look in Jean's face. So I hit the test button again. She nodded while the test passed again. I hit the test button several more times, and sure enough there was an occasional failure.

"Yeah," I said, "if the two calls to `makeUniqueWorkingDirectoryName` are too close together, then `currentTimeMillis` returns the same time and the names aren't unique."

"Oh dear." said Jean.

"I suppose I could just use a static integer and increment for each name, but then the names would start over each time we restarted the server. That might cause collisions too."

"Good thinking dear." said Jean.

"Why don't we use both techniques." and I stopped talking and started typing.

```

public class SMCRemoteServer {
    ...
    private static int workingDirectoryIndex = 0;
    ...
    private static String makeUniqueWorkingDirectoryName() {
        return "workingDirectory" +
            System.currentTimeMillis() + "_" +
            workingDirectoryIndex++;
    }
    ...
}

```

Now the tests worked repeatedly and I was quite satisfied with myself. I was certain Jean would want a break soon, and we had gotten something done. On the other hand, what we got done was something very different from the code that I deleted before Jean came in.

"Alphonse."

"Yes, Jean."

"Would you mind deleting all those left over working directories dear? They're causing quite a big of clutter in our development directory. I just hate clutter, don't you? I hate it in my code, I hate it in my room, and I hate it in my directories."

I looked at the directory and noticed that there were dozens and dozens of left over working directories. I hung my head as I realized that my tests had not been cleaning up after themselves. I sheepishly made the required changes.

```
public void testMakeWorkingDirectory() throws Exception {
    File workingDirectory = SMCRemoteServer.makeWorkingDirectory();
    assertTrue(workingDirectory.exists());
    assertTrue(workingDirectory.isDirectory());
    SMCRemoteServer.deleteWorkingDirectory(workingDirectory);
}

public void testWorkingDirectoriesAreUnique() throws Exception {
    File wd1 = SMCRemoteServer.makeWorkingDirectory();
    File wd2 = SMCRemoteServer.makeWorkingDirectory();
    assertFalse(wd1.getName().equals(wd2.getName()));
    SMCRemoteServer.deleteWorkingDirectory(wd1);
    SMCRemoteServer.deleteWorkingDirectory(wd2);
}
```

"That's much better dear. Thank you. Now, I think it's time for a break, don't you?"

"Sounds good to me."

Jean had sewn 16 squares together into a 4X4 grid.

To be continued...

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

www.objectmentor.com/resources/articles/CraftsmanCode/Craftsman_21_Patchwork.zip

The Craftsman: 22

SMCRemote Part XII

Bug Eye.

Robert C. Martin
8 January 2004

*...Continued from last month. You can download last month's code from:
w www.objectmentor.com/resources/articles/CraftsmanCode/Craftsman_21_Patchwork.zip*

Jean invited me join her in the journeyman's lounge, but I needed to clear my head. So I excused myself and went out to an observation bubble. The starbow was a brilliant stripe of color painted across the heavens. It was a concentric with our ship, and so always appeared below us whenever we looked down through the transparent floor of the bubble. The ship's rotation made the starbow look like a river of colored stars slowly streaming under the floor.

It was hard for me to believe that I had just met Jean for the first time this morning. Somehow it seemed like a much longer time. As I looked back on that time I realized that in the few hours we spent together we got quite a bit done; and yet through it all I had felt so *frustrated*. Somehow the way she worked made me feel like were moving so slowly. Jean seemed to think that moving slowly was a good thing. She had given me more than one lecture about taking the time and care to build the best software we can. I couldn't disagree with what she was saying, and I was frankly impressed by how much we had really gotten done today, and yet it still felt too slow.

I guess, during my classes in school, I had gotten used to writing code quickly and spending lots of time debugging. Somehow the debugging didn't seem slow to me. Somehow it felt like I was going fast. But when I worked with Jerry, Jasmine, and Jean we wrote our code much more slowly. And we wrote all those tests that seemed to take such a long time. We hadn't spent *any* time debugging so far; and yet somehow that didn't make it *feel* faster. It probably *was* faster, but it felt ponderous.

As I took the turbo back up to our lab, I resolved to stop feeling frustrated. Our pace was good, our quality was high, and my feelings of frustration were just old baggage that I needed to get rid of.

When I got to the lab a fellow about my age was sitting in my seat. There was no sign of Jean.

"Hello." I said. "Can I help you?"

"Uh... hi...uh, I'm Avery. Jean said I should work with you for the rest of the day."

"Oh. She did? Uh..."

"Yeah, she said you were supposed to show me how to write unit tests."

"Oh...uh...Really? Don't you know?"

"Yeah, I know...uh...Jason fired me."

"Jason was your Journeyman? He *fired* you?"

"Yeah...uh...Not really, he just asked Jean if he could stop working with me, so Jean told me to work with you for today."

"Why did Jason *do* that?"

“I worked through a break while Jason was gone, and wrote a whole bunch of code without any tests. When he came back he got all upset and told me to delete it. I got mad back at him and told him I wouldn’t delete it. So he just left.”

I felt a bit sheepish. Did everybody go through something like this? “He just left?” I asked.

“Yeah, he got all bug-eyed and red in the face, and then just walked out of our lab. Next thing I know Jean is telling me to work with you while she finds me a new journeyman. She said you’d understand.”

Now I felt even more sheepish. “Uh, yeah, I guess I do. When did you start your apprenticeship?”

“Last week, same as you. I worked with Jimmy, and Joseph last week, and with Jason today. I got along OK with the other two, but there was just something about Jason that set me off. I guess I set him off too.”

“I guess that happens sometimes. Shall we get to work?”

“Why not?”

I told him about the `SMCRemote` project. I explained what Jean and I had been up to for the last few hours. I told him about the client software that we had gotten working this morning and the server software that Jean and I had been putting together since then. When I finished he said: “It sounds like you are about ready to have the server run a compile.”

“Yes, I think we are. Tell you what, if I write the test, will you make it pass?”

“Sure, why not. I guess I’m going to have to get used to this testing stuff.”

“Yes, I think you are. OK, so here’s the test I’m thinking of.”

```
public void testCompileIsRunInEmptyDirectory() throws Exception {
    File dummy = new File("dummyFile");
    dummy.createNewFile();
    CompileFileTransaction cft = new CompileFileTransaction("dummyFile");
    dummy.delete();

    CompilerResultsTransaction crt = SMCRemoteServer.compile(cft, "ls >files");

    File files = new File("files");
    assertFalse(files.exists());

    crt.write();
    assertTrue(files.exists());

    BufferedReader reader = new BufferedReader(new FileReader(files));
    HashSet lines = new HashSet();
    String line;
    while ((line = reader.readLine()) != null) {
        lines.add(line);
    }
    reader.close();
    files.delete();

    assertEquals(2, lines.size());
    assertTrue(lines.contains("files"));
    assertTrue(lines.contains("dummyFile"));
}
```

“Yikes!” said Avery. “OK, let me see if I understand this. You create a empty dummy file and load it into a `CompileFileTransaction`. Then you ‘compile’ it but use a ‘`ls >files`’ command instead of the normal `compile` command. You expect the `compile` function to return a `CompilerResultsTransaction`. You expect that transaction to have the file that contains the output of the ‘`ls`’ command. You read that file and make sure that the ‘`ls`’ command saw nothing but the `dummyFile` and the `files` file. I guess that proves that the `compile` was run in a directory whose sole contents was the `dummyFile`.”

This Avery character was no fool. “Yes, that’s how I see the test working. Can you make it pass?”

“Jimmy told me you should always make the test fail before you try to make it pass. He said to do the easiest thing that would make the test fail. So, I guess something like this.”

```
public static CompilerResultsTransaction
compile(CompileFileTransaction cft, String command)
{
    return null;
}
```

Avery ran the tests, and they failed. “OK, this fails because it returns a null pointer. We can fix that like so.

```
public static CompilerResultsTransaction
compile(CompileFileTransaction cft, String command) throws Exception
{
    return new CompilerResultsTransaction("");
}
```

“OK, that fails because there’s no filename for the CompilerResultsTransaction. So we’re going to need a file. We get that file by executing the compiler.

```
public static CompilerResultsTransaction
compile(CompileFileTransaction cft, String command) throws Exception
{
    executeCommand(command);
    return new CompilerResultsTransaction("files");
}
```

“Hmm, this fails for the same reason. The files file doesn’t exist. But how could that be.”

After some research we found that the Runtime.exec() function does not interpret the > as a file redirection command. Fixing that was a matter of changing the test as follows:

```
CompilerResultsTransaction crt =
    SMCRemoteServer.compile(cft, "sh -c \"ls >files\"");
```

“OK, now the test fails at the assertFalse(files.exists()) line. This is because we are executing the ls command in the current directory. We need to create a subdirectory and execute it there.”

```
public static CompilerResultsTransaction
compile(CompileFileTransaction cft, String command) throws Exception
{
    File wd = makeWorkingDirectory();
    //How do I execute the command in the working directory?
    executeCommand(command);
    return new CompilerResultsTransaction("files");
}
```

“Alphonse, I can create the working directory with that function that you and Jean wrote. But how do I get the command to execute in that directory. There doesn’t seem to be any way to tell Runtime.exec() what directory to run in.”

Avery and I searched the documents, but we couldn’t find any obvious way to solve this. Then I had an idea.

“Avery, why don’t we prefix the command with a “cd workingDirectory;”.

“Hmmm. That might work but it means that we put the sh -c stuff in the wrong place. We should have put it in SMCRemoteServer.executeCommand. I’ll bet that the Runtime.exec() function can’t

deal with multiple commands and semicolons.”

“You’re probably right. Even if you aren’t, the `executeCommand` function is still a better place for the `sh -c` stuff. Let’s move it.

```
public static boolean executeCommand(String command) {
    Runtime rt = Runtime.getRuntime();
    try {
        Process p = rt.exec("sh -c \"" + command + "\"");
        p.waitFor();
        return p.exitValue() == 0;
    }
    catch (Exception e) {
        return false;
    }
}
```

“OK, now let’s do that `cd` thing.

```
public static CompilerResultsTransaction
compile(CompileFileTransaction cft, String command) throws Exception
{
    File workingDirectory = makeWorkingDirectory();
    String wd = workingDirectory.getName();
    executeCommand("cd " + wd + ";" + command);
    return new CompilerResultsTransaction("files");
}
```

“And now, once again, it fails at the new `CompilerResultsTransaction("files")` line. And it’s obvious why! That function is not executing in the subdirectory. We need to add the subdirectory path to the file name.”

```
return new CompilerResultsTransaction(wd + "/files");
```

“No, that doesn’t work either. Now it fails because the `CompilerResultsTransaction` thinks the name includes the subdirectory path, and the write function is trying to write the file back into the subdirectory.”

I sighed. “This is complicated.”

“No,” said Avery, “we just have to pass the subdirectory to the `CompilerResultsTransaction` constructor so that it can load the file without storing the path in the filename.

```
public static CompilerResultsTransaction
compile(CompileFileTransaction cft, String command) throws Exception
{
    File workingDirectory = makeWorkingDirectory();
    String wd = workingDirectory.getName();
    executeCommand("cd " + wd + ";" + command);
    return new CompilerResultsTransaction(workingDirectory, "files");
}
```

```
public class CompilerResultsTransaction implements Serializable {
    private FileCarrier resultFile;

    public CompilerResultsTransaction(File subdirectory,
                                     String filename) throws Exception {
        resultFile = new FileCarrier(subdirectory, filename);
    }

    public void write() throws Exception {
```

```

        resultFile.write();
    }
}

public class FileCarrier implements Serializable {
    private String fileName;
    private LinkedList lines = new LinkedList();
    private File subdirectory;

    public FileCarrier(String fileName) throws Exception {
        this(null, fileName);
    }

    public FileCarrier(File subdirectory, String fileName) throws Exception {
        this.fileName = fileName;
        this.subdirectory = subdirectory;
        loadLines();
    }

    private void loadLines() throws IOException {
        BufferedReader br = makeBufferedReader();
        String line;
        while ((line = br.readLine()) != null)
            lines.add(line);
        br.close();
    }

    private BufferedReader makeBufferedReader()
        throws FileNotFoundException {
        return new BufferedReader(
            new InputStreamReader(
                new FileInputStream(new File(subdirectory, fileName))));
    }

    public void write() throws Exception {
        PrintStream ps = makePrintStream();
        for (Iterator i = lines.iterator(); i.hasNext(); )
            ps.println((String)i.next());
        ps.close();
    }

    private PrintStream makePrintStream() throws FileNotFoundException {
        return new PrintStream(
            new FileOutputStream(fileName));
    }

    public String getFileName() {
        return fileName;
    }
}

```

“Whoah! Now it’s getting pretty far. It’s failing at the `assertEquals(2, lines.size())` line of the test. And again it’s pretty clear why. We never wrote the input file. That should be easy to fix!”

“Not that easy. We’ll have to make sure we write it into the subdirectory.”

“Ah, you’re right! We have to add the `workingDirectory` to the `FileCarrier.write` function. Good catch!”

```

public static CompilerResultsTransaction
compile(CompileFileTransaction cft, String command) throws Exception
{
    File workingDirectory = makeWorkingDirectory();

```

```
String wd = workingDirectory.getName();
cft.sourceFile.write(workingDirectory);
executeCommand("cd " + wd + ";" + command);
return new CompilerResultsTransaction(workingDirectory, "files");
}
}

public void write() throws Exception {
    write(null);
}

public void write(File subdirectory) throws Exception {
    PrintStream ps = makePrintStream(subdirectory);
    for (Iterator i = lines.iterator(); i.hasNext();)
        ps.println((String)i.next());
    ps.close();
}

private PrintStream
makePrintStream(File subdirectory) throws FileNotFoundException {
    return new PrintStream(
        new FileOutputStream(new File(subdirectory, fileName)));
}
```

“Bang! And that’s it! Now it passes.”

“Sweet!”

To be continued...

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

w www.objectmentor.com/resources/articles/CraftsmanCode/Craftsman_22_BugEye.zip

The Craftsman: 23

SMCRemote Part XIII

Raggedy.

Robert C. Martin
19 February 2004

*...Continued from last month. You can download last month's code from:
www.objectmentor.com/resources/articles/CraftsmanCode/Craftsman_22_BugEye.zip*

20 Feb 2002, 14:00

Avery looked at me with a smug smile on his pimply face. His bushy red eyebrows added intensity to the look. He had gained confidence over the last hour, and his stammering had disappeared. “OK, so now the server executes compiles in an empty directory. Now I think its time to test the whole server from end to end.”

“You mean you want to open a socket with the server, send in a `CompileFileTransaction` and see if the appropriate `CompilerResponseTransaction` comes out?”

“Precisely, Alphonse. Precisely.” As he gained confidence he started speaking with an odd formality. It was a bit strange, but also a bit fun. Playing along I said: “I see, Avery. Then I suggest that you compose the appropriate test case, and then I’ll endeavor to satisfy it.”

“Excellent suggestion, Alphonse. Shall I proceed?”

“Please do.”

Avery faced the screen, making a show of stretching his fingers and squaring his shoulders before touching the keyboard. He typed:

```
public void testServerEndToEnd() throws Exception {  
  
}
```

He paused with mock intensity and said: “Do you agree with the name?”

“Indeed!” I replied. I was getting into this. He continued to type while he talked.

“Our first goal should be to instantiate the `SMCRemoteServer` and bind it to some port number that is convenient for our test case.”

```
public void testServerEndToEnd() throws Exception {  
    SMCRemoteServer server = new SMCRemoteServer(999);  
}
```

“I appreciate the intent of that code, but notice that it fails to compile. Apparently there is no constructor that takes an `int`.”

“Well spotted, Alphonse. Well spotted. We must therefore create just such a constructor.”

```
public class SMCRemoteServer {  
    ...  
    public SMCRemoteServer(int port) {  
    }  
    ...  
}
```

“Nicely done, Avery. I suggest that this test will pass in its current form.”

“I believe you may be correct, Alphonse. Shall we try?”

“Yes, I believe we should.”

Avery pushed the test button and the green bar flashed on the screen indicating that the test had passed.

“As expected.” Avery said with a self-satisfied tone. “Next we’ll have our test connect to that socket like so:”

```
public void testServerEndToEnd() throws Exception {  
    SMCRemoteServer server = new SMCRemoteServer(999);  
    Socket client = new Socket("localhost", 999);  
}
```

“I expect, Avery, that this will fail.”

“I agree, Alphonse. We never created the server socket.” Pushing the test button quickly yielded a red bar on the screen. We looked at each other and nodded, pleased with our mutual prescience.

I held my hands out in front of me, as a polite gesture requesting Avery to slide the keyboard in my direction. He did so with a flourish and a knowing look.

“Thank you Avery. To make this test pass we’ll use a utility named SocketService that Jerry and I wrote last week.”

```
public SMCRemoteServer(int port) throws Exception {  
    SocketService service = new SocketService(port, new SocketServer() {  
        public void serve(Socket theSocket) {  
            try {  
                theSocket.close();  
            } catch (IOException e) {  
            }  
        }  
    });  
}
```

“Nicely done, Alphonse. Shall I run the test?”

“Certainly!” I slid the keyboard back to him.

Again we smiled and nodded – making a show of it -- as the green bar flashed on the screen.

“Now then,” harrumphed Avery, “The server should respond with a connection message.”

“Yes indeed it should, Avery.” I said, remembering back to last Thursday. “Jerry and I decided that the message would be a string that begins with SMCR.”

“Well then, that’s easy enough to test.”

```
public void testServerEndToEnd() throws Exception {  
    SMCRemoteServer server = new SMCRemoteServer(999);  
    Socket client = new Socket("localhost", 999);  
    ObjectInputStream is = new ObjectInputStream(client.getInputStream());  
    String header = (String) is.readObject();  
    assertTrue(header.startsWith("SMCR"));  
}
```

```
}
```

“Yes, that looks quite proper. Quite proper.” I said accepting the keyboard as Avery brandished it in my direction. I ran the test, and noted with satisfaction as it failed due to an `EOFException`. Now then, to make this pass we merely create the appropriate string and send it out the socket– like so!”

```
public SMCRremoteServer(int port) throws Exception {
    SocketService service = new SocketService(port, new SocketServer() {
        public void serve(Socket theSocket) {
            try {
                ObjectOutputStream os =
                    new ObjectOutputStream(theSocket.getOutputStream());
                os.writeObject("SMCR");
                theSocket.close();
            } catch (IOException e) {
            }
        }
    });
}
```

The test produced a green bar, and we repeated the smile and nod ritual. I passed the keyboard back to Avery holding it as though I were passing a fencing foil to my opponent at the start of a match. “Your weapon sir.”

“At your disposal, sir.” He responded. “I believe we must now prepare to send the `CompileFileTransaction`. This will require us to first write a source file containing code that the SMC compiler can compile.”

“Ah, indeed, Jean and I created such a file just this morning. The code is in the `testExecuteCommand()` function. We should be able to extract it into a function of its own named `writeSourceFile()`.”

```
private File writeSourceFile(String theSourceFileName) throws IOException {
    File sourceFile = new File(theSourceFileName);
    PrintWriter pw = new PrintWriter(new FileWriter(sourceFile));
    pw.println("Context C");
    pw.println("FSMName F");
    pw.println("Initial I");
    pw.println("{I{E I A}}");
    pw.close();
    return sourceFile;
}
```

“Excellent! Thank you Alphonse. So now I can create a `CompileFileTransaction`, send it to the server, and expect a `CompilerResultsTransaction` back – right?”

“I’d say that was correct, Avery.”

```
public void testServerEndToEnd() throws Exception {
    SMCRremoteServer server = new SMCRremoteServer(999);
    Socket client = new Socket("localhost", 999);
    ObjectInputStream is = new ObjectInputStream(client.getInputStream());
    ObjectOutputStream os = new ObjectOutputStream(client.getOutputStream());
    String header = (String) is.readObject();
    assertTrue(header.startsWith("SMCR"));
    File sourceFile = writeSourceFile("mySourceFile.sm");
    CompileFileTransaction cft = new CompileFileTransaction("mySourceFile.sm");
    os.writeObject(cft);
    os.flush();
    CompilerResultsTransaction crt =
```

```

        (CompileResultsTransaction) is.readObject();
        assertNotNull(crt);
    }

```

With the red bar from the failing test on the screen, Avery said. “Alphonse, do you agree that this code expresses my intent.”

“Yes, I think it expresses it very well. And now if you’ll hand me the keyboard, I’ll make it pass.”

```

public void serve(Socket theSocket) {
    try {
        ObjectOutputStream os =
            new ObjectOutputStream(theSocket.getOutputStream());
        ObjectInputStream is =
            new ObjectInputStream(theSocket.getInputStream());
        os.writeObject("SMCR");
        os.flush();
        CompileFileTransaction cft =
            (CompileFileTransaction) is.readObject();
        CompilerResultsTransaction crt = new CompilerResultsTransaction();
        os.writeObject(crt);
        os.flush();
        theSocket.close();
    } catch (Exception e) {
    }
}

```

“And sure enough, pass it does.”

“Well done Alphonse!”

“Thank you Avery. There really isn’t anything to it.”

“In that case, I’ll give you one more challenge.” He took the keyboard and began to type again.

```

public void testServerEndToEnd() throws Exception {
    SMCRremoteServer server = new SMCRremoteServer(999);
    Socket client = new Socket("localhost", 999);
    ObjectInputStream is = new ObjectInputStream(client.getInputStream());
    ObjectOutputStream os = new ObjectOutputStream(client.getOutputStream());
    String header = (String) is.readObject();
    assertTrue(header.startsWith("SMCR"));
    File sourceFile = writeSourceFile("mySourceFile.sm");
    CompileFileTransaction cft = new CompileFileTransaction("mySourceFile.sm");
    os.writeObject(cft);
    os.flush();
    CompilerResultsTransaction crt =
        (CompilerResultsTransaction) is.readObject();
    assertNotNull(crt);
    File resultFile = new File("F.java");
    assertFalse(resultFile.exists());
    crt.write();
    assertTrue(resultFile.exists());
}

```

“Aha!” I said. “You actually want me to invoke the compiler, eh?”

“That I do, my dear Alphonse. That I do.”

“Then prepare yourself for compilation, young Avery.”

```

public void serve(Socket theSocket) {
    try {
        ObjectOutputStream os =
            new ObjectOutputStream(theSocket.getOutputStream());

```

```

        ObjectInputStream is =
            new ObjectInputStream(theSocket.getInputStream());
        os.writeObject("SMCR");
        os.flush();
        CompileFileTransaction cft =
            (CompileFileTransaction) is.readObject();
        String command = buildCommandLine(cft.getFilename());
        CompilerResultsTransaction crt = compile(cft, command);
        os.writeObject(crt);
        os.flush();
        theSocket.close();
    } catch (Exception e) {
    }
}

```

We both leaned over the monitor as I pushed the test button.

Green Bar. Smile. Nod.

“We’ve done well, Avery. But I think it’s time we cleaned this code up a bit.”

“I quite agree, Alphonse, it has gotten a little, shall we say, raggedy?”

Together we worked through the new server code and split it up into a batch of smaller methods that worked together to tell the story. When we were done it looked like this; and all the tests still passed.

```

public SMCRremoteServer(int port) throws Exception {
    SocketService service =
        new SocketService(port, new SMCRremoteServerThread());
}

private static class SMCRremoteServerThread implements SocketServer {
    private ObjectOutputStream os;
    private ObjectInputStream is;
    private CompileFileTransaction cft;
    private CompilerResultsTransaction crt;

    public void serve(Socket theSocket) {
        try {
            initializeStreams(theSocket);
            sayHello();
            readTransaction();
            doCompile();
            writeResponse();
            theSocket.close();
        } catch (Exception e) {
        }
    }

    private void readTransaction() throws IOException, ClassNotFoundException {
        cft = (CompileFileTransaction) is.readObject();
    }

    private void initializeStreams(Socket theSocket) throws IOException {
        os = new ObjectOutputStream(theSocket.getOutputStream());
        is = new ObjectInputStream(theSocket.getInputStream());
    }

    private void writeResponse() throws IOException {
        os.writeObject(crt);
        os.flush();
    }

    private void sayHello() throws IOException {

```

```
        os.writeObject("SMCR");
        os.flush();
    }

    private void doCompile() throws Exception {
        String command = buildCommandLine(cft.getFilename());
        crt = compile(cft, command);
    }
}
```

“OK, that’s much better. I think it’s time for a break, don’t you?” I said to Avery.

“Yeah, I guess we’ve been at it for a couple of hours. Let’s go to the game room and play LandCraft.”

To be continued...

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

www.objectmentor.com/resources/articles/CraftsmanCode/Craftsman_23_Raggedy.zip

The Craftsman: 24 Dosage Tracking I OH NO!

Robert C. Martin
30 March 2004

*...Continued from last month. You can download last month's code from:
www.objectmentor.com/resources/articles/CraftsmanCode/Craftsman_23_Raggedy.zip*

20 Feb 2002,18:00

Whew! What an afternoon! I just got out of the strangest meeting I've ever been in. I had to write this down before dinner or else I'd forget the details. It started as Avery and I were on our way to the rim to play LandCraft in the game room. We had just gotten SMCRemote to do its first remote compile, and we wanted to celebrate. We didn't get very far, though, because Jean pinged us just as we were leaving the lab.

"Hello my dears!" she said to both of us though our lapel coms. I hope it's not too much of a bother, but I wonder if you could join me up in the .36g lounge in alpha shaft. I've got something new for you two to work on, and...well...why don't you just come up here and I'll explain everything." Avery and I looked at each other, grimaced, shrugged, and headed for the alpha shaft turbo.

Jean must have pre-coded the doors, because they opened for us before we could knock. When we stepped in we saw Jasmine, Jerry, Jean, and two people I'd never met before. One was a tall middle-aged woman with striking red hair, and the other was a thin, bespectacled, and pleasantly smiling fellow just a few years older than I. Jasmine was standing and gathering her belongings. She appeared to be getting ready to leave.

Avery took one look at Jasmine, and his buggy eyes got even buggier. Apparently he'd never seen her before. It was good to see someone else's first reaction to her— it put things in perspective.

Jean beamed at us and said: "You boys made it down here in good time! I'm always so impressed by the vigor of young boys, aren't you Jasmine dear? Oh, but there I go gabbing away again and distracting everyone from the point. I just don't know why I do that."

She seemed to gather herself, and in that fraction of a second I could see a determination on her face that I'd somehow missed before.

"Alphonse, Avery, this is Carole and Jasper. Jasper is going to be working with you and Jerry on a new project. Carole is the sponsor for this new project. She'll be working very closely with you too, helping to define the requirements -- oh, you call them stories nowadays don't you. Goodness, I'm sometimes so forgetful. You'll be starting on this project first thing in the morning. Carole, why don't you summarize dear."

"Excuse me, Jean." I blurted. "But what about our SMCRemote project? We just got the very first compile to run and we -- er, I -- was looking forward to getting back to it." I looked at Avery for support, but didn't get it. He still had his eyes locked on Jasmine.

“Don’t you worry about that, Aphonse dear. Jasmine will be taking that project over from you. She’ll be working with Adelaide. SMC remote is a wonderful learning project for new apprentices, but you boys have bigger fish to fry right now.”

“That’s right Hot Shot” Jasmine’s eyes nailed me – again. “Don’t you worry about SMCRemote. I’m going to go gather up Adelaide now, and she and I will pick up the pieces. Good work on getting that compile running.” She looked around at everyone in the room, tossing her midnight hair with each buoyant turn of her head. “See you all later.” She said, and then strode purposefully out of the lounge.

I could feel the atmosphere in the room change in her wake. Avery’s gaze followed her until she was gone. An odd smile grew on Jean’s face, and then vanished. She turned to Carole and said: Carole...

Carole stood and gestured for us to sit down. Jerry and Jasper were already seated. There were two chairs next to them arranged so that all four of us were facing Carole. As soon as we were seated she began. From her very first utterance you could feel ambition and energy exuding from her.

“Avery and Alphonse, I’m glad to meet you.” Her voice was loud, and her tone direct. “Jean has been telling me some very impressive things about you. Jasper and Jerry, it will be a pleasure to work with you again. Last time we had a lot of fun, didn’t we?”

Jerry fidgeted uneasily, but they nodded congenially at each other, and I could feel a kind of anticipation building.

“So, let me tell you what’s been going on.”

Jerry and Jasper fixed their full attention on her. Avery and I glanced at each other and followed suite.

“Three days ago the Hazard Avoidance guys detected a fluctuation in the starlight along our vector. The stars directly in front of us appear to be dimming and reddening. Spectro-analysis shows significant H₂ absorption lines in the dimmed stars. Apparently we are approaching a cloud of molecular hydrogen.”

She let that sink in for a second. It didn’t mean much to me, but Avery seemed to have recovered himself, and was very focused on her words.

“This dimmed patch of stars is growing perceptibly, meaning that it is close. The astronomy team says we’re likely to enter this cloud in about two months.”

Avery had his gaze fixed on Carole, and was holding his chin with his right hand. He moved that hand, pointing it at Carole and said:

“That shouldn’t be a problem for us. Our Ice Shield will protect us from it. Molecular clouds aren’t usually very dense after all.”

Carole gave Avery an appraising glance and said: “That’s true Avery. Those five cubic miles of ice running ahead of us will certainly deflect most of the H₂ particles away from the ship. Oh there will be some erosion, but astronomy doesn’t think this cloud is deep or dense enough to make much of a dent in our shield. However, there does appear to be a risk of diffraction.”

Avery nodded, trying to look knowledgeable, but it was clear that he was puzzled. Carole continued:

“From the point of view of the shield the cloud will appear to be a beam of neutral H₂ molecules all moving directly towards us at nearly light speed. The shield is a disk, and the beam will be diffracted around the edges of the disk. Some of those diffracted H₂ molecules will strike us. This won’t cause a problem for the ship, or anyone on board. The skin of the ship is designed to absorb such particles. However, it does pose a slight risk for the folks doing maintenance outside.”

“You mean radiation risk?” I said?

Jerry’s head jerked in my direction, an unpleasant awareness beginning to dawn on his face.

“Precisely!” Carole beamed. Avery looked annoyed, perhaps jealous that I had pleased her. He said:

“So what do you need us for?”

Jerry’s fists clenched.

“We need a new dosage tracking system.” She said.

Jerry groaned. “Oh No, not again!”

Carole’s smile nearly split her face in half. “Ah, I see you remember our last adventure with Dosage Tracking.”

Jerry looked miserable. He hung his head and said: “I hoped I’d never see it again. Last time it was

the cleanup from a fuel spill, now it's diffracting molecular hydrogen. God, when will this application die?"

Jean walked over to Jerry and put her hand on his shoulder. "There, there, Jerry dear. We aren't going to make you work on that old abomination you wrote when *you* were an apprentice. This time you get a team, and the opportunity to do it *right*."

I wasn't sure what *right* meant, but by the sternly emphatic way she said it, it must mean something pretty significant. Jerry just sat there shaking his head breathing deeply. Carole looked almost triumphant. Jean looked kindly determined. The rest of us had no idea what was going on.

Carole continued. "First thing in the morning – 0800 -- we'll meet in your lab up on 44. At that time we'll go over the stories for the system, and plan the first iterations. We'll also sketch out some of the acceptance tests we'll be using – We *will* write those acceptance tests this time, won't we Jerry?"

Jerry looked more miserable than ever. He nodded while staring at the floor, while Jean smiled benignly next to him.

The meeting broke up pretty quickly after that. Avery and I were the last to leave the lounge.

"Do you have any idea what a Dosage Tracking system is?" I asked Avery.

"Of course I do." Avery said with a tone of annoyed superiority.

"What is it?" I asked.

Avery looked at me with almost a sneer on his face. "Go look it up, Hot Shot." And he strode off without a glance back.

Clearly something is bugging him, but I can't figure out what it could be.

To be continued...

The Craftsman: 25 Dosage Tracking II Register Suit

Robert C. Martin
28 April 2004

...Continued from last month.

21 Feb 2002, 0800

The impactor that set us on our journey was 22km in diameter, and moving at 53km/sec when it slammed into the Pacific. We had launched two months earlier and were in a parking orbit 60° ahead of the Earth, waiting for the inevitable. We knew that the impact would kick up a lot of debris so we didn't want to be in near-Earth space. I guess they figured 1AU was safe.

I've seen recordings of the impact. I don't want to discuss them. Communication with parts of the Earth continued for a few weeks, but steadily diminished and then ceased. I guess it wasn't much fun down there.

That was 43 subjective years ago, in 1959. Since then we've been star-hopping; looking for a suitable home. The first ten systems we've visited haven't been very promising. There are plenty of planets, but other than the ammonia breathing dribins of a Centauri 5, we haven't seen anything even close to a biosphere.

Now we're about to plow through a cloud of molecular hydrogen at nearly C, and to protect our outside maintenance engineers we need to rewrite an old Dosage Tracking system that Jerry and Jasper were involved with some years back.

I walked into the lab on 44 just before 0800. Carole and Jerry were already there. Jerry looked resigned and was chatting with Carole about something that he clearly found embarrassing. Their conversation broke up before I could get close enough to join. A few minutes later, Jasper, Avery, and Jean walked in together. Avery gave me a nod and a smile as though nothing were wrong. Perhaps nothing was.

I wanted to talk to him but Carole convened the meeting before I could reach him. We all started moving towards the conference table at the end of the lab. I grabbed a chair, and Avery took the one next to me. He gave me a conspiratorial nod, just as Carole started talking.

"Jean and I have written up some initial stories for the new Dosage Tracking System. By the way, we're calling it 'DTrack'. I'll walk through the stories with you, and you can ask all the questions you like. The first story is 'Register Suit'."

Carol placed an index card on the table with the words "Register Suit" written on it.

"Our system tracks the radiation dosage received by outside maintenance workers by integrating the dosages received by the space suits that they wear. Each suit has a dosimeter integrated into its systems. When a suit is checked out for use, its dosimeter is read before the suit is released to the worker. When the suit is checked back in the dosimeter is read again. The difference is added to the total dose for that worker."

So the first thing we need is an inventory of suits. A suit is introduced to the system with this story.”

Jerry spoke up. “I supposed we’re using the bar code patches that are sewn onto the suit to identify them?”

“That’s right, Jerry dear.” Said Jean. “As I’m sure you remember the bar code contains a six character alphanumeric string that uniquely identifies each suit.”

Jerry grabbed the card and wrote ‘Bar Code Patch: X(6) on it.

Jasper poked Jerry in the ribs and said: “Stop writing COBOL on the cards, Jerry.”

“I can’t help it.” He responded. “It’s just the way I think.”

I asked: “So when a new suit is produced, it is given a new bar code patch and then registered with the system? How does that registration take place?”

Jerry said: “Yeah, that’s right Alphonse. The new suit is tagged and then entered into the system using a bar code reader.”

Carole added: “The new suit is carried to the Outside Maintenance suit-up room by hand. The clerk there selects the “Register New Suit” function on the screen and then scans the bar code. This registers the new suit with the DTrack system, and sends a message back to manufacturing telling them that the new suit is accounted for.”

Jasper grabbed the card and wrote: ‘Register New Suit function: on screen. Send confirmation to mfg.’

“I’ve never understood why we don’t create a suit clearing house.” Jerry said. “It doesn’t make any sense for maintenance to talk to manufacturing like that.”

“Later, Jerry, later.” Said Carole. “I know how you feel about this issue, and I agree. But the H₂ cloud is two months away and we *have* to be ready for it. Otherwise you’ll be spending your shifts patching up the database corruption in that old COBOL system of yours, and I will too— and that’s not the way *I* want to spend the next six to eight months.”

Jerry grimaced, but nodded acceptance.

“Is that how big the cloud is; eight light-months?” asked Avery.

“That’s astronomy’s best guess as of last night.” Replied Carole.

“Why does manufacturing need to know that the suit has been registered with DTrack?” I asked.

Jean replied: “Aphonse, dear, we track every suit from manufacturing through usage to decommissioning. When one department releases a suit another department registers it and the two departments exchange a message so that each knows what happened to the suit. That way we always know where the suits are. Can you imagine how awful it would be for the poor folks who have to use the suits if they didn’t know the history of the suit, how old it was, what repairs have been done to it, what radiation exposure it has taken? Oh, I just don’t want to think about it.”

“What if production doesn’t know about the suit that’s being registered?” Avery asked.

“Production will send a denial message, and DTrack will not accept the registration.” Carole replied.

I grabbed the card and wrote “Reject registration on denial” on it. Nobody seemed to mind.

“Will production send an acceptance message if they recognize the suit?”

“Yes dear.” Jean responded.

Avery blurted: “What if there is no response from production?”

“Wait 10 seconds and then deny registration.” Carole said.

Avery grabbed the card and wrote: ‘10s time out & reject’.

“What if the suit is already registered?” I asked.

“Reject the registration and do not send the confirmation message to production.” Replied Carole.

I reached for the card but Avery was already writing: ‘If already reg’d, reject reg & don’t send conf to prod.’

“OK, one last thing.” Said Carole. “Once registered, the suit should be scheduled for an inspection. This is just a flag in the database record that will prevent the suit from being checked out for use until it has been inspected.”

Avery still held the card, almost as if he owned it. He scribbled: 'Sched for inspection' on it and continued to hold on to it with a proprietary demeanor.

"Are there any more questions about this story?" asked Carole. There was silence.

Jerry said: "OK, let's estimate it. Jasper, Alphonse, Avery, this story has a few complications, but it's pretty simple overall. I suggest we estimate it at four."

Jasper nodded, but I was confused. "Four what?" I asked. "Man hours?"

"No, just four." Replied Jerry. We don't assign units to these estimates; we just use them to compare one story to another. So a story that's twice as hard would be an eight. One half as hard would be a two."

Jasper reached for the card, and there was an awkward moment where it appeared that Avery would not relinquish it. But then, with a discernable grimace, he handed the card to Jasper. In the upper right corner of the card Jasper wrote the number four and drew a circle around it. Then he placed the card back on the table. Avery made a move towards it, but then thought better of it and backed off.

Carole said: "OK, now how do we test this?"

"Test it? What do you mean?" I asked.

Carole looked meaningfully at Jerry and said: "Would you care to answer your apprentice, Jerry?"

Jerry sighed and looked up at me with an air of inevitability. "Alphonse, Avery, so far you've been working on very simple systems that were designed more for your training than for actual use. DTrack is a production system, and the rules are a bit different. In a production system every requirement is specified in terms of executable acceptance tests. When the acceptance tests pass, then the requirement is done."

"Are acceptance tests like unit tests?" I asked.

"No, not at all. Acceptance tests look just like requirements. They can be read by all the stakeholders and officers. Most can write them too. Even Carole can write them." Jerry shot an evil glance towards Carole who responded by sticking out her tongue. "They are written in a system called FitNesse, which allows anyone with appropriate access to read them, write them, change them, and even execute them."

"How can they look just like requirements?" Avery asked, genuinely interested.

Carole stepped over and said: "You're about to find out. Jerry, let's write up an acceptance tests for Register Suit". And the two of them sat down at a terminal and began to type.

To be continued...

The Craftsman: 26

Dosage Tracking III

A Tabled Requirement.

Robert C. Martin
11 May 2004

...Continued from last month.

21 Feb 2002, 0900

Our ship, the *Dyson*, can accelerate at 1G for 60 months without refueling. During the first year of an interstellar trip we accelerate to nearly C. During the last year we decelerate to the reference frame of our destination. At speeds close to C the rest of the universe is subjectively so small that we can go just about anywhere in a few weeks or months. Theoretically that means we could go just about anywhere in the universe in two subjective years.

In the last 43 years we've stopped at ten stellar systems. We spend a year or so at each one, exploring, refueling and refitting. Then it's off to the next – in search of a home for the million frozen embryos on board. Usually we stop accelerating at about .999C, and coast for several months as a way to conserve fuel. There have been two stellar systems that did not provide us with enough Uranium to adequately refuel, and the captain isn't the kind of person to trade time for risk.

Right now we are at peak velocity, and have been coasting for a few months. We've got another eight months to go before we've crossed the 20 light-years of this jump, and start to decelerate. So we're going to plow through that molecular hydrogen cloud in front of us at a screaming clip.

Avery and I stood directly behind Jerry and Carole as they began to work on the acceptance test for the *Register Suit* story. Jerry opened a network browser and went to a site named FitNesse¹. He did some things on the screen that I didn't follow but quickly wound up at a page whose name appeared to be *RegisterNormalSuit*.

"OK." He said. "Let's describe what normally happens when we add a suit."

"What do you mean by normal?" asked Avery.

Carole said: "Nothing goes wrong. Everything works as it is supposed to."

"Right." Said Jerry. "So first we want to assert that the current suit inventory is empty."

"What's normal about that?" Avery sneered.

"Avery dear," Jean said soothingly, "we start by making the simplest assumptions. Don't worry, we'll also specify the cases where the database is not empty. Right now, though, it's just easier to assume that there aren't any suits in the database."

Jerry typed something and the following table appeared on the screen.

Suit inventory parameters

¹ <http://fitnesse.org>

Number of suits?
0

“What’s that?” I asked.

“It’s an assertion.” Jerry responded. “I am asserting that there are no suits in the database. Next I want to assert that there is a request to register a new suit.”

He continued to type, and another table appeared right below the first:

Suite Registration Request
bar code
314159

“I’m confused.” I said. “Why is there a question mark in the first table, but not in the second?”

Jasper walked over and gave me a big toothy grin. “Woah, Jerry, you’ve got a bright one here. Good eye, Alphonse!”

Out of the corner of my eye I could see Avery stiffen. He didn’t like Jasper complimenting me.

“The reason” said Jerry, “is that the first table is a query, whereas the second is a statement. In the first table we are asking the system how many suits are in the database. In the second table we are telling the system that suit number 314159 is being registered.”

“That’s right.” blurted Avery quickly, “you see, Alphonse, the first table can be checked, but the second table is just a fact.”

Jasper’s eyebrows shot up. “Very *good* Avery! Gosh, Jerry, I think we’ve got two keepers here.”

Avery stood next to me, smiling, but I could tell he was feeling superior.

“Wait.” I said. “What do you mean the first table can be checked? That doesn’t make a lot of sense to me.”

Avery started to answer but couldn’t seem to find the words. “Uh, well, the uh…”

Jerry stepped in and rescued him. “FitNesse is going to execute these tables.” He said. “When it executes the first table it will check to be sure that the number of suits is zero. That’s what the question mark tells FitNesse to do. If the number of suits is not zero, then that cell in the table will turn red, otherwise it will turn green.”

Avery blinked, but didn’t say anything. I, on the other hand, forged right on ahead. “You mean the cells change color?”

“Right.” Said Jerry. He pointed to the screen that held the two tables. “Do you see the ‘Test’ button on the screen? When I push it, FitNesse will read the tables one by one. For each table it will pass some data into the DTrack system, and read some other data out. The question mark is for data that comes *out* of DTrack. If the data coming out matches the data in the table, then FitNesse turns the cell green. Otherwise it turns it red.”

Avery muscled in again. “I see! So the first table *asks* DTrack how many rows are in the suit database, and will turn red if any number other than zero is returned. The second table *tells* DTrack to register suit 314159.”

“That’s about it.” Said Jasper Jovially.

“Let’s finish this test.” Said Carole impatiently.

Jerry turned back to the console. “OK, so next we want to make sure that the appropriate message gets sent to manufacturing.”

Message sent to manufacturing		
message id?	message argument?	message sender?
Suit Registration	314159	Outside Maintenance

Avery looked puzzled. “So when FitNesse executes this table, will it ask Manufacturing what messages it received?”

“No,” said Jasper. “We’ll catch the message before it goes.” Then he looked at Carole mischievously, and said “It would sure burn Courtney’s breeches if we brought her system down by sending wild messages every time we tested, eh Carole?”

Carole rolled her eyes and said “Let’s try to focus here. What’s next?”

I said: “I guess we need to assert that manufacturing sends back an acceptance message.”

“Right you are, Alphonse.” Said Jerry as he typed in the appropriate table.

message id	message argument	message sender	message recipient
Suit Registration Accepted	314159	Manufacturing	Outside Maintenance

“You forgot the question marks.” Avery said. You could tell he was pleased to catch Jerry in an error.

“Did I?” Jerry replied.

Avery looked uncomfortable. I thought I knew why Jerry had left the question marks off, but I held my peace.

Carole said: “No question marks are necessary because we are telling the DTrack system that Manufacturing sent this message. We aren’t asking.” Carole’s patience was clearly wearing thin. She wanted to get this done. “What’s next Jerry?”

“OK, having received that message, the DTrack system should enter the suit into the database. So now the database should have the suit in it.”

Suits in inventory	
bar code	next inspection date
314159	2/21/2002

“Why did you put today’s date as the inspection date?” I asked.

Avery said: “Because, Alphonse, according to Carole’s story, newly registered suits have to be scheduled for inspection.”

“Yes, I remember that,” I replied, “but why *today’s* date? The test won’t work if we run it tomorrow. Are we going to have to change that date every day that we run this test?”

Jasper quipped “That’s your job, Jerry. We want you to come in every morning and change the date.”

“No thanks” Said Jerry. “No, we need to specify ‘today’s’ date in the test. So let’s do that as the first table.”

DTrack Context
Today’s date
2/21/2002

“OK.” Said Carole; still trying to move things along. “I think that’s the story. Now let’s dress it up a little.” She took the keyboard and began to write words around the tables. When she was done, the page looked like this:

Normal suit registration.

- *We assume that today is 2/21/2002.*

DTrack Context
Today’s date
2/21/2002

- *We also assume that there are no suits in inventory.*

Suit inventory parameters
Number of suits?
0

- *We register suit 314159.*

Suit Registration Request
bar code
314159

- *DTrack sends the registration confirmation to Manufacturing.*

Message sent to manufacturing		
message id	message argument	message sender
Suit Registration	314159	Outside Maintenance

- *Manufacturing accepts the confirmation.*

Message received from manufacturing			
message id?	message argument?	message sender?	message recipient?
Suit Registration Accepted	314159	Manufacturing	Outside Maintenance

- *And now the suit is in inventory, and is scheduled for immediate inspection*

Suits in inventory	
bar code?	next inspection date?
314159	2/21/2002

“Great!” Said Carole. “A very nice requirement.”

I had to admit, it was pretty clear. But there was something I still didn’t understand.

“How do you get FitNesse to execute those tables?” I asked.

Jerry looked up and said: “Sit down, Alphonse; you too Avery; lets make this requirement turn red!”

To be continued...

The Craftsman: 27

Dosage Tracking IV

Carole's way, or the HighWay.

Robert C. Martin
28 May 2004

...Continued from last month.

21 Feb 2002, 1000

In the first decade of the 20th century, Percival Lowell predicted the existence of a ninth planet by studying the motion of Uranus. Though he searched, he died before he could complete the effort. In 1929 a young man named Clyde Tombaugh came to Lowell Observatory and resumed Lowell's search for planet X. The procedure was both delicate and tedious. Two photographic plates, taken of the same stretch of sky, but on different nights, were put into a device known as a "blinker". The blinker displayed the two plates one at a time, quickly alternating between them. Any object on one plate that was in a different position on the other would appear to blink. Clyde Tombaugh found Pluto on the 18th of February, 1930. In the summer of 1935 he found the angel of death.

The newspapers named the rock "Clyde", and had a few days poking fun at the notion that Clyde might actually hit the Earth in 1959. But the world was headed for war, and even a possible doomsday rock couldn't hold the papers' attention for long. Besides, as the astronomers kept saying, the odds of a collision were millions to one against.

Carole and Jasper grabbed another workstation to work on the next acceptance test, while Avery, Jerry, and I started working on making the *Register Normal Suit* test run.

"OK, the first thing we need to do is write the fixture for the first table." Said Jerry.

"What's a fixture?" I asked.

"A fixture is a Java class that binds the table to the DTrack application." Jerry replied.

"But there is no DTrack application." Avery complained.

"True." Said Jerry. "We write the tests and fixtures first to make sure that the application is designed to be testable."

"Oh, sort of like writing unit tests first." I said.

"Yes, it's a bit like that", replied Jerry, "except we write the whole test and fixture before writing any of the application. Indeed, we write many tests and fixtures before writing the application."

"What does a fixture look like?" Avery asked.

Jerry pulled up the test that he and Carole had just finished. "Look at the first table." He said.

DTrack Context
Today's date
2/21/2002

“Now watch what happens to that table when I hit the *Test* button.”

```
DTrack Context
-----
Could not find fixture: DTrackContext.
Today's date
2/21/2002
```

“Does that mean we have to write a class named DTrackContext?” I asked.

“That’s exactly what it means.” Jerry said, passing me the keyboard. “Put it in the `dtrack.fixtures` package.

So I wrote:

```
package dtrack.fixtures;

public class DtrackContext {
}
```

“Great.” Said Jerry. “Now compile that and run the test again.”

It compiled without a problem, of course; but when I hit the *Test* button, I got the same error as before.

“Do you know why?” Jerry asked.

I thought I did, and I started to answer, but Avery beat me to it by saying: “It’s a classpath issue; FitNesse doesn’t know where the `DTrackContext.class` file is.”

“Right you are, Avery.” Jerry beamed. I could see a smug little smile flicker on Avery’s face.

Jerry took the keyboard and made the following changes to the test page:

```
!path C:\MyProjects\DosageTrackingSystem\classes

! |DTrack Context|
|Today's date|
|2/21/2002|
```

“This tells FitNesse what the classpath of the fixtures is.” Said Jerry.

“But it still fails.” Complained Avery, who had just hit the *Test* button.

Jerry looked expectantly at us.

“Oh!” I said. “The name of the class is `dtrack.fixtures.DTrackContext`, not just `DTrackContext`.”

“Right again!” Said Jerry as Avery Scowled. “Why don’t you make that change?”

So I changed the page to look like this:

```
!path C:\MyProjects\DosageTrackingSystem\classes

! |dtrack.fixtures.DTrackContext|
|Today's date|
|2/21/2002|
```

And it displayed like this:

```
dtrack.fixtures.DTrackContext
Today's date
2/21/2002
```

“I can sort of see the syntax of this.” I said. The strokes are table cell separators. But what is the bang (!) at the beginning?”

“Don’t worry about that for now.” Said Jerry. “You can read up on FitNesse in your spare time. The syntax is pretty easy to get used to. For now, let’s just concentrate on getting this fixture written. So run the test.”

I pushed the Test button, and saw a different kind of failure.

```
dtrack.fixtures.DTrackContext
-----
DTrackContext is not a fixture.
Today's date
2/21/2002
```

“Good!” Said Jerry. “It found the fixture class.”

“Yeah, but it says it isn’t a fixture.” Avery whined.

“I can fix that.” Jerry said, as he took the keyboard. He made the following changes to the fixture class.

```
package dtrack.fixtures;
import fit.ColumnFixture;

public class DTrackContext extends ColumnFixture {
}
```

“That’s better!” Jerry said as he pushed the *Test* button.

```
dtrack.fixtures.DTrackContext
-----
Today's date
-----
Could not find todaysDate.
2/21/2002
```

“Not much!” Said Avery with a smirk.

“What is it looking for?” I asked? “A variable?”

“Bingo!” Said Jerry, who continued typing.

```
package dtrack.fixtures;
import fit.ColumnFixture;
import java.util.Date;

public class DTrackContext extends ColumnFixture {
    public Date todaysDate;
}
```

“Ick!” Cried Avery. “A public variable! That’s not a very OO construct!”

Jerry looked calmly over at Avery and said: “So what?”

“It breaks encapsulation!” Avery sputtered with righteous indignation.

“Fixture classes aren’t encapsulated in the usual manner.” Jerry explained. “Public variables are one of the ways we communicate with them. Anyway this isn’t the time for a lesson on the true principles of OO. Right now we want to get this fixture done.” So he pushed the *Test* button as Avery rolled his eyes impatiently.

```
dtrack.fixtures.DTrackContext
Today's date
2/21/2002
```

Avery burst out with a huge guffaw: “Oh great! All that work to make it look normal again!”.

“Right!” Said Jerry. “Now we know that the fixture is being found and that the data is getting into it as expected.”

“It doesn’t look as nice as it did before.” I said. “That package name dirties things up a bit. I like the way the variable name uses spaces and punctuation. Can’t the fixture name do the same?”

“Indeed it can!” Said Jerry, as he opened the test page and made the following changes:

```
!path C:\MyProjects\DosageTrackingSystem\classes

!3 Normal suit registration.

! | Import |
| dtrack.fixtures |

'' * We assume that today is 2/21/2002.''
! | DTrack Context |
| Today's date |
| 2/21/2002 |
```

This displayed as:

classpath: C:\MyProjects\DosageTrackingSystem\classes

Normal suit registration.

Import
dtrack.fixtures

- We assume that today is 2/21/2002.*

DTrack Context
Today's date
2/21/2002

“OK, that’s much nicer, I said. Now what do we do with that date?”

“Good point.” Said Jerry. “We need to put that date somewhere that the rest of the DTrack system can get it from.”

Avery assumed a superior air and said: “Wouldn’t it be easier if DTrack just used the regular Date class to get today’s date? Why do we have to invent a whole new mechanism for something as simple as today’s date?”

“Because the tests need to control the date in order to make sure the application manages it correctly.” Jerry replied, a little annoyed.

“Yeah, but that’s just extra work that slows us down! We need to have this done in two months!” Avery had raised his voice enough for Carole to overhear. She quickly came over, looked Avery in the eye, and said:

“No, Avery, it’s not extra work, and it doesn’t slow us down. We go much faster when we write these

tests and build systems that are testable. You are right, Avery, we've only got two months. And the only way we'll make it is if we write the tests and follow our disciplines. We've done it both ways -- haven't we Jerry? (he grimaced, but nodded) -- and I can tell you that as long as *I'm* the customer on this project, *we're* going to be writing acceptance tests, and *you* will be doing it the way *Jerry* leads." And she strode back over to Jasper in a huff.

Avery had paled under Carole's tongue lashing. Now he looked at Jerry and me and said: "Whoa!"

"Yeah, she can be a little intense at times." Jerry said calmly. "The point is that we've decided to do things this way, and if you want to be part of the team, you'll have to go along."

"I still think it's a waste of time." Avery said under his breath.

"Suspend your disbelief for awhile." Said Jerry. "Trust me, we're not fools. This is the best way for us to proceed."

Avery shrugged, but nodded. He looked at me and rolled his eyes. I just stayed out of it.

"OK." I said. "Now what about that date? How does our fixture communicate it to the rest of the system, and how should the rest of the system gain access to the date?"

Jerry looked at the two of us, then glanced over his shoulder at Carole who had busied herself with Jasper, and then sheepishly he said: "I think it's time for a break. Let's get out of here for a few minutes and we can talk about why public variables are sometimes appropriate."

So the three of us left the lab and headed for the nearest lounge.

To be continued...

The Craftsman: 28

Dosage Tracking V

An Encapsulation Break

Robert C. Martin
29 June 2004

...Continued from last month.

21 Feb 2002, 1100

As tensions in Europe rose and eventually blossomed into war, Clyde was all but forgotten by the public -- but not by Tombaugh. He continued to track Clyde, computing and re-computing its orbit. His results were published in the appropriate journals, and were noted with growing concern by scientists on both sides of the growing political divide. The odds for a collision with Earth, though still very low, continued to rise with every new measurement.

By early 1939, when it was clear to most scientists that they would soon be unable to communicate with their colleagues on the other side of the Atlantic, Tombaugh decided to hold an astronomical conference in Stockholm. The conference was very well attended, and not just by astronomers. It seemed that scientists of many disciplines felt that this conference could be their last chance for an international gathering.

Clyde was not a major topic of the formal conference, but around the bar, and in the lounges, the talk often drifted to: "what if?" These discussions weren't serious so much as they were entertaining; but they served to generate some interesting ideas. Some were about destroying Clyde, others were about deflecting it, and still others were about escaping to Venus, or Mars.

Despite the range of ideas, everyone agreed that the fundamental problem was energy. All these solutions required energies that were beyond our abilities to generate; and so they were all considered to be bar-room fantasies.

Then, on the last night of the conference, Leo Szilard, Neils Bohr, and Lise Meitner broke the news that their investigations into nuclear reactions indicated that such energies just might be within reach. In view of the political climate, it was clear to everyone there that this news was not all that good. A few stayed very late that night to discuss...options. They whimsically named themselves: The Stockholm Contingent.

Jerry, Avery, and I went to the observation lounge at the high-gee end of the gamma arm. We each got a cup of coffee and sat at a table watching the starbow whirl under our feet.

Jerry looked over the table at Avery and said: "So, Avery, you complained about my use of public variables in the test fixture we were writing."

Avery grimaced and said: "Well, it's not a very OO thing to do."

"Can you define OO for me?" Asked Jerry? "Why are public variables not OO?"

"Because OO is about encapsulation, and public variables aren't encapsulated."

“Do you think that OO requires all variables to be encapsulated?”

“Of course!” replied Avery.

Jasmine was at a nearby table with Adelaide. She seemed interested in this conversation.

Jerry asked: “What bad thing happens if you have an un-encapsulated variable?”

Avery made a disdainful face and said: “Other programs could change your variables!”

“What if that is what you intend? What if you *want* other programs to change certain variables?”

“Well then there’s something wrong with your design!” Avery proclaimed with righteous indignation.

Jasmine loomed over our table and said: “Oh pooh, Avery, that’s just silly.”

Avery hadn’t noticed Jasmine until that moment. When he saw her, the expression on his face changed to a mix between embarrassment and panic. He sputtered as though to answer her, but Jasmine kept right on talking.

“Look you meatballs, it’s not the rules, it’s the reasons. The aim of OO is to help you manage dependencies. Most of the time making your variables private helps you to do that. But sometimes private variables just get in the way.”

Jerry butted in: “Now wait a minute, Jasmine. I agree that OO languages help you to manage dependencies, but the more important benefit is expressivity. It is easier to express concepts using an OO language.”

“Yeah, yeah, sure, sure.” Replied Jasmine. But it’s not lack of expressivity that turns software systems into a tangled mass of interconnected modules; it’s mismanaged dependencies that do that. I can create very expressive systems that are tangled all to hell. I agree that expressivity is important, but keeping the coupling low is much more important!”

“How can you say that? Your approach is so sterile! Where is the art? Where is the finesse? All you care about is that the dependencies go in the right direction.”

“What is artful about nicely named modules that are coupled into a tangled mess? The art is in the structure, Jerry. The art is in the careful management of dependencies between modules.”

“Structure is important, sure, but…”

It seemed to me that this was likely to go on for awhile, and I really wanted to know about the public variables, because they bothered me too. So I interrupted them. “Excuse me, Jasmine, Jerry, but what does this have to do with public variables?”

The two of them looked at Avery and me as though they just remembered we were there. Jerry said: “Oh, uh, sorry about that. This is an old argument between Jasmine and I. Look, we make variables private to tell other programmers to ignore them. It’s a way of expressing to others that the variables aren’t their problem, and that we want them to mind their own business.”

Jasmine rolled her eyes. “Oh, here we go again! Look boys, (I didn’t like being called a boy – especially by Jasmine. I saw Avery flinch too.) we make variables private to reduce coupling. Private variables are a firewall that dependencies cannot cross. No other module can couple to a private variable.”

“That’s such an ice-cold view” said Jerry. “Don’t you see how dehumanizing it is?”

I wanted to stop this before it got going again, so I blurted out my next question: “OK, but when is it OK for a variable to be public?”

Avery seemed to recover himself at that. He looked at me and proclaimed: “Never!”

That stopped Jasmine and Jerry in mid-stride. They both turned to Avery and said: “Nonsense – Ridiculous”.

“Look, Avery” said Jasmine “sometimes you *want* to couple to a variable.”

“Right” Said Jerry. “Sometimes a public variable is the best way to communicate your intent.

“When is that?”

“Well, like in a test fixture.” Jerry replied. “The data used by the fixture comes from one source, but the test is triggered by another. One party has to load the data, and the other party has to operate on it. We want to keep them separate.”

“Right” said Jasmine. “It’s all about the coupling. The important thing is that the source of the data is

not coupled to the application being tested. The fact that the variables in the fixture are public is harmless.”

“But what if someone uses them?” complained Avery?

“Who would?” Replied Jerry. “They are variables in a test fixture. Everybody on this project knows that those variables are under the control of FitNesse¹ and aren’t for anyone else to use.”

“But they might!”

Jasmine rolled her eyes and said: “Yeah, and they might take one of your private variables and make it public. What’s the difference?”

Avery spluttered some more, but couldn’t seem to face her down. So I asked the obvious question: “But couldn’t you use setters and getters? Why make the variables public?”

“Exactly!” Avery splurled.

“Why bother?” Asked Jerry.

“Right,” Said Jasmine. “It’s just extra code that doesn’t buy you anything.”

“But I thought getters and setters were the right way to provide access to variables?” I replied.

“Getters and setters can be useful for variables that you specifically want to encapsulate.” Said Jerry, but you don’t have to use them for all variables. Indeed, using them for all variables is a nasty code smell.”

“Right!” responded Jasmine with a grin. “PeeYew! There’s nothing worse than a class that has a getter and setter for each variable, and no other methods!”

“But isn’t that how you are supposed to implement a data structure?” I asked.

“No, not at all.” Replied Jasmine. “A class can play two roles in Java. The first is as an object, in which case we usually make the variables private and provide a few getters and setters for the most important variables. The other role a class can play is as a data structure, in which case we make the variables public and provide no methods. Test fixtures are primarily data structures with a few methods for moving the data back and forth between FitNesse and the application under test.”

“Right.” Said Jerry. “A class that has getters and setters but no other methods isn’t an object at all. It’s just a data structure that someone has wasted a lot of time and effort trying to encapsulate. That encapsulation buys nothing in the end. Data structures, by definition, aren’t encapsulated.”

“Agreed.” Said Jasmine. “Objects are encapsulated because they contain methods that change their variables. They don’t have a lot of getters and setters because the data in an object remains hidden for the most part.”

“Right.” Said Jerry.

“Right.” Said Jasmine.

And then the two of them smiled at each other in a way that made my stomach turn. Avery looked like he had just bitten into a lemon.

To be continued...

¹ <http://fitnesse.org>

The Craftsman: 29

Dosage Tracking VI

Move the Date

Robert C. Martin
6 August 2004

...Continued from last month.

21 Feb 2002, 1130

Between April and July of 1939 the political and astronomical news continued to worsen. War seemed inevitable, and the Earth remained at the center of Clyde's most probable path. The Stockholm Contingent changed from a small gathering of scientists in the bar, to a small and secret network of scientists around the world. Though the group had no official leader, it was Lise Meitner who provided vision and direction. Based in Stockholm, she remained able to communicate with most of her colleagues around the world. Slowly and carefully she began to recruit them.

By September, when the fits and starts of hostilities finally erupted into full-scale war in Europe, the Stockholm Contingent had members in neutral and hostile nations alike. This group did not know exactly what it was going to do, but they were convinced that the world should be paying more attention to Clyde than to fighting. They were also convinced that the newly discovered Uranium fission reactions were the key defense against Clyde, and too terrible to be used as a weapon of war. They also knew that the warring states would consider their views to be treasonous.

Jerry and Jasmine walked ahead of us on our way back to the lab. The whole way they talked and laughed, and acted all buddy-buddy. Avery and I looked disgustedly at each other, but kept quiet.

Back in the lab, we sat down and started working on the test fixture again.

"So, what are we going to do with that date?" asked Jerry.

"We need to save it so that the application can access it when it needs to know today's date." I replied.

Avery still didn't like this, and made his opinion known by rolling his eyes; but he stayed mute.

"Right." Said Jerry. "Now let me show you how we do that."

He typed the following code.

```
public class UtilitiesTest extends TestCase {
    public void testDateGetsTodayWhenNoTestDateIsSpecified() throws Exception {
        Date now = new Date();
        Utilities.testDate = null;
        assertEquals(now, Utilities.getDate());
    }
}
```

Jerry looked over at Avery and asked: “Can you make that pass Avery?”

Avery sighed disdainfully as he took the keyboard. He said: “I suppose you’ll want me to use a *public* variable.” And then he typed the following without waiting for an answer:

```
public class Utilities {  
    public static Date testDate = null;  
  
    public static Date getDate() {  
        return new Date();  
    }  
}
```

He clicked the test button and got a green bar.

“Great!” Said Jerry.

But Avery looked impatiently at Jerry and said:

“Yeah, but watch this.”

Avery started clicking the test button repeatedly. He got three or four green bars, and then the test failed with the message:

```
junit.framework.AssertionFailedError:  
expected:<Tue Feb 21 11:33:16 2000 (subjective)>  
but was: <Tue Feb 21 11:33:16 2000 (subjective)>
```

Before Jerry could react, Avery said: “You wrote the test wrong. Sometimes the two dates won’t be exactly the same. The difference is probably as small as a millisecond, but it’s enough to fail the test. I’ll fix that by writing the test a bit more intelligently.” And he changed the test as follows:

```
public void testDateGetsTodayWhenNoTestDateIsSpecified() throws Exception {  
    GregorianCalendar now = new GregorianCalendar();  
    GregorianCalendar systemDate = new GregorianCalendar();  
    Utilities.testDate = null;  
    now.setGregorianChange(new Date());  
    systemDate.setGregorianChange(Utilities.getDate());  
    long difference = now.getTimeInMillis() - systemDate.getTimeInMillis();  
    assertTrue(Math.abs(difference) <= 1);  
}
```

And then Avery turned towards Jerry and hit the test key repeatedly. He never once looked at the screen, but the test passed every time.

Jerry looked back at Avery coldly and said. “I concede the point, Avery. Thank you. However, that function is a bit cluttered now.” And Jerry took the keyboard back and made the following changes:

```
public void testDateGetsTodayWhenNoTestDateIsSpecified() throws Exception {  
    Utilities.testDate = null;  
    assertTrue(DatesAreVeryClose(new Date(), Utilities.getDate()));  
}  
  
private boolean DatesAreVeryClose(Date date1, Date date2) {  
    GregorianCalendar c1 = new GregorianCalendar();  
    GregorianCalendar c2 = new GregorianCalendar();  
    c1.setGregorianChange(date1);  
    c2.setGregorianChange(date2);  
    long differenceInMS = c1.getTimeInMillis() - c2.getTimeInMillis();  
    return Math.abs(differenceInMS) <= 1;  
}
```

“OK, I think that expresses our intent a bit better.” Jerry said as he watched the test repeatedly pass. Avery just sniffed.

I was feeling a bit like I was in a war zone. I didn’t know why Avery and Jerry were showing such animosity towards each other, but I was concerned that it was distracting them from the task at hand. So I grabbed the keyboard and said:

“OK, I think I know what the next test is.” And I wrote:

```
public void testThatTestDateOverridesNormalDate() throws Exception {
    Date t0 = new GregorianCalendar(1959, 11, 5).getTime();
    Utilities.testDate = t0;
    assertEquals(t0, Utilities.getDate());
}
```

Jerry watched the test fail and said: “Right Alphonse. Now make it pass.”

But Avery grabbed the keyboard and said: “I will.” And he wrote the obvious code:

```
public static Date getDate() {
    return testDate != null ? testDate : new Date();
}
```

And as the test passed, he muttered: “And that’s one hell of a lot of time wasted to get one simple and obvious line of code working.”

“It wasn’t *wasted*....” Jerry started. But then he stopped and shook his head. He took a deep breath and said: “OK, now we have to get the DTrackContext fixture to set the testDate. So he wrote the following test:

```
public class DTrackContextTest extends TestCase {
    public void testExecuteSetsTestDate() throws Exception {
        Date t0 = new GregorianCalendar(1959, 11, 5).getTime();
        DTrackContext c = new DTrackContext();
        c.todaysDate = t0;
        Utilities.testDate = null;
        c.execute();
        assertEquals(t0, Utilities.testDate);
    }
}
```

As I watched the test fail, I said: “OK Jerry, I see what you’ve done. You’ve created an instance of the fixture and jammed the t0 date into todaysDate just like FitNesse¹ would. Then you expect t0 to somehow get set into the Utilities.testDate field. I suppose that it’s the execute method of the fixture that does this?”

“Right. Remember that the FitNesse table we are working on looks like this:” and he pulled the table up on the screen.”

DTrack Context
Today's date
2/21/2002

“As FitNesse processes this table it will jam the 2/21/2002 date into the todaysDate field of the DTrackContext fixture, and then it will call execute() on that fixture.”

“OK, then I can make this test pass by doing this:” and I grabbed the keyboard and typed:

¹ www.fitnesse.org

```

public class DTrackContext extends ColumnFixture {
    public Date todaysDate;

    public void execute() throws Exception {
        Utilities.testDate = todaysDate;
    }
}

```

The tests passed, and I understood a bit more about how FitNesse worked. Then Jerry said: “You should run the acceptance test too.” So I went to the RegisterNormalSuit page on the FitNesse server and clicked the test button. There was no change.

```

DTrack Context
Today's date
2/21/2002

```

“Good.” Said Jerry. “We haven’t broken anything.”

“We haven’t accomplished anything either.” Said Avery.

Jerry glared at Avery, looked sadly over at me, then sighed and got out of his chair and walked over to Jean. The two of them started talking quietly. I looked at Avery and said: “Avery, what’s wrong, why are you complaining so much?”

“He a dufus!” Said Avery. “He doesn’t know *anything* about object oriented design, and all this testing nonsense is just make-work for morons. We’ve been working on this suit registration requirement for nearly four hours, and the only production code we’ve got to show for it is this:”, and he pulled up the Utilities class on the screen.

```

public class Utilities {
    public static Date testDate = null;

    public static Date getDate() {
        return testDate != null ? testDate : new Date();
    }
}

```

“And even this class is nothing but a hack to allow testing. Four hours, three people, that’s twelve man hours, and we’ve done jack cheese! I think this whole group is wacked out. I think Mr. C must be an idiot. How can he think he’ll get this DTrack project done in two months when we squander hours and hours doing nothing but these stupid tests? If I were the captain I’d put Mr. C. out the airlock.”

As Avery ranted, his voice grew louder and his eyes started to bug out. He didn’t notice Jerry and Jean walking up behind him.

“That will be quite enough of that, young man!” Jean said sternly.

Avery snapped around, and his face went from anger to dismay. Jean was no longer the kindly grandmother we had come to know. She was something else entirely. It was clear from her tone of voice and the expression on her face that we were to remain silent and attentive, and that we were not to move.

“Speaking of the captain and his chief software craftsman that way is not acceptable behavior in this department. I won’t tolerate it. Avery, yesterday you insulted Jason so badly he refused to work with you anymore. So I put you to work with Alphonse to see if any of his good manners might rub off on you. You two seemed to be getting along, and I dared hope that you had learned your lesson. But now you’ve been hostile to Jerry, and to the whole team, department, and even the *ship*. What shall we do with you?”

To be continued...

The source code for this article can be found at:

`www.objectmentor.com/resources/articles/CraftsmanCode/Craftsman_29_DosageTrackingSystem.zip`

The Craftsman: 30 Dosage Tracking VII The “Woodshed”

Robert C. Martin
3 September 2004

...Continued from last month.

As 1939 waned, and Europe descended into the chaos of war, the Earth remained centered within Clyde's narrowing path. Though the probability of a collision was still quite small, it continued to grow. The members of the Stockholm Contingent, communicating through Lise Meitner's secret network, eventually concluded that their only responsible course was to act as though collision were certain.

The Contingent faced a dilemma. So far they had managed to keep the discovery of Uranium fission within their membership. However, if atomic power were to be used as a defense against Clyde, it would take the resources of a very rich nation to gain the necessary technology in time. In Europe, all such nations were at war, and would certainly use that technology to produce terrible weapons.

The Contingent resolved to go where the war had not yet reached. Leo Szilard, a founding member of the Contingent, convinced Albert Einstein to write a letter to Franklin Roosevelt on behalf of the Contingent. The import of this letter was underscored by the fact that a quorum of the Contingent had somehow managed to arrive on U.S. soil by the time the letter was delivered.

21 Feb 2002, 1230

Jean took Avery to a small two-man conference room with glass walls. They were still in there when Jerry and I got back from lunch. The look on Avery's face made me glad that I was not the one in that room with Jean.

Jerry said, “We'd better get back to work. Let's see how much progress we can make by the time he gets out.”

“Do you think he'll be back?” I asked?

Jerry glanced over at Avery and Jean with a knowing look on his face. “I'm pretty sure of it, Alphonse. Now let's get to work.”

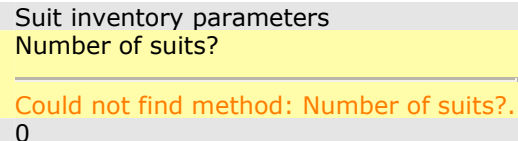
We sat at our workstation and Jerry ran the RegisterNormalSuit acceptance test. The first two tables showed no errors, but the third look like this:

Suit inventory parameters
Could not find fixture: SuitInventoryParameters.
Number of suits?
0

“So we need a fixture named `SuitInventoryParameters`, right?” I asked.
“Right” said Jerry. “Go ahead and see if you remember how to write it.”
So I grabbed the keyboard and typed the following:

```
public class SuitInventoryParameters extends ColumnFixture {  
}
```

Running the acceptance test now produced this:



```
Suit inventory parameters  
Number of suits?  
-----  
Could not find method: Number of suits?..  
0
```

“That’s not quite what I expected.” I said. “The last time we did this¹ it didn’t say anything about a method. It wanted a variable.”

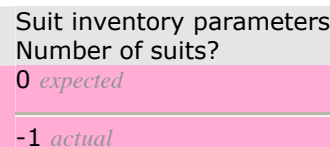
“That’s right Alphonse, but this time it wants a method because there is a question mark following the name.”

“Oh, right, you mentioned something about that earlier this morning. Question marks mean that the table is asking a question of the application. So we are asking the system how many suits it has in inventory?”

“Right. So go ahead and write that method.”

```
public class SuitInventoryParameters extends ColumnFixture {  
    public int numberOfSuits() {  
        return -1;  
    }  
}
```

Now the acceptance test looked like this:



```
Suit inventory parameters  
Number of suits?  
-----  
0 expected  
-1 actual
```

“Good.” Jerry said. “Now connect the fixture to the application.”

“What function in the application gets the number of suits in inventory?” I asked.

“Good question.” Jerry replied. “What function do you think it should be?”

I looked over the code in the project for a few seconds and said: “We could put it in the `Utilities` class for now.”

“We could. I don’t think it will stay there long though.”

So I modified the code as follows:

```
public class SuitInventoryParameters extends ColumnFixture {  
    public int numberOfSuits() {  
        return Utilities.getNumberOfSuitsInInventory();  
    }  
}
```

¹ September, 2004, “Swiss Wisdom”: <http://www.sdmagazine.com/documents/s=7764/sdm04091>

```

    }
}

public class Utilities {
    public static Date testDate = null;

    public static Date getDate() {
        return testDate != null ? testDate : new Date();
    }

    public static int getNumberOfSuitsInInventory() {
        return -1;
    }
}

```

There was no change in the acceptance tests results. “Shall I make this table pass now?” I asked Jerry. “No, let’s connect up the other fixtures first.” He replied. The next table in the test looked like this:

Suit Registration Request bar code 314159

Connecting it up to FitNesse² was easy.

```

public class SuitRegistrationRequest extends ColumnFixture {
    public int barCode;
    public void execute() {
        Utilities.registerSuit(barCode);
    }
}

public class Utilities {
    ...
    public static void registerSuit(int barCode) {
    }
}

```

The next table was a bit trickier. It looked like this:

Message sent to manufacturing		
message id?	message argument?	message sender?
Suit Registration	314159	Outside Maintenance

The basic structure of the fixture was easy enough. The question marks told me that each of the column headers was a method. So I wrote the following.

```

public class MessageSentToManufacturing extends ColumnFixture {
    public String messageId() {
        return null;
    }

    public int messageArgument() {
        return -1;
    }
}

```

² www.fitnesse.org

```

public String messageSender() {
    return null;
}
}

```

But then I was stuck. “How do I connect this to the application?” I asked.

“Do you remember what this table is checking?”

“Sure, we’re verifying the contents of the message that DTrack is supposed to send to the manufacturing system.”

“Right. So you need to get that message, and unpack it.”

“How do I do that? None of the methods in this fixture seem to be the appropriate place.”

“I agree. They aren’t. However, FitNesse gives you another choice. The `execute` method in `ColumnFixture` is called before any column header methods are called. So in the `execute` method you could ask DTrack for a copy of the message that was sent, and then the column header methods could unpack that message.”

“OK, I think I get it.” And I changed the code as follows:

```

public class MessageSentToManufacturing extends ColumnFixture {
    private SuitRegistrationMessage message;
    public void execute() throws Exception {
        message = (SuitRegistrationMessage)
            Utilities.getLastMessageToManufacturing();
    }

    public String messageId() {
        return message.id;
    }

    public int messageArgument() {
        return message.argument;
    }

    public String messageSender() {
        return message.sender;
    }
}

public class SuitRegistrationMessage {
    public String id;
    public int argument;
    public String sender;
}

public class Utilities {
    ...
    public static Object getLastMessageToManufacturing() {
        return new SuitRegistrationMessage();
    }
}

```

This made the table look like this:

Message sent to manufacturing		
message id?	message argument?	message sender?
Suit Registration <i>expected</i>	314159 <i>expected</i>	Outside Maintenance <i>expected</i>
null <i>actual</i>	0 <i>actual</i>	null <i>actual</i>

The next table was very simple. It looked like this:

Message received from manufacturing			
message id	message argument	message sender	message recipient
Suit Registration Accepted	314159	Manufacturing	Outside Maintenance

I connected it to DTrack using the following fixture:

```
public class MessageReceivedFromManufacturing extends ColumnFixture {
    public String messageId;
    public int messageArgument;
    public String messageSender;
    public String messageRecipient;
    public void execute() {
        SuitRegistrationAccepted message =
            new SuitRegistrationAccepted(messageId,
                                         messageArgument,
                                         messageSender,
                                         messageRecipient);
        Utilities.acceptMessageFromManufacturing(message);
    }
}

public class SuitRegistrationAccepted {
    String id;
    int argument;
    String sender;
    String recipient;

    public SuitRegistrationAccepted(String id, int argument,
                                    String sender, String recipient) {
        this.id = id;
        this.argument = argument;
        this.sender = sender;
        this.recipient = recipient;
    }
}

public class Utilities {
    ...
    public static void acceptMessageFromManufacturing(Object message) {}
}
```

“The Utilities class is collecting an awful lot of cruft.” I complained.

“Yes, it is. We’ll go back and refactor it as soon as we get this acceptance test to pass. But for now let’s get that last table connected.”

I sighed and looked at the last table. It was a bit different:

Suits in inventory	
bar code?	next inspection date?
314159	2/21/2002

“It looks like this table could be asking for more than one suit.” I said.

“Well, the table only expects one suit, but one of the failure modes is that the number of suits is not one. You have to use a different kind of fixture for this. Let me show you.”

Jerry grabbed the keyboard and wrote the following:

```

public class SuitsInInventory extends RowFixture {
    public Object[] query() throws Exception {
        return Utilities.getSuitsInInventory();
    }

    public Class getTargetClass() {
        return Suit.class;
    }
}

public class Suit {
    public Suit(int barCode, Date nextInspectionDate) {
        this.barCode = barCode;
        this.nextInspectionDate = nextInspectionDate;
    }

    private int barCode;
    private Date nextInspectionDate;

    public int barCode() {
        return barCode;
    }
    public Date nextInspectionDate() {
        return nextInspectionDate;
    }
}

public class Utilities {
    ...
    public static Suit[] getSuitsInInventory() {
        return new Suit[0];
    }
}

```

When Jerry ran the test, the table looked like this:

Suits in inventory	
bar code?	next inspection date?
314159 <i>missing</i>	2/21/2002

I looked at this code for a few minutes and then said: “I think I understand. You overrode the query() method of SuitsInInventory to return an array of Suit objects. You also overrode the getTargetClass() method to return Suit.class. Any item listed in the table but not present in the array is marked as missing.”

“Right.” Said Jerry. “Moreover, if the query() method had returned more than one Suit object, it would have been marked as extra.”

“OK, so now we’ve got the whole test page connected to the DTrack application. Now let’s make it pass.”

“Right. I’ll bet we can do that before Avery get’s out of the woodshed.”

“The woodshed?”

“That’s what we call that little class walled conference room.”

I looked over at Avery and Jean in the woodshed. It looked like a pretty heavy, and one-sided conversation.

To be continued...

The source code for this article can be found at:

`www.objectmentor.com/resources/articles/CraftsmanCode/Craftsman_30_DosageTrackingSystem.zip`

The Craftsman: 31

Dosage Tracking VIII

Turn off this Force Field

Robert C. Martin
30 October 2004

...Continued from last month.

The Nimbus Project was begun in the early months of 1940. By mid-year, as the Germans marched into Paris, and the skies over Britain were darkened by German bombers, General Leslie R. Groves and J. Robert Oppenheimer were establishing a huge facility at Los Alamos, New Mexico.

FDR took the war in Europe much more seriously than the threat of a “rock from outer space”, but also understood what a gift the Contingent had bestowed upon America. The official charter of the Nimbus project was to develop atomic, electronic, and rocketry technologies necessary to defend the United States against foreign enemies. Indeed, this is all that the generals and policy-makers really expected. It was just a side benefit that these technologies were exactly those that the Contingent insisted were necessary as a defense against Clyde.

Meanwhile Clyde’s target ellipse continued to shrink, with Earth remaining near its center. As 1940 drew to a close the odds of a collision were still thousands to one against; yet when Werner Von Braun successfully launched an A4 rocket in the New Mexican desert, the target he had on his mind was not on Earth.

21 Feb 2002, 1330

Jerry and I took a quick break. The sweet strains of *Turn off this Force Field* were lilting from the speakers in the break room. The sad melody made me think of Avery in the Woodshed with Jean. I wondered what was going on in there.

Avery and Jean were still in there when we returned from the break. Jerry and I looked at each other as we sat down, but said nothing about what was on both our minds. Instead, Jerry said: “OK, let’s make this acceptance test pass. Here is the first table that’s failing.”

Suit inventory parameters	
Number of suits?	
0	<i>expected</i>
-1	<i>actual</i>

“Right.” I said. “We can easily make it pass by just changing the appropriate method in the `Utilities` class.” So I grabbed the keyboard and typed:

```
public static int getNumberOfSuitsInInventory() {  
    return 0;  
}
```

```
}
```

Sure enough the test turned green.

Suit inventory parameters
Number of suits?
0

But Jerry was shaking his head. “No Alphonse, we don’t want to make it pass that way. We want to make it *really* pass.”

I pointed to the green cell on the screen and said: “What do you mean? It looks to me like the test is really passing.”

“Acceptance tests aren’t the same as unit tests. We don’t use them for the same purposes. We use unit tests to help us design our classes and methods. We use acceptance tests to make sure that our system behaves as specified. Instead of simply making the acceptance test turn green, what we really want to do is develop the `getNumberOfSuitsInInventory` method so that it works the way it should. And for that, we’re going to need some unit tests.”

“I’m not sure I follow you.” I said. “Aren’t we supposed to do the simplest thing we can think of to make our tests pass?”

“For unit tests that close to the truth. But we make acceptance tests pass with code that’s been well unit tested.”

“OK, I think I get it. If I’m tempted to do something ultra simple to get an acceptance test to pass, it means I should really be writing a unit test.”

Jerry smiled. “That’s a good way to think about it.”

“OK, so let’s add a test case to `UtilitiesTest`.” I took the keyboard again and wrote:

```
public void testNoSuitsInInventory() throws Exception {
    assertEquals(0, Utilities.getNumberOfSuitsInInventory());
}
```

This test passed immediately, as expected. So then I wrote:

```
public void testOneSuitInInventory() throws Exception {
    Utilities.addSuit(new Suit(1, new Date()));
    assertEquals(1, Utilities.getNumberOfSuitsInInventory());
}
```

This didn’t compile because there was no `addSuit` method. So I continued to write:

```
public static void addSuit(Suit suit) {
}
```

And then I stopped. “How should we add the suit?” I asked.

“Good question.” Said Jerry. “Where do you think it should go?”

“Well, I guess we need a database of some kind. Should we set one up now?”

“No, it’s too early to do that now.” Jerry replied.

“Well then I can’t finish this method.” I said.

“Sure you can.” Jerry urged. “Just use an interface.”

“An interface to what?” I didn’t understand what he was telling me.

Jerry sighed and said: “An interface to a gateway.”

The melodic softness of *Turn Off This Force Field* started running through my head again. Jerry’s words weren’t making any sense, so I just kind of stared at him while humming the melody in my head.

After about fifteen seconds Jerry rolled his eyes, grabbed the keyboard, and said:

“One way to deal with a database is to create an object called a gateway. A gateway is simply an object that knows how to operate on records in a database. It knows how to create them, update them, query them, delete them, and so forth. In this case we are going to create something called a TABLE DATA GATEWAY¹. A TDG knows how to operate on rows within a particular database table.”

Jerry typed the following:

```
package dtrack.gateways;

import dtrack.dto.Suit;

public interface SuitGateway {
    public void add(Suit suit);
}
```

“OK, I think I see.” I said. So we should be able to modify theUtilities class like this.” And I took the keyboard and typed:

```
public class Utilities {
    ...
    private static SuitGateway suitGateway;

    public static void addSuit(Suit suit) {
        suitGateway.add(suit);
    }
}
```

Everything compiled, so I hit the test button by habit. We got a `NullPointerException` in `addSuit`, just as you’d expect.

“Now what?” I said.

“Create an implementation of `SuiteGateway` that keeps the suits in RAM.” Said Jerry.

“We’re not going to *leave* it that way, are we? I mean these suits *do* need to get written to a *real* database don’t they?”

Jerry looked at me a little oddly and said: “What are you afraid of, Alphonse? Do you think we’ll forget to store the data on disk?”

“No, it’s just that this seems...I don’t know...out of order somehow.”

“Well, it’s not out of order, let me tell you. One of the worst ways to design a system is to think of the database first. I know.” Jerry glanced over to the woodshed for a moment. “Believe me, I know. What we want to do is push off decisions about the database for as long as possible. Eventually, we’ll figure out some way to make sure the suits are stored on disk. I don’t know whether we’ll use a *real* database (whatever that is) or not. Perhaps we’ll just take all the data in RAM and write it to flat files periodically. I just don’t know right now, and I don’t *want* to know. We’ll work out those details as we go along.”

I didn’t like the sound of that. It seemed to me that you could paint yourself into a corner pretty quickly if you didn’t think about the database up front.

“Having the Database Argument?”

It was Carole. She must have overheard us talking. Jerry nodded knowing and said: “Right on schedule.”

Carole smiled and said: “Alphonse, remind me to tell you about the first Dosage Tracking System Jerry and I wrote a few years ago.”

“Don’t you dare!” Jerry retorted with mock fear and anger.

Carole smiled, shook her head, and walked back to her workstation.

¹ *Patterns of Enterprise Application Architecture*, Martin Fowler, Addison Wesley, p. 144

“OK, Alphonse, let’s write a simple RAM based implementation of SuitGateway.” So I took the keyboard and typed:

```
public class InMemorySuitGateway implements SuitGateway {
    private Map suits = new HashMap();
    public void add(Suit suit) {
        suits.put(new Integer(suit.barCode()), suit);
    }
}
```

Then I modified the Utilities class to create the appropriate instance.

```
public class Utilities {
    ...
    private static SuitGateway suitGateway = new InMemorySuitGateway();
}
```

And now the test failed with:

expected:<1> but was:<0>

“OK, Alphonse, you know how to make this pass, don’t you?”

I nodded, and added the remaining code.

```
public class Utilities {
    ...
    public static int getNumberOfSuitsInInventory() {
        return suitGateway.getNumberOfSuits();
    }
}

public interface SuitGateway {
    public void add(Suit suit);
    int getNumberOfSuits();
}

public class InMemorySuitGateway implements SuitGateway {
    private Map suits = new HashMap();
    public void add(Suit suit) {
        suits.put(new Integer(suit.barCode()), suit);
    }

    public int getNumberOfSuits() {
        return suits.size();
    }
}
```

And now all the unit tests passed, and so did the suit Inventory Parameters table of the acceptance test.

I looked at this code for a few seconds and then I said: “Jerry, I’m not so sure that this Utilities class is named very well.”

“Really?” He said – though I could tell he was smiling inwardly.

“Yeah, and I’m not so sure that static functions like getNumberOfSuitsInInventory ought to be there either.”

“What do you think it should look like?”

“Maybe we should rename Utilities to be Gateways. Maybe it should just have static variables holding all the gateway objects. And maybe people can make all their calls through the gateways.” I said.

“Interesting idea.” Jerry said.

“Hi guys.”

It was Avery. Jerry and I quickly turned at the sound of his voice. Both Avery and Jean were standing there waiting to talk with us. I could faintly hear *Turn Off This Force Field* coming from the open door of the break room.

To be continued...

The source code for this article can be found at:

`www.objectmentor.com/resources/articles/CraftsmanCode/Craftsman_31_DosageTrackingSystem.zip`

The Craftsman: 32

Dosage Tracking IX

Indubitably!

Robert C. Martin
16 November 2004

...Continued from last month.

In August of 1939, Hitler and Stalin signed a non-aggression pact. Though the pact between such ideological enemies was very unpopular with European and American Communists, the practical result was that by the end of 1940 the USSR had seized the eastern half of Poland and annexed the Baltic States; vastly expanding it's empire and creating a long shared border with Germany.

Over the years Stalin had deeply insinuated an espionage network within the highest levels of the United States government. One member of this network, Whittaker Chambers, was so outraged by the growing "friendliness" between Stalin and Hitler that he defected to the U.S. in 1938. Chambers provided the Roosevelt administration with details on dozens of other communist spies. One of these spies, Alger Hiss, was a top State Department official and a trusted friend of FDR. FDR refused to believe the accusations against Hiss and denounced Chambers instead – allowing Stalin's network to continue operating through 1940.

So Stalin knew all about Nimbus, including the Atomic Bomb research and Von Braun's rocketry successes. Stalin also knew he could not defend against the growing US threat alone. So in the early months of 1941 he negotiated a deep alliance with Hitler and the Axis powers, offering some of the Nimbus intelligence as part of the bargain.

The US had a monopoly on the great minds, but their secrets had been sold, and they had lost the element of surprise. Now a great Eurasian power was rapidly growing, and aligning against them.

21 Feb 2002, 1430

Jean didn't give us a minute to respond. "Avery and I have had a nice little chat, and I think we've got everything worked out. Don't you agree Avery? I know you're a fine young man, just a bit high spirited really. You wouldn't make such a fuss if you didn't *care*, would you? Of course you wouldn't. Anyway, Jerry and Alphonse, I'll leave Avery in your fine care and go on about the rest of my day. My goodness, but don't I have a lot of little details to take care of! It never ends, boys, it never ends." And she picked up her knitting basket and shambled off to chat with Carole.

Avery just stood there looking sheepish while Jerry and I put our hands in our pockets and looked at the floor. Finally Jerry said: "I'm glad you're out of there Avery. It's no fun in there – I know. Look, I have to go work with Jasper for awhile. Alphonse, can you and Avery finish up this test page?"

He was going to leave me alone with Avery after the woodshed! What would I say to him? "Sure, Jerry, we can work on it."

"Great, thanks. I'll be back in about an hour or so." And Jerry walked away.

“You were in there a long time.” I said to Avery. “Was it bad? What did she say? Did she yell at you? I felt awful watching you in there.”

“Yeah, well, Jerry’s right about it not being fun. She didn’t actually yell, but... Look, I probably shouldn’t talk all that much about it. She made me realize a few things about myself that I don’t like. I’m going to think about them for awhile. Perhaps I’ll be able to tell you more later, but now I’d just like to get back to work; if you don’t mind.”

“Oh, yeah, sure. No problem. I’ll show you what we’ve been doing.”

“...and Alphonse, it means a lot that you were concerned about me. Thanks.”

“Er, sure... uh, so here’s the thing. Jerry and I have been working on the Register Normal Suit page. We just got the first failing table to pass. You and I should get the second failing table to pass.”

I pointed to the page and Avery studied it.

Normal suit registration.

Import
dtrack.fixtures

We assume that today is 2/21/2002.

DTrack Context
Today's date
2/21/2002

We also assume that there are no suits in inventory.

Suit inventory parameters
Number of suits?
0

We register suit 314159.

Suit Registration Request
bar code
314159

DTrack sends the registration confirmation to Manufacturing.

Message sent to manufacturing		
message id?	message argument?	message sender?
Suit Registration <i>expected</i>	314159 <i>expected</i>	Outside Maintenance <i>expected</i>
null <i>actual</i>	0 <i>actual</i>	null <i>actual</i>

Manufacturing accepts the confirmation.

Message received from manufacturing			
message id	message argument	message sender	message recipient
Suit Registration Accepted	314159	Manufacturing	Outside Maintenance

And now the suit is in inventory, and is scheduled for immediate inspection

Suits in inventory	
bar code?	next inspection date?
314159 <i>missing</i>	2/21/2002

“OK, so we want to get that Message sent to manufacturing table working?”

“Right.” I said. We want to make sure that when we register the suit in the Suit Registration Request table, a message gets sent to manufacturing with the registration information.”

OK, well it looks real easy to get that to pass. All we have to do is modify the `getLastMessageToManufacturing()` method as follows:

```
public class Utilities {  
    ...  
    public static Object getLastMessageToManufacturing() {  
        SuitRegistrationMessage message = new SuitRegistrationMessage();  
        message.sender = "Outside Maintenance";  
        message.id = "Suit Registration";  
        message.argument = 314159;  
        return message;  
    }  
    ...  
}
```

Sure enough the table turned green.

Message sent to manufacturing	message id?	message argument?	message sender?
Suit Registration	314159		Outside Maintenance

“Yeah, that’s what I thought too. But Jerry told me not to do that. He said that it was OK to do the simplest thing to get a *unit* test to pass; but not an acceptance test. He said that if you are tempted to do something too simple to get an acceptance test to pass, you should write some unit tests instead.”

Avery looked confused. “What kind of unit test should we write?”

“Well, we want to make sure that we send a message to manufacturing when a suit is registered.”

“Yeah, but that’s what the acceptance test checks.”

He had a point. “You have a point.”

“Yeah, do we want to write a unit test that checks the exact same thing as the acceptance test?”

He had another point. “You have another point.”

“So what do we do?” Avery was clearly trying to fight his incredulity. If he burst out again so soon after the woodshed, Jean might just leave him in there permanently.

“Look, writing a unit test is no big deal. If it happens to overlap with the acceptance test, then so be it. Let’s just write the unit test and then show Jerry when he gets back.”

“OK, whatever you say Alphonse, but I’ll write it if you don’t mind.”

“Sure, go ahead.”

So Avery grabbed the keyboard and wrote:

```
public class UtilitiesTest extends TestCase {  
    ...  
    public void testRegisterSuitSendsMessageToMfg() throws Exception {  
        Utilities.registerSuit(7734);  
        SuitRegistrationMessage message =  
            (SuitRegistrationMessage) Utilities.getLastMessageToManufacturing();  
        assertEquals("Suit Registration", message.id);  
        assertEquals("Outside Maintenance", message.sender);  
        assertEquals(7734, message.argument);  
    }  
}
```

This failed for the following reason:

expected:<7734> but was:<314159>

“OK, now if we follow Jerry’s rule and make this fail the simplest way possible we have to change `getLastMessageFromManufacturing` again.”

```
public static Object getLastMessageToManufacturing() {
    SuitRegistrationMessage message = new SuitRegistrationMessage();
    message.sender = "Outside Maintenance";
    message.id = "Suit Registration";
    message.argument = 7734;
    return message;
}
```

I clicked the unit test and the acceptance test. “OK, now the unit test passes, but the acceptance test fails.”

“Hmmm.” said Avery.

“Yes!” I responded.

“My point exactly!” said Avery.

“Indeed!” I responded.

We looked at each other and laughed for a second.

“OK, I think I’m seeing Jerry’s logic.” Avery said. “To get both tests to pass, we have to do something intelligent. We can’t just keep doing the simplest thing.”

“Do you have something in mind?”

“Yeah, watch this.” And Avery started typing again.

```
public class Utilities {
    ...
    private static Manufacturing manufacturing = new MockManufacturing();
    ...
    public static void registerSuit(int barCode) {
        manufacturing.registerSuit(barCode);
    }

    public static Object getLastMessageToManufacturing() {
        SuitRegistrationMessage message =
            (SuitRegistrationMessage) manufacturing.getLastMessage();
        return message;
    }
    ...
}
```

```
package dtrack.external;
```

```
public interface Manufacturing {
    public void registerSuit(int barCode);
    public Object getLastMessage();
}
```

```
package dtrack.mocks;
```

```
import dtrack.external.Manufacturing;
import dtrack.messages.SuitRegistrationMessage;
```

```
public class MockManufacturing implements Manufacturing {
    private Object lastMessage;

    public void registerSuit(int barCode) {
        SuitRegistrationMessage msg = new SuitRegistrationMessage();
        msg.id = "Suit Registration";
    }
}
```

```
        msg.sender = "Outside Maintenance";
        msg.argument = barCode;
        lastMessage = msg;
    }

    public Object getLastMessage() {
        return lastMessage;
    }
}
```

“Wow.” I said in admiration as I pushed the test buttons. The unit tests pass now, and so does the Message to manufacturing table.”

“Yeah.” Said Avery. But his brow was furrowed. “I don’t like this. The Mock object shouldn’t be building the message. That should be done by the real Manufacturing object. I also don’t like that argument variable in the message, or the fact that this Utilities class is turning in to one big hodge podge. This code is a mess. It really needs to be cleaned up.”

“Yeah, I agree. I said the same thing to Jerry just before you came back.”

“Should we clean it up before he gets back here?”

I was a little worried about that. I didn’t want to get Jerry mad by changing things that he wasn’t ready to change. On the other hand, the code *was* a mess, and if we didn’t clean it he’d be just as likely to be mad about that.

“OK.” I said. “Let’s see how much damage we can do before Jerry gets back here!”

“Indeed!”

“Indubitably!”

To be continued...

The source code for this article can be found at:

www.objectmentor.com/resources/articles/CraftsmanCode/Craftsman_32_DosageTrackingSystem.zip

The Craftsman: 33 Dosage Tracking X Cleanup on Aisle 10.

Robert C. Martin
4 December 2004

...Continued from last month.

The Heisenberg Uncertainty.

Werner Heisenberg loved Germany, but despised the Nazis. In 1939 he joined Neils Bohr's cell of the Contingent. It was gut-wrenching decision for him to leave Germany for the United States; and unfortunately it took him too long to make the decision.

The SS had never really stopped watching Heisenberg after the "White Jew" incident. The SS had also notice that the behavior of many German scientists had changed. When Von Braun went missing, the SS heightened their attention on other scientists. To their horror they found that Von Braun wasn't the only one missing. But they found Heisenberg at a bus stop.

The exodus from Germany had been carefully organized and synchronized. But Heisenberg's patriotic dilemma made him miss a beat. He was supposed to be on a particular bus, but found he could not make himself get aboard. He watched it go by. As he stood there at the bus stop, steeling himself to get on the next bus, the blackshirts found him.

Under the ministrations of experts – and the SS had experts – a man will reveal information without ever saying a word. Body-language reactions to well-timed pictures, or statements, will reveal what the man's tongue holds back. Heisenberg never uttered a word. He resisted with a will. One of his torturers was later heard to say: "The man fought well. His heart was in it."

He may have fought well, but what he revealed indirectly was enough. When Stalin later approached Hitler to extend their alliance, and offered the intelligence he had gathered from Nimbus as an inducement, Hitler could appreciate both the accuracy and value of that information.

In early 1941 the Quadripartite Alliance (The Axis) between Germany, Japan, Italy, and the USSR was formed; and the doom of the Eastern Hemisphere was sealed.

21 Feb 2002, 1500

"So," I said intelligently, "let's begin with that mock object. You're right, it shouldn't be building the message." I pulled up the code. It looked like this:

```
public class MockManufacturing implements Manufacturing {  
    ...  
    public void registerSuit(int barCode) {  
        SuitRegistrationMessage msg = new SuitRegistrationMessage();  
        msg.id = "Suit Registration";  
        msg.sender = "Outside Maintenance";  
        msg.argument = barCode;  
        lastMessage = msg;  
    }  
}
```

```

    }
    ...
}

```

“First of all”, I continued, “the id should be set in the `SuitRegistrationMessage` constructor.”

“Agreed” said Avery. And he started to type.

```

public class SuitRegistrationMessage {
    public String id;
    public int argument;
    public String sender;
    final static String ID = "Suit Registration";

    public SuitRegistrationMessage() {
        id = ID;
    }
}

```

“OK, the tests still pass.” Avery said. “Now let’s look at the sender field. That shouldn’t be set by the mock. Setting that field is a function that the production code should do.”

“Agreed.” I replied. “But, what part of the production code should do it?”

“Uh....Hmmm.”

“Yeah. What are we really trying to accomplish here?”

“Well, the acceptance test is making sure that we built and sent the appropriate message to Manufacturing.”

“Right!” Avery said. “So what we need to do is build and send the message in *production* code.”

“OK, but we can’t really send it to Manufacturing. We’re just testing.”

“Right, so we create a Mock that overrides just the part that does the sending.”

I saw the light. “Ah, OK, you mean Manufacturing is an abstract class that has a `send` method, and our mock overrides that method.”

“Yeah, I think so.”

“OK, so lets change `MockManufacturing` so that it’s in the right form, and then we can move the methods to a base class.” I began to type.

```

public class MockManufacturing implements Manufacturing {
    private Object lastMessage;

    public void registerSuit(int barCode) {
        SuitRegistrationMessage msg = new SuitRegistrationMessage();
        msg.sender = "Outside Maintenance";
        msg.argument = barCode;
        send(msg);
    }

    private void send(SuitRegistrationMessage msg) {
        lastMessage = msg;
    }

    public Object getLastMessage() {
        return lastMessage;
    }
}

```

“OK, the tests still pass. Now let’s move the `registerSuit` method up into a base class.”

“What should the name of that class be?” Avery asked.

“Manufacturing, of course.”

Avery shook his head and pointed to the screen.

“Oh!” I said, “There’s already a Manufacturing interface.” OK, let’s just make it a class and *then* move the registerSuit method up into it”.

Avery grabbed the keyboard and started to type.

```
public abstract class Manufacturing {
    public void registerSuit(int barCode) {
        SuitRegistrationMessage msg = new SuitRegistrationMessage();
        msg.sender = "Outside Maintenance";
        msg.argument = barCode;
        send(msg);
    }

    public abstract Object getLastMessage();
    protected abstract void send(SuitRegistrationMessage msg);
}
```

```
public class MockManufacturing extends Manufacturing {
    private Object lastMessage;

    protected void send(SuitRegistrationMessage msg) {
        lastMessage = msg;
    }

    public Object getLastMessage() {
        return lastMessage;
    }
}
```

“That’s much better.” I said, as I watched the tests pass. “But I don’t like that getLastMessage method in Manufacturing. It seems to me that’s just a methods for the tests to use and should only be in MockManufacturing.”

“I think I agree with you.” Avery said. So he removed the getLastMessage method from the Manufacturing class.

“Ack! Now it won’t compile.”

“Yeah, that Utilities class depends on it.” I said.

```
public class Utilities {
    ...

    public static Object getLastMessageToManufacturing() {
        SuitRegistrationMessage message =
            (SuitRegistrationMessage) manufacturing.getLastMessage();
        return message;
    }
    ...
}
```

“And who call that method.” Avery asked.

I grabbed the keyboard and did a where-used search. “Just the UtilitiesTest unit-test class, and our MessageSentToManufacturing fixture. We can fix it, just by casting the manufacturing variable to a MockManufacturing.” So I typed:

```
public static Object getLastMessageToManufacturing() {
```

```

        SuitRegistrationMessage message = (SuitRegistrationMessage)
            ((MockManufacturing)manufacturing).getLastMessage();
        return message;
    }

```

“OK, that’s really ugly.” I said, as I watched the tests pass.

“Yeah, let’s get that method out of there. We should be able to have both the fixture and the unit test call the manufacturing object directly. Since they both know that the manufacturing object is a MockManufacturing, we shouldn’t need the cast.”

Avery grabbed the keyboard and changed the unit test first.

```

public class UtilitiesTest extends TestCase {
    ...
    public void testRegisterSuitSendsMessageToMfg() throws Exception {
        MockManufacturing mfg = new MockManufacturing();
        mfg.registerSuit(7734);
        SuitRegistrationMessage message =
            (SuitRegistrationMessage) mfg.getLastMessage();
        assertEquals("Suit Registration", message.id);
        assertEquals("Outside Maintenance", message.sender);
        assertEquals(7734, message.argument);
    }
}

```

“OK, that still works.” I said. “But it doesn’t really make sense to keep it in UtilitiesTest does it?”

“No, clearly not.” And Avery kept typing. He moved the testRegisterSuitSendsMessageToMfg method to a new class named ManufacturingTest. All the tests passed.

Avery was on a roll. “Next, we want to change the fixture.” He kept typing.

```

public class Utilities {
    ...
    public static MockManufacturing manufacturing = new MockManufacturing();
    ...
}

```

```

public class MessageSentToManufacturing extends ColumnFixture {
    ...
    public void execute() throws Exception {
        message =
            (SuitRegistrationMessage)Utilities.manufacturing.getLastMessage();
    }
    ...
}

```

“The tests still all pass.” He said. “So we should be able to get rid of that getLastMessageToManufacturing method.” He did a quick where-used to prove that nobody called it, and then deleted it.

“We should be able to get rid of Utilities.registerSuit the same way!” I exclaimed, and I grabbed the keyboard. I removed the offensive method, and changed the SuitRegistrationRequest fixture to call registerSuit through Utilities.manufacturing. The tests all still passed.

The Utilities class now looked like this.

```

public class Utilities {
    public static Date testDate = null;
    private static SuitGateway suitGateway = new InMemorySuitGateway();
    public static MockManufacturing manufacturing = new MockManufacturing();
}

```

```

public static Date getDate() {
    return testDate != null ? testDate : new Date();
}

public static int getNumberOfSuitsInInventory() {
    return suitGateway.getNumberOfSuits();
}

public static void acceptMessageFromManufacturing(Object message) {}

public static Suit[] getSuitsInInventory() {
    return new Suit[0];
}

public static void addSuit(Suit suit) {
    suitGateway.add(suit);
}
}

```

“Heck,” I said, “we could get rid of `getNumberOfSuitsInInventory`, and `addSuit` the same way!” And I kept frantically typing.

“This is *much* better.” Avery said. “That `Utilities` class is starting to disappear. It really needs to disappear too.”

“Yeah, I agree. But I don’t think we can get rid of any more of it just now. So why don’t we look at that argument variable in the `SuitRegistrationMessage` class.”

But before we could get started Jerry came back.

“Hi guys, how’d it going?”

“Pretty good” we both replied.

“Did you get that test page to pass yet?”

“We’ll...” We looked at each other. “We made some progress on that, but then we decided to refactor the code a bit. It was getting pretty ugly.”

Jerry raised his eyebrows and said: “You...what?”

“Well, we got rid of most of the `Utilities` class, and...

“You....refactored? Let me take a look.

We showed Jerry our changes. He nodded at each one. Then he said, “Nicely done guys; nicely done. This code is in much better shape than when I left it an hour ago. I think this’ll help us make much faster progress now.”

Avery and I looked at each other and smiled. I felt pretty good about the cleanup we’d done. It was good to know that Jerry thought so too.

“OK, guys, I’m going to go talk to Carole for a bit. Why don’t you get this page to finally pass.”

To be continued...

The source code for this article can be found at:

www.objectmentor.com/resources/articles/CraftsmanCode/Craftsman_32_DosageTrackingSystem.zip

The Craftsman: 34 Dosage Tracking XI That's Final.

Robert C. Martin
19 January 2005

...Continued from last month.

The Beaches.

Without an eastern front to worry about, Hitler could focus his attention on Britain. He reactivated operation sea-lion; and this time he did not waver. The Luftwaffe paid a high price, but by sheer weight of numbers Goering finally reduced the RAF to tatters. The Royal Navy also forced the Germans to pay dearly for their gains; but in the end fared no better. By the late spring of '42, invasion seemed inevitable.

The United States longed to help, but was in political turmoil. The growing "friendship" between Stalin and Hitler, and the formation of the Axis powers, caused FDR to think again about the accusations against Alger Hiss. He authorized an investigation into the possibility that the Soviets had spies in US government. This investigation yielded terrifying results.

So as the German invasion barges crossed the channel from Le Havre to Brighton, the US was caught in the midst of a deep political purge from which it could not afford to be distracted. The UK was left to face the enemy alone.

They fought on the beaches. They fought on the landing grounds. They fought in the fields and in the streets, and in the hills. But in the summer of '42, after Churchill was killed by a well-aimed artillery shell, Britain finally fell to the Third Reich.

21 Feb 2002, 1600

Avery and I gave each other a conspiratorial glance.

"Let's run the test and see what's left to do." I said.

"Roger that." said Avery as he pushed the test button on the FitNesse page. This is what we saw:

Normal suit registration.

```
Import
dtrack.fixtures
```

We assume that today is 2/21/2002.

```
DTrack Context
Today's date
2/21/2002
```

We also assume that there are no suits in inventory.

Suit inventory parameters
Number of suits?
0

We register suit 314159.

Suit Registration Request
bar code
314159

DTrack sends the registration confirmation to Manufacturing.

Message sent to manufacturing		
message id?	message argument?	message sender?
Suit Registration	314159	Outside Maintenance

Manufacturing accepts the confirmation.

Message received from manufacturing			
message id	message argument	message sender	message recipient
Suit Registration Accepted	314159	Manufacturing	Outside Maintenance

And now the suit is in inventory, and is scheduled for immediate inspection

Suits in inventory	
bar code?	next inspection date?
314159 missing	2/21/2002

“OK”, I said, “it’s the last two tables that matter. The `MessageReceivedFromManufacturing` fixture needs to call something that puts the confirmed suit into the inventory; and then the `SuitsInInventory` fixture needs to read out the entire inventory.”

“Right.” Avery replied. “And suit number 314159 should be the only suit in there.”

“Yah. So what does that `MessageReceivedFromManufacturing` fixture look like?”

Avery pulled it up on the screen.

```
public class MessageReceivedFromManufacturing extends ColumnFixture {
    public String messageId;
    public int messageArgument;
    public String messageSender;
    public String messageRecipient;
    public void execute() {
        SuitRegistrationAccepted message =
            new SuitRegistrationAccepted(messageId,
                                         messageArgument,
                                         messageSender,
                                         messageRecipient);
        Utilities.acceptMessageFromManufacturing(message);
    }
}
```

“Ew, yuk, it’s another one of those `Utilities` methods that we need to get rid of” complained Avery.

“Yeah, but let’s not get rid of it until we make it pass. Can you pull up the `Utilities` class?”

```
public class Utilities {
    ...
    public static SuitGateway suitGateway = new InMemorySuitGateway();
}
```

```

...
public static void acceptMessageFromManufacturing(Object message) {}
...
}

```

“Well! That explains why the suit is missing. The `acceptMessageFromManufacturing` method doesn’t do anything.”

“So, do you think we should write a unit test for adding a suit to inventory?” Avery asked.

“I think we already have one.” I said, as I grabbed the keyboard and brought up one of the unit tests.

```

public void testOneSuitInInventory() throws Exception {
    Utilities.suitGateway.add(new Suit(1, new Date()));
    assertEquals(1, Utilities.suitGateway.getNumberOfSuits());
}

```

“Oh yeah, I forgot about that.” Said Avery. “So now all we have to do is call that `add` method in `suitGateway`.”

“That’s the way I see it too.” And so I typed the following.

```

public static void acceptMessageFromManufacturing(Object message) {
    SuitRegistrationAccepted suitAck = (SuitRegistrationAccepted) message;
    Suit acceptedSuit = new Suit(suitAck.id, getDate());
    suitGateway.add(acceptedSuit);
}

```

“Whoops, that doesn’t compile!” I said. “The `id` field must not be public.” So I made the appropriate changes.

```

public class SuitRegistrationAccepted {
    public String id;
    public int argument;
    public String sender;
    public String recipient;

    public SuitRegistrationAccepted(
        String id, int argument, String sender, String recipient) {
        this.id = id;
        this.argument = argument;
        this.sender = sender;
        this.recipient = recipient;
    }
}

```

“OK, now it compiles, and the unit tests still pass.” I said. “Now let’s see what the FitNesse test is doing.” And I hit the test button on the FitNesse page; but there was no change.

I was so focused on why the test results hadn’t changed, that I forgot about Avery and started mumbling to myself. “Oh! I forgot to change the `SuitsInInventory` fixture.” I started scrolling around in the screen without comment.

“Hay, slow down!” Avery said. “Remember me? I’m not real happy about making those variables public.”

I found this annoying. We’d already discussed the difference between classes and data structures, and I didn’t feel like having the debate again. I just wanted to get this FitNesse test to pass. So I said “Yeah.” and just kept right on looking at `SuitsInInventory`.

```

public class SuitsInInventory extends RowFixture {
    public Object[] query() throws Exception {

```

```

        return Utilities.getSuitsInInventory();
    }

    public Class getTargetClass() {
        return Suit.class;
    }
}

```

“Oh drat! It’s another one of those `Utilities` functions.” I mumbled.

Avery didn’t get the hint. “I mean, shouldn’t those variables be private?”

“Maybe.” I said, and I brought up the `Utilities.getSuitsInInventory()` method.

```

public static Suit[] getSuitsInInventory() {
    return new Suit[0];
}

```

“Oh, no wonder!” I subvocalized. “It’s returning a dummy array.” I started to change it; but the keyboard was suddenly yanked out from under my fingers. “Hay!”

“Hay, yourself!” said Avery brandishing the keyboard. “I’m part of this too. You can’t just bulldoze your way past me. I don’t like those public variables.”

He was right, of course. I felt dumb. “Sorry.” I said. “I just want to figure this out.”

“Me too, but we have to work together, don’t we?”

“Right, we do.” I shook my head to clear it. “OK, the public variables, why don’t you like them?”

“There are a lot of reasons that I don’t like public variables. Frankly, I’m still reeling from this morning’s conversation with Jerry and Jasmine about encapsulation.”

“OK, but they made sense, right? I mean there really is a difference between a data structure and an object.”

“Yeah, I can sort of see that; and I accept that `SuitRegistrationAccepted` is a data structure and not an object.”

“So what’s your beef?”

“Well, look at it. This class represents a message. The variables are loaded by the constructor. Once loaded, nobody should have the right to change them. Changing those variables would be like somebody intercepting one of my emails, and changing it before it was delivered.”

“Yeah, OK, so we won’t change them. Why do we have to make the variables private?”

“Don’t you think it would express our intent better if the variables were private, or at least protected?”

“I see what you mean. By making them public we imply that they are changeable; by making them private we are telling other programmers that they shouldn’t change them.”

“Right, we’d supply a public accessor method like `getArgument()`, and everyone would know that those variables shouldn’t be changed.

“Hmmm. I think there’s another way to express your intent. Er, may I have the keyboard to show you?”

“Man, what a keyboard hog!” But Avery smiled as he handed the keyboard back to me. I typed:

```

public class SuitRegistrationAccepted {
    public final String id;
    public final int argument;
    public final String sender;
    public final String recipient;
    ...
}

```

Avery looked at the screen for a few seconds and then said: “I hadn’t thought of it that way, but you’re

right; this is better. It keeps the variables accessible, and yet no one can change them.”

“Yeah, this is better than making them private and adding accessor methods. Now can we make this test pass?”

Avery grabbed the keyboard and said: “Sure, but I’m going to do the typing for awhile.”

I rolled my eyes, but yielded.

“So,” said Avery, “we need to return an array of `Suit` objects from the `getSuitsInInventory` method. That should be pretty easy.” And Avery typed the following:

```
public static Suit[] getSuitsInInventory() {  
    return suitGateway.getArrayOfSuits();  
}
```

```
public interface SuitGateway {  
    public void add(Suit suit);  
    int getNumberOfSuits();  
    Suit[] getArrayOfSuits();  
}
```

```
public class InMemorySuitGateway implements SuitGateway {  
    private Map suits = new HashMap();  
    ...  
    public Suit[] getArrayOfSuits() {  
        return (Suit[]) suits.values().toArray(new Suit[0]);  
    }  
}
```

And with that change, the unit tests, and the FitNesse page all passed.

“Pretty easy.” I said.

“Yeah, but we’ve really got to get rid of that `Utilities` class.” replied Avery. “Let’s see if we can do that before Jerry tells us to quit for the day.”

To be continued...

The source code for this article can be found at:

www.objectmentor.com/resources/articles/CraftsmanCode/Craftsman_34_DosageTrackingSystem.zip

The Craftsman: 35

Dosage Tracking XII

What a day.

Robert C. Martin
22 February 2005

...Continued from last month.

Surprise, surprise!

In the early Summer of 1942, while britons were hopelessly battling Hitler's armies in the south and east of England, and the United States was frantically routing out the soviet spies from its upper echelons, the Empire of Japan struck at Pearl Harbor with a massive surprise attack.

Admiral Yamamoto had wanted to strike six months earlier, but the negotiations with Hitler and Stalin had caused delays. Even so, the U.S. was in such political turmoil that he felt confident he was about to eliminate the American threat and then run wild in the Pacific.

But that's not the way things happened. Not at all.

At 0740 on June 6th, Saburo Sakei was flying his zero south towards Oahu in a group of two dozen fighters and torpedo bombers. He kept a constant vigil over the sky and sea, looking left and right, up and down, hunting for planes and ships that could threaten him. When he was just 8 miles from Pearl he saw one of the zeros in his group suddenly explode without warning. He instinctively yanked on his stick to break out of formation, and a half second later saw another companion blow up.

Later, he told his captors that it was like being in a pop-corn popper. Plane after plane exploded before the pilots could even begin to take evasive action. Those few that managed to break out of formation fared little better. Sakei could just see the white vapor trails ripping northwards into the flight of zeros and following (!) those who managed to break formation.

Of course he didn't see the one that was following him.

21 Feb 2002, 1630

"Yeah, but we've really got to get rid of that `Utilities` class." replied Avery. "Let's see if we can do that before Jerry tells us to quit for the day."

"That sounds like a plan." I replied.

"Indeed!"

I pulled up the `Utilities` class on the screen.

```
public class Utilities {
    public static Date testDate = null;
    public static SuitGateway suitGateway = new InMemorySuitGateway();
    public static MockManufacturing manufacturing = new MockManufacturing();

    public static Date getDate() {
        return testDate != null ? testDate : new Date();
    }
}
```

```

    }

    public static void acceptMessageFromManufacturing(Object message) {
        SuitRegistrationAccepted suitAck = (SuitRegistrationAccepted) message;
        Suit acceptedSuit = new Suit(suitAck.argument, getDate());
        suitGateway.add(acceptedSuit);
    }

    public static Suit[] getSuitsInInventory() {
        return suitGateway.getArrayOfSuits();
    }
}

```

We both stared at it for about 30 seconds. I said: “I suggest we get rid of that last method, `getSuitsInInventory`, I’d wager that it’s quite easy to obliterate.”

“I think I might agree with you.” Avery invoked the ‘find usages’ command. The result showed that it was only used in one place: a fixture class named `SuitsInInventory`. He brought it up on the screen.

```

public class SuitsInInventory extends RowFixture {
    public Object[] query() throws Exception {
        return Utilities.getSuitsInInventory();
    }
    ...
}

```

Avery studied the screen for a few seconds and then said: “I think obliteration is imminent.” He clicked on the `getSuitsInInventory` function and invoked the ‘inline’ operation. The call to the function was automatically replaced with its implementation...

```

public class SuitsInInventory extends RowFixture {
    public Object[] query() throws Exception {
        return Utilities.suitGateway.getArrayOfSuits();
    }

    public Class getTargetClass() {
        return Suit.class;
    }
}

```

...and the function was automatically removed from `Utilities`.

We looked at each other with evil grins and we both said: “Obliterated!”

All the unit tests still passed, and the FitNesse page still passed. Nothing was broken.

“On to the next function.” I said.

We both stared at the `acceptMessageFromManufacturing`

“Hmmm.” Said Avery. “I presume you agree that obliteration is not indicated in this case.”

“Indeed.” I said. “I think relocation is a more appropriate option.”

“Quite, quite. But where?”

This was a good question. This function actually had a bit of policy logic in it. This was the function that accepted the acknowledgement of transfer from manufacturing. This was the function that stored the suit in our inventory.

“What do you call something that finalizes a registration?” I asked.

“A secretary? A clerk? Uh, a ... *Registraar!*”

We both looked at each other with a grin and repeated: “A *Registraar!*”

I grabbed the keyboard and clicked on the method. I invoked the ‘move’ operation and typed `dtrack.policy.Registraar`. After confirming that I did indeed want to create the `policy` package

and the `Registraar` class, the method was automatically moved.

```
public class Registraar {
    public static void acceptMessageFromManufacturing(Object message) {
        SuitRegistrationAccepted suitAck = (SuitRegistrationAccepted) message;
        Suit acceptedSuit = new Suit(suitAck.argument, Utilities.getDate());
        Utilities.suitGateway.add(acceptedSuit);
    }
}
```

“Relocated!” we both shouted.

Now the `Utilities` class was looking very anemic.

```
public class Utilities {
    public static Date testDate = null;
    public static SuitGateway suitGateway = new InMemorySuitGateway();
    public static MockManufacturing manufacturing = new MockManufacturing();

    public static Date getDate() {
        return testDate != null ? testDate : new Date();
    }
}
```

“Death is coming to this class.” I said with a sneer.

“Without a doubt. And I think I see our next objective. What would you say to removing that `suiteGateway` variable?”

“I think I’d enjoy it very much.” I replied.

Avery grabbed the keyboard, clicked on the doomed variable, and invoked the ‘move’ operation. He selected the `SuiteGateway` class as the target. And the variable was automatically moved.

```
public interface SuitGateway {
    SuitGateway suitGateway = new InMemorySuitGateway();

    public void add(Suit suit);
    int getNumberOfSuits();
    Suit[] getArrayOfSuits();
}
```

“Interesting.” I said. “The initialization moved too. I’m not at all certain that I like that. It seems to me that `SuitGateway` should not know about its derivatives.”

“I’m afraid I must agree with you, Alphonse. The initialization needs to go somewhere else. I wonder where?”

This was taking some thought, and we were beginning to slip out of our formal banter.

“We need some kind of initialization functions somewhere... But who would call it?”

“Yeah, I don’t know. Let’s put a *todo* next to this one and ask Jerry about it when he comes back. I don’t want this problem to derail us from obliterating the `Utilities` class.”

“Agreed! Obliteration is our objective!”

And Avery annotated the initialization with a *todo* comment.

```
public interface SuitGateway {
    // todo move initialization somewhere else.
    SuitGateway suitGateway = new InMemorySuitGateway();

    public void add(Suit suit);
    int getNumberOfSuits();
    Suit[] getArrayOfSuits();
}
```

```
}
```

“OK, now let’s find out who uses that variable.” I said. I grabbed the keyboard and did a ‘find usages’ on the variable. There were 6 usages. They all looked about like this:

```
SuitGateway.suitGateway.getNumberOfSuits()
```

“I’d say that’s somewhat redundant.” I bantered.

“I’d agree. I propose we change the name of the variable.” I clicked on the variable and invoked the ‘rename’ command, changing the variable’s name to `instance`. Now all the usages looked about like this:

```
SuitGateway.instance.getNumberOfSuits()
```

Avery looked at the screen for a moment and said: “That’s certainly an improvement. But I think we can do better.” Avery used the ‘rename’ function to change the name of `SuitGateway` to `ISuitGateway`. Next he moved the `instance` variable to a new class named `SuitGateway`. Next he found all the usages of the `instance` variable and altered the first by eliminating the variable from the expression as shown below.

```
SuitGateway.getNumberOfSuits()
```

This caused the statement to turn red, showing that it would not compile. Avery clicked on the red lightbulb next to that line of code and selected the ‘create method `getNumberOfSuits`’ option. This automatically created the method in the `SuiteGateway` class. Finally, Avery modified that method to delegate through the `instance` variable. All the tests still passed.

Avery changed all the other usages of the `instance` variable in similar fashion. Now `SuitGateway` looked like this:

```
public class SuitGateway {
    // todo move initialization somewhere else.
    public static final ISuitGateway instance = new InMemorySuitGateway();

    public static int getNumberOfSuits() {
        return instance.getNumberOfSuits();
    }

    public static Object[] getArrayOfSuits() {
        return instance.getArrayOfSuits();
    }

    public static void add(Suit suit) {
        instance.add(suit);
    }
}
```

“Avery, you are a genius!” I said. I was duly impressed.

“I know.” He said with a smirk. And we both laughed.

“This is really pretty.” I went on. The `SuitGateway` is really simple to use, and yet it’s also polymorphic. We can put any derivative of `ISuitGateway` into that `instance` variable that we want.”

“Yes, but we still have to figure out how to initialize it.”

“Yeah. So what does our `Utilities` class look like now?”

I grabbed the keyboard and brought it up on the screen.

```
public class Utilities {
    public static Date testDate = null;
```

```
public static MockManufacturing manufacturing = new MockManufacturing();

public static Date getDate() {
    return testDate != null ? testDate : new Date();
}
}
```

“Wow!” I said. “Nothing left but the date stuff, and that odd `MockManufacturing` variable.”

“Yeah, and I’ll bet you that we can get rid of that variable the same way we got rid of the `SuitGateway` variable.”

“Yeah! A `Manufacturing` class, holding a reference to an `IManufacturing` object. Neat!”

Just then, Jerry stepped up. “Hay guys, it’s time for dinner.”

“Huh!” Avery and I looked at each other in surprise. Where had the time gone?

Jerry stared at the `Utilities` class. “Gee, it looks like you guys have been busy cleaning things up, eh? Good. Tell me all about it at dinner. Let’s go.”

So we left the lab for the day...and what a day it had been!

To be continued...

The source code for this article can be found at:

www.objectmentor.com/resources/articles/CraftsmanCode/Craftsman_35_DosageTrackingSystem.zip

The Craftsman: 36

Dosage Tracking XIII

The Color Purple

Robert C. Martin
20 March 2005

...Continued from last month.

The readout indicated 0.97. Turing was not prepared for this. His heart fell. His eyes glazed. And, not for the first time, suicide taunted him. He looked around at the huge device that so many of his associates had helped him to build. Flowers, Von Neumann, Eckert, all had played their part. But the machine was his. The Automatic Computing Engine was his brain-child. He designed it. He programmed it. And now... Now he must deal with what it told him.

He felt as if the 30,000 vacuum tubes had betrayed him. As he stared at the number, 0.97, he realized just how much he had hoped that ACE would declare their salvation. But 0.97 was a virtual death sentence.

Two months ago, the Japanese fleet in the Pacific had been routed, in no small part by the work that he and his fellows in the contingent had contributed to project Nimbus. How joyful those days had been! How they had buoyed his hopes. But ACE had computed the orbital elements of Clyde with more precision and speed than had ever before been possible. And the answer it gave left only three chances out of a hundred that humanity might survive. "We are lost," he murmured, "We are lost."

Yet even as Turing turned away from the number that spelled humanity's doom, an artificial sun blossomed over the Nevada desert.

21 Feb 2002, 1700

"Alphonse, can I ask you a question?"

"I think you just did."

Avery gave me a good-natured punch in the arm and said: "I'm being serious."

I ogled him and said: "You? Serious?" Then I notice the look on his face and got serious myself. "Sure, go ahead."

We were on our way from the lab to the general mess hall at the rim. I liked watching the starbow through the floor ports in the mess hall. Jerry, Jean, and the others were walking several yards ahead of us. They would probably go to the Journeyman's lounge for dinner. Jean favored the lower g. Avery kept his voice low. Apparently he didn't want the others to hear this conversation.

"How much do you think we accomplished today?"

"Avery! Not this again! You just got reamed through for this two hours ago."

"I'm not complaining, Alphonse. I'm just concerned. I know that this is the way Jean and Carole want to run the project, and I know that Jerry and Jasmine don't question it anymore. But I have my doubts. I'm willing to set them aside and work with the team, but I want to know your opinion. So how much do you think we got done today?"

"It seems to me that we finished the Register Suit requirement."

“Did we finish it? Do you think that we could really register a suit now?”

I thought about this for a second, as we stepped into the turbo lift. “Well, no. We don’t have the bar code scanners hooked up to the system yet, and we don’t have a real database or anything, and we really weren’t sending messages over sockets. But we did get the logic of the requirement working.”

We rode in silence as the lift hauled us 50 meters from the inner rim on 44 to the outer rim at 60. We could feel our weight increase, as we unconsciously braced ourselves against the mild anti-spinward coriolis force.

Avery didn’t speak again until we were half way between the lift and the mess hall. When he did speak he rushed his words together as though speaking them had released a pent-up frustration. “I think we could have written a lot more code in one day. We hardly wrote anything code at all. That one little requirement should have been coded in less than an hour. We should have gotten five or ten requirements like that coded today.”

Avery’s face was reddening, and his eyes were starting to bug out.

“I agree with you Avery.” I said. “We *could* have gotten a lot more *coded* today. But would it have been *working*? Would it have been *done*? That Suit Registration story is *done*. It’s done in a way that I don’t remember anything from school ever being done before. It’s coded, it executes, its been tested against unit tests and acceptance tests. It’s *done*!”

We turned into the mess hall, picked out some trays, and stood in the cafeteria line. I prepared two hot dogs loaded with mustard, onions, sauerkraut, and hot-peppers! My mouth started to water. Yum! Avery absently made himself a peanut butter and jelly sandwich. We both got a tall glass of lemonade. Then we seated ourselves at a table that had a clear view through one of the floor ports. The starbow shimmered there as always.

Avery took a long pull of his lemonade and said: “OK, you are right, that tiny bit of the Register Suit feature is done. But don’t you think we could have done a lot more today?”

I took a huge bite out of a hotdog. Mustard dribbled out of the bun, and got all over my face. A hot pepper burst in my mouth mixing with the powerful bite of the mustard. Heaven! “Avery, perhaps you... are remembering what it was like to write... projects for school.”

“Yeah, sort of. I mean we got a *lot* done in a short period of time!”

I wiped the mustard from my mouth with the back of my hand. “Perhaps it seemed that way to you. Perhaps your classes were more productive than mine. What I remember was frantic bursts of coding followed by long droughts of debugging, only to deliver a project that was buggy, and had taken lots of long midnight hours to get working. Sure the coding went fast. It was everything else that took so long.”

“Well, you make a good point, but...” There was a dab of grape jelly on his cheek.

“Avery,” I said around another dribbling bite of hotdog, “how much debugging did... we do today? We got Carole’s acceptance... test passing without any debugging at all! It’s done Avery! Tomorrow we’re going to start working on the next requirement. The day after that we’ll work on the next. And Friday we’ll work on yet another. If we keep up this pace, just you and I will get four stories done. Jerry and Jasper will get three of their own done. We’ll have seven stories done! And the week after that we should have ten more done. Gosh, if we knew how many stories Carole was going to write, we could just divide by ten to figure out how many weeks this project is going to take!”

Avery was chewing a big bite of his sandwich, and trying to talk at the same time. The result wasn’t pretty. “Yeah.... OK.... but.... that assumes that all the stories are... the same size. Some are... going to be a lot harder... than that.”

I picked up my second hot dog. Mustard laden sauerkraut splayed around my fingers. “Yeah...” I took a bite. “you’re probably right.... But.... That’s OK because... we can estimate which ones are... bigger and.... so.... we’ll still be able to tell... how many stories we can... get done in a week, and... how many weeks until we can deliver the... project.”

Avery’s sandwich was too full of jelly. A big glob dripped out into his hands as he took his next bite. “You could be right... I mean, maybe working this way...is more predictable. I suppose that Carole... would find that... comforting. But... What about QA?... Won’t that take... a long time?”

I crammed the soakstall – the butt end of my hotdog – into my mouth, licking the mustard and pepper juices from my fingers while chewing and talking. “QA?... Do you think... we’ll need much?... I mean

with all... the testing... we are doing. Don't... you think...

Suddenly I couldn't breathe. The soakstall had gone down the wrong way. I tried to cough it out, but it wouldn't come. Avery saw my distress and started whacking me on the back. Two whacks and the soakstall came flying out of my mouth. I took a deep grateful breath, and then apologized to the people at the next table.

"You'd better be more careful when you eat those." said Avery.

"Yeah, I guess I should. C'mon, I'm still hungry. Let's see if there's any pizza." I started walking towards the cafeteria line again.

Avery followed me. "Hay, Alphonse?"

"Yeah?"

"Did you know that you have two purple handprints on the back of your shirt?"

To be continued...

The Craftsman: 37

Dosage Tracking XIV

Handling Rejection

Robert C. Martin
27 April, 2005

...Continued from last month.

Through the last quarter of 1942 FDR's advisors were pressing him to declare war upon the Axis. The successful detonation of an Atomic bomb in August, and the increasing supply of those terrible weapons and the rockets to deliver them, was emboldening the hawks. And, horrible though it was, it made sense. It seemed unlikely that the U.S. would be unable to hold the technological advantage for long. Eliminating the virulence of fascism might be possible now, but the chance could quickly disappear leaving the U.S. to face a huge and fearsome opponent.

The contingent had waited to tell the President about Turing's results. They had checked, and double checked the results. They had improved the computers, the telescopes, the programs, and the observations. But the results would not be denied. A 22km rock named Clyde was almost certainly going to slam into the Pacific at 53km per second on April 29th, at 0943GMT.

On November 12th, 1942 as FDR read the report from the contingent, the certainty of destruction seemed to leave the pages and clamp down upon his skull like an iron fist. He called out to his secretary: "Grace, I have a terrific headache!" Grace Tully found him collapsed over his desk, never to regain consciousness again.

22 Feb 2002, 0800

I walked into the lab, just as people were gathering for the morning stand-up meeting. Avery, I, Jasper, Jerry, Carole, and Jean stood together in a circle. Each of us answered three questions: What did you do yesterday? What do you plan to do today? What's in your way?

When my turn came I said: "Avery and I got the `RegisterNormalSuit` acceptance test to pass. I don't know what I'm doing today, and nothing but that ignorance is in my way." I saw Carole role her eyes at that remark, and Jerry gave a little smirk.

Then it was Avery's turn, and he simply said: "My report is the same as Alphonse's."

After the standup Jean came over and said: "Boys, I think I can solve your problems. Avery, dear, why don't you work with Jerry? He already has his next acceptance test. Alphonse, I'd like you to work with Jasper today. He'll be putting together the acceptance test for manufacturing rejection. I'm sure you and Jasper will get along wonderfully; you are both such fine young men." She gave us a warm smile and then said "Off with you! Shoo!"

I caught Avery's eye, and gave him a regretful wave, and then headed over to where Jasper was working. He was staring intently at his screen and didn't seem to know I was there. So I said: "Hello Jasper, I guess we're working together today."

Jasper jerked around with a big grin on his face and said "Alphonse! Yeah, great. Hay, should I call

you Al, or Fonse? I kind of like Fonse. Whaddya say?”

His toothy grin, and the sparkle in his eyes, put me off guard. I was about to say that “Fonse” is the kind of nickname that sticks, and that I wasn’t sure if I really wanted it to stick; but he cut me off and said: “Great! Now let’s get busy on this new test.”

I sighed and sat down. “Jean said something about a rejection from Manufacturing?”

“Yeah, that’s right. Here look at the story card.” He handed me the index card from Carole’s planning meeting yesterday morning¹.

Register new suit.

- Bar Code Patch X(6)
- Register new suit, screen function.
- Send confirmation to mfg.
 - o Reject registration on denial.
 - o 10s time out & reject
- If already reg’d
 - o Reject reg & don’t send conf to prod.
- Sched for inspection.

“Oh, yeah.” I said. “I remember now. Carole told us that if someone tries to register a suit that didn’t come from manufacturing, we should reject the registration.”

“Right.” Said Jasper. “We don’t want someone trying to register some old suit they found in a locker somewhere. Eh?”

“Yeah. OK, so we need a new acceptance test for this case, don’t we?”

“Right you are, Fonse, I was just working on that when you came up here.” He pointed to his screen. “I just created a new page named `SuitRegistrationRejectedByManufacturing`. I took the `RegisterNormalSuit` page that you guys got working yesterday, and pasted it into this new page.”

I examined the page, and sure enough it was a perfect copy of the `RegisterNormalSuit` page. “So I guess you want to alter it to reflect that manufacturing rejects the confirmation request?”

“Right again, Fonse. Want to take a crack at it?”

“Uh, sure.” I wasn’t sure if I liked his over-friendly demeanor. He didn’t seem to mean anything by it, but I thought it could get annoying after awhile. I edited the page, changing a few comments and titles. The meat of the difference was to change the message sent my manufacturing to a rejection, and to then assert that the suit did not get placed into inventory. The result, complete with test results, looked like this:

¹ <http://www.sdmagazine.com/documents/s=7764/sdm0407h/>

```
Import
dtrack.fixtures
```

DTrack Context
Today's date
2/21/2002

Suit inventory parameters
Number of suits?
0

Suit Registration Request
bar code
314159

message id?	message argument?	message sender?
Suit Registration	314159	Outside Maintenance

Message received from manufacturing			
message id	message argument	message sender	message recipient
Suit Registration Rejected	314159	Manufacturing	Outside Maintenance

Suits in inventory	
bar code?	next inspection date?
314159 surplus	Thu Feb 21 00:00:00 CST 2002

“I think I can.” I said. I opened up the fixture that handled the “Message received from manufacturing” table. It looked like this:

[illegible]

```

        Registrar.acceptMessageFromManufacturing(message);
    }
}

```

It just passed the table data along to the `acceptMessageFromManufacturing` method of the `Registrar` class. That method looked like this:

```

public class Registrar {
    public static void acceptMessageFromManufacturing(Object message) {
        SuitRegistrationAccepted suitAck = (SuitRegistrationAccepted) message;
        Suit acceptedSuit = new Suit(suitAck.argument, Utilities.getDate());
        SuitGateway.add(acceptedSuit);
    }
}

```

I made the following simple change:

```

public class Registrar {
    public static void acceptMessageFromManufacturing(Object message) {
        SuitRegistrationAccepted suitAck = (SuitRegistrationAccepted) message;
        if (suitAck.id.equals("Suit Registration Accepted")) {
            Suit acceptedSuit = new Suit(suitAck.argument, Utilities.getDate());
            SuitGateway.add(acceptedSuit);
        }
    }
}

```

When I ran the test, it passed.

“Nicely done, Fonce; but you didn’t write a unit test!”

“Do you really think one is necessary for just this `if` statement?”

“How hard would it be to write?”

I shook my head and just typed. It was a simple test to write.

```

public class RegistrarTest extends TestCase {
    public void testAcceptRegistration() throws Exception {
        final String acceptId = "Suit Registration Accepted";
        SuitRegistrationAccepted suitAck =
            new SuitRegistrationAccepted(acceptId, 9999, "me", "you");
        Registrar.acceptMessageFromManufacturing(suitAck);
        Suit[] suits = (Suit[])SuitGateway.getArrayOfSuits();
        assertEquals(1, suits.length);
        assertEquals(9999, suits[0].barCode());
    }
}

```

The test passed on its first try. I felt a little smug. So as I looked over to Jasper and raised an eyebrow. His supercilious grin framed by his sugarbowl haircut made him look clueless.

“You didn’t write the negative case.” He said, still grinning widely.

I thought to myself: “This must be how Avery feels.” I suppressed the desire to roll my eyes, and simply kept typing.

```

    public void testRejectRegistration() throws Exception {
        final String rejectId = "Suit Registration Rejected";
        SuitRegistrationAccepted suitNak =
            new SuitRegistrationAccepted(rejectId, 9999, "me", "you");
        Registrar.acceptMessageFromManufacturing(suitNak);
        Suit[] suits = (Suit[])SuitGateway.getArrayOfSuits();
    }
}

```

```
    assertEquals(0, suits.length);  
}
```

This test also worked the first time. So I raised my eyebrow again.

“Hey Fonse, the name of that class isn’t quite right, is it?”

I did a silent little curse. I had been hoping he wouldn’t notice what I had notice while typing this test. The name `SuitRegistrationAccepted` really wasn’t the right name for this class anymore. I hated to admit it, but this is something that the unit test forced me to see, that the acceptance test completely hid from me. So I changed the name to `SuitRegistrationAcceptanceMessage`.

Jasper tilted his head as though thinking about something, and then cheerily said: “Fonse, do you think those message id’s should really be long strings? Do you think those strings ought to be scattered around the code? I think that’s kind of ugly don’t you?”

Drat, he did it again. That was just what I was thinking. Strings like “Suit Registration Accepted” and “Suit Registration Rejected” probably weren’t good values to use as message ids. Moreover, they probably shouldn’t be scattered around the code. “Yeah, I agree. What do you think we should do about it?”

“I think we should get all the tests to pass, and then refactor those id strings out of the code altogether. Perhaps we can make them independent classes.”

“Uh, wait. The tests *do* all pass.”

“Have you run them all since that last change you made Fonse?”

“Yeah, I...” Actually I had only run the `testRejectRegistration` test. I hadn’t run any of our other unit tests. So I turned back to the screen and pushed the button the ran *all* the existing unit tests in the DTrack system. To my horror, `testRejectRegistration` failed! “Wait! That test just passed!”

“Yeah, I think we’ve got a database cleanup problem.” Smiled Jasper. “Suits are being placed into inventory, and not removed at the start of each test. Try running all the acceptance tests. I’ll bet they fail too.”

Sure enough, as I pushed the *suite* button on the top level of our acceptance tests, the `RegisterNormalSuit` page passed, but the `SuitRegistrationRejectedByManufacturing` page – the page I had just gotten working ten minutes ago, failed.

Jasper smacked his hands together and started rubbing them. “OK Fonsie, my young apprentice, let’s clean this all up.”

The code for this article can be located at:

http://www.objectmentor.com/resources/articles/CraftsmanCode/Craftsman_37_Do_sageTrackingSystem.zip

To be continued...

The Craftsman: 38

Dosage Tracking XV

Test Independence

Robert C. Martin
11 May 2005

...Continued from last month.

President Henry Wallace was sworn into office on the 13th of November, 1942. Wallace was a scientist and a visionary. Before he took office he had dreamed of conquering fascism, and creating a just and lasting peace throughout the world. He felt the Earth was on the verge of “The Century of the Common Man”.

A weaker man might have given up hope upon learning that “the common man” had less than twenty years to live. But Wallace was not cowed. He had read the Contingent’s reports on the use of Atomic Energy to deflect, or at least escape, Clyde. He determined that the resources of the entire world must be brought to bear on this problem, and he was determined to lead the U.S. in making that happen.

Only one thing was in his way... The Eastern Hemisphere.

22 Feb 2002, 0830

“The first thing to do, my dear Fonse, is to get all these tests to pass. Do you know why they are failing?” Jasper’s toothy grin was both congenial and grating.

“I think so.” I said, trying not to appear befuddled. “Apparently the first test to run changes the database in a way that the next test doesn’t expect.”

“You got it, buddy! I’m pretty sure that’s the case. So we need to clear the database between each test case.”

This sounded strange to me, like extra work. “Why don’t we just change the second test to expect what the first test leaves behind? That way we don’t have to clean up. Even better, it means that the tests can build on each other. Each test leaves the database all set up for the next test.” I was proud of myself for thinking this through.

Jasper wagged his finger at me. His thick blond hair, cut off at his ears, jiggled as he shook his head. “Nosirree, Fonse ol’ pal. We don’t let tests depend on each other. We want to be able to run any test at any time. We don’t want to have to run them in sequence.”

In the back of my mind I could see Avery rolling his eyes at this thought. I suppressed that gesture, and simply asked: “Why is that? Wouldn’t it be much easier to run the tests in sequence?”

“It might be a little bit easier *right now*”, Jasper said with sugary emphasis on the last two words, “but it would make it very hard to diagnose problems later on.” Jasper put his arm around my shoulder as he continued his syrupy lecture: “Think about what would happen, Fonse, if one of the tests in your sequence failed? *All* the downstream tests would fail too, wouldn’t they?”

I politely disengaged from his arm and said: “Well, sure, but so what? You’d know what test failed.”

A glimmer of regret passed over Jasper’s face as I backed away from him. “True, but you wouldn’t know if any of the downstream tests would have passed. To find out you’ll have to get that first failing test to pass. If other, later, tests fail, you’ll have to get them all working one by one.”

Jasper paused for a second and then said: “It’d be like compiling a C++ program. Have you ever done that?”

I shook my head.

“Never mind. You can see that it would be inconvenient. It’s better for each test to fail for it’s own reasons, than because a previous test failed. Also, it’s very convenient to be able to run any test at any time. If we put them in a sequence, then we wouldn’t be able run individual tests.”

I sighed. This actually made some sense. I wondered why I didn’t like hearing sense from Jasper. Perhaps it was because the more he talked, the wider his grin got, making it look like his head was split in half.

“OK.” I said, “So we need to clear the database in front of each test case.”

Jasper winked and said: “Attaboy, Fonse!”

This was torture. I didn’t know how much longer I could keep cool. So I faced the keyboard and ran all the unit tests again. Two unit test files were failing: `RegistrarTest`, and `UtilitiesTest`. I opened each, and looked them over. Sure enough, each of the failing test cases made use of the `SuitGateway`, our front end to the database. I opened `SuitGateway` and saw this:

```
public class SuitGateway {
    // todo move initialization somewhere else.
    public static final ISuitGateway instance = new InMemorySuitGateway();

    public static int getNumberOfSuits() {
        return instance.getNumberOfSuits();
    }

    public static Object[] getArrayOfSuits() {
        return instance.getArrayOfSuits();
    }

    public static void add(Suit suit) {
        instance.add(suit);
    }
}
```

I thought to myself that perhaps it was time to follow the advice in that `todo` comment. So I changed the initialization of the `instance` variable to `null`, and ran all the unit tests. Of course I got lots of null pointer exceptions, but only in `RegistrarTest` and `UtilitiesTest`. So I opened `RegistrarTest` and modified it as follows:

```
public class RegistrarTest extends TestCase {
    protected void setUp() throws Exception {
        SuitGateway.instance = new InMemorySuitGateway();
    }

    ...
}
```

The `setUp` function is called in front of every test function, so now each test function will start with an empty database. I pushed the test button and *all* the unit tests now passed, even the ones in `UtilitiesTest`!

“That’s strange.” I said.

Jasper grinned at me like he’d just heard a funny joke. “C’mon, Fonse, use your head there buddy. The last test case in `RegistrarTest` just left the database empty.”

I could feel the heat rise in my face. I hoped the flush wasn’t visible. I took a deep breath and said: “I see what you mean. But I still need to clear the database in `UtilitiesTest`, just to make sure.”

“Oh, absolutely Fonse! I won’t argue with that!”

His Cheshire cat smile hung in space behind my closed eyes. When it faded I made the same change to `UtilitiesTest`, and the unit tests all continued to pass.

Jasper gave me a gentle elbow in the ribs and said: “Nice sailing captain, now let’s fix those acceptance tests.”

Something cold began to rise in me, replacing my flush with ice, or perhaps steel. I took another deep breath, and realized it was the last one I was willing to take. I ran the acceptance tests. Sure enough they all failed for null pointer violations.

The first fixture in both acceptance tests was `DTrackContext`. This fixture was made for initialization. So I made the following changes.

```
public class DTrackContext extends ColumnFixture {
    public Date todaysDate;

    public void execute() throws Exception {
        Utilities.testDate = todaysDate;
        SuitGateway.instance = new InMemorySuitGateway();
    }
}
```

This made sure that each of the two test pages now started with a cleared database. I pushed the suite button, and saw green; all the acceptance tests pass.

I knew it was coming without even looking. I could feel his enthusiastic congratulations before they left his mouth. I waited for it with ice water in my veins, knowing what I had to do.

He patted me on the head. “Nice work Fonse!”

He patted me on the head! I closed my eyes. I was done. There would be *no more* of this. I tightened my stomach and visceral muscles and turned towards him.

“Jasper, my name is *Alphonse*. Please don’t call me Fonse anymore. And *please* stop being so patronizing. I’m not going to be able to work with you if you keep treating me like a pet. I know you are more experienced than I am, but that doesn’t make me a low grade moron who needs constant cuddling. Now, if you’ll excuse me, I’m going to the washroom.”

Had I really just said all that? Had I really been so calm and articulate? As I approached the washroom I could feel the ice-water drain away to be replaced with fear and embarrassment. I stood in the washroom and waited for the shakes to stop. Jean was going to be furious with me. She was going to take me to the woodshed. But I couldn’t go on like that. I had to stop him.

Didn’t I?

The code for this article can be located at:

http://www.objectmentor.com/resources/articles/CraftsmanCode/Craftsman_38_DoSageTrackingSystem.zip

To be continued...

The Craftsman: 39

Dosage Tracking XVI

Test Refactoring

Robert C. Martin
7 June 2005

...Continued from last month.

By hook, and by crook, President Wallace poured money into project Nimbus. He directed the Contingent to develop technologies pursuant to: 1. Saving the human race from Clyde. 2. Defending this effort against the Axis powers.

The latter of these two imperatives enjoyed more short term success than the former. Technologies for weapons systems were created and deployed in absurdly short time frames. A now famous quote by Edward Teller summarized the period: "When you fight for a desperate cause and have good reasons to fight, you usually win."

By August of 1943, Von Braun's rockets were launching cameras into orbit to spy on Europe and Asia. Canisters of film were ejected from these satellites and recovered by aircraft that snagged them from the sky as they fell on their parachutes.

That's how the U.S. learned that the Axis was planning to invade Mexico.

22 Feb 2002, 0830

When I returned from the washroom, Jasper was still sitting at the workstation waiting for me. His smile had been replaced by a hang-dog grimace.

"Alphonse, look." He said with an "aw shucks" tone. "I'm sorry; I guess I was trying too hard. The truth is that I just recently made Journeyman status, and I've never had an apprentice before. Can we start over?"

I had been on the verge of apologizing myself, so I was glad he beat me to it. "Sure, Jasper, we can start over. Just ease off the attitude a bit, OK?"

"I'll try, Alphonse. And if I forget, don't hesitate to remind me. Deal?"

"Deal." I wasn't sure if this would last, but we had a lot of work to get done, and we didn't need personality problems getting in the way. "So where are we?"

Jasper turned to face the screen, and said: "We just got the `SuitRegistrationRejectedBy-Manufacturing` test running. So the next test we should work on is the rejection of a duplicate suit."

"You mean if someone tries to register a suite that's already registered, we should reject it? That makes sense."

"Right. So let me show you this neat new fixture that Rick Mugridge wrote. It's called `DoFixture`, and it really helps writing acceptance tests like this one."

"Rick who?"

"Rick Mugridge. He's an astrologator by training, but has a passion for programming. So lets start by creating the new test page."

Jasper opened the RegisterSuit page and made the following change:

```
|^RegisterNormalSuit|'The Happy Path.'|  
|^SuitRegistrationRejectedByManufacturing|'Manufacturing rejects the  
registration.'|  
|^RejectDuplicateRegistration|'You can't register a suit that's already  
registered.'|
```

Then he clicked on the RejectDuplicateRegistration link and copied the following onto the new page:

```
!path C:\DosageTrackingSystem\classes  
  
!3 DTrack rejects a duplicate suit registration.  
  
!|Import|  
|dtrack.fixtures|  
  
'We assume that today is 2/21/2002.'  
!|DTrack Context|  
|Today's date|  
|2/21/2002|
```

I pointed to the first line on the screen and said: “All three of our test pages has that !path line. Is there a way to put it in one place?”

Jasper winked and said: “Sure beans, Fon...Alphonse. We can just move it to the parent page.” So he made the following change to the RegisterSuit page, and removed the !path line from the three subpages.

```
!path C:\DosageTrackingSystem\classes  
  
|^RegisterNormalSuit|'The Happy Path.'|  
|^SuitRegistrationRejectedByManufacturing|'Manufacturing rejects the  
registration.'|  
|^RejectDuplicateRegistration|'You can't register a suit that's already  
registered.'|
```

Then Jasper typed the following table into the RejectDuplicateRegistration page:

```
|Do Fixture|  
|start|dtrack.fixtures.DTrackFixture| | |
|set suit|314159|as registered|  
|check|register suit|314159|false|  
|check|was a message sent to manufacturing|false|  
|check|count of registered suits is|1|  
|check|error message|Suit 314159 already registered.|true|
```

“OK, that’s different.” I said. “Let’s see if I understand it. First you force suit 314159 into the registration database. Next you try to register 314159, and expect it to fail because it’s a duplicate. You make sure that no message was sent to manufacturing, that the number of suits in the database is still one, and that an appropriate error message was created.”

Jasper’s smile was back. “Right you are Alphonse old sport!”

Maybe Jasper just couldn’t turn his attitude off. I ignored it and asked: “This is pretty simple to understand, but what’s that start line at the top?”

Jasper raised his eyebrows twice and said: “Good catch! That names the class, DTrackFixture, that has all the methods that this fixture will call.”

I didn’t understand this and must have looked puzzled because Jasper said: “I’ll show you. Go ahead and click the test button.” I obeyed and saw:

Could not find fixture: DoFixture.

“Right,” said Jasper. “You’ve got to import the package `fitLibrary`.”

So I added `fitLibrary` to the `Import` table as follows:

```
!!Import|
|dtrack.fixtures|
|fitlibrary|
```

Now when I hit the test button I got a whole bunch of error messages. The first was:

Unknown class: DTrackFixture

Before Jasper could comment I said: “OK, so I need to create a class named `dtrack.fixtures.DTrackFixture`, right?”

“Correctomundo, Al! Uh, Alphonse.”

He was clearly making an effort. I was surprised that it was so hard for him, but I made no comment. I just typed:

```
package dtrack.fixtures;

public class DTrackFixture {
}
```

That made the first error message go away. The next message was:

Unknown: "setSuitAsRegistered" with 1 argument

Again, before Jasper could comment I said: “OK, this must be a method of `DTrackFixture`, right?” I could see him starting to wind up to deliver another syrupy answer so I held up my hands and gave him a meaningful look. He paused, puzzled; then gave me a sardonic smile, and simply said: “Right.” “Progress!” I thought to myself. I typed the method:

```
public class DTrackFixture {
    public void setSuitAsRegistered(int barcode) {
    }
}
```

That made the error message go away. Likewise I was able to make all the other error messages go away one by one. Eventually the fixture looked like this:

```
public class DTrackFixture {
    public void setSuitAsRegistered(int barcode) {
    }

    public boolean registerSuit(int barcode) {
        return true;
    }

    public boolean wasAMessageSentToManufacturing() {
        return true;
    }
}
```

```

public int countOfRegisteredSuitsIs() {
    return 0;
}

public boolean errorMessage(String message) {
    return false;
}
}

```

Now when I hit the test button, there were no more error messages. Instead, I had red cells in the table that looked like this:

Do Fixture			
start	dtrack.fixtures.DTrackFixture		
set suit	314159	as registered	
check	register suit	314159	false expected
			true actual
check	was a message sent to manufacturing	false expected	
		true actual	
check	count of registered suits is	1 expected	
		0 actual	
check	error message	Suit 314159 already registered.	true expected
			false actual

Jasper shook his head as he looked at the screen. “You figured that out pretty fast, Alphonse. Do you think you can wire the fixture up to the application?”

There wasn’t even a *hint* of condescension in his demeanor. I think he was actually impressed instead of pretending to be. “Sure, I think so.”

The first method of the fixture was easy. All I had to do was jam the specified suit into the database:

```

public void setSuitAsRegistered(int barcode) {
    SuitGateway.add(new Suit(barcode, Utilities.getDate()));
}

```

Then I changed `DTrackFixture.registerSuit` to call `Manufacturing.registerSuit`.

```

public boolean registerSuit(int barcode) {
    return Utilities.manufacturing.registerSuit(barcode);
}

```

The `wasAMessageSentOtManufacturing` method simply checked to see if a message was sent.

```

public boolean wasAMessageSentToManufacturing() {
    return Utilities.manufacturing.getLastMessage() != null;
}

```

The `countOfRegisteredSuitsIs` method was also trivial.

```

public int countOfRegisteredSuitsIs() {
    return SuitGateway.getNumberOfSuits();
}

```

But I didn’t know what to make of the `errorMessage` method. So I left it unchanged. When I hit the

test button, I saw:

Do Fixture			
start	dtrack.fixtures.DTrackFixture		
set suit	314159	as registered	
check	register suit	314159	false expected
			true actual
check	was a message sent to manufacturing	false expected	
		true actual	
check	count of registered suits is	1	
check	error message	Suit 314159 already registered.	true expected
			false actual

I looked expectantly at Jasper. He said: “Hot ziggedy dog, Alphonse! -- er -- Sorry, I mean, uh, good. But what about the `errorMessage` method?”

“I don’t know exactly what to do for this method. It looks like you are trying to see if an error message was printed. But I don’t know how to check that.”

“Oh SURE you do, Alphonse! -- er -- I mean, I think we want to make a mock object that represents the screen, and remembers the messages that were sent to it. The `errorMessage` method then simply asks if a particular error message has been sent to the screen.”

Of the two Jaspers, I liked the second one better. His idea made sense. So I said: “OK, so we could create an `IConsole` interface that has a method like `display(String message)`. And then we can create a dummy `Console` that implements that interface and saves any message sent there. That sounds easy.” So, bit by bit I wrote the following test, and the code that made it pass:

```
public class MockConsoleTest extends TestCase {
    private MockConsole c;

    protected void setUp() throws Exception {
        c = new MockConsole();
    }

    public void testEmptyConsole() throws Exception {
        assertEquals(0, c.numberOfMessages());
    }

    public void testOneMessage() throws Exception {
        c.display("message");
        assertEquals(1, c.numberOfMessages());
        assertTrue(c.hasMessage("message"));
        assertFalse(c.hasMessage("noMessage"));
    }

    public void testTwoMessages() throws Exception {
        c.display("messageOne");
        c.display("messageTwo");
        assertEquals(2, c.numberOfMessages());
        assertTrue(c.hasMessage("messageOne"));
        assertTrue(c.hasMessage("messageTwo"));
        assertFalse(c.hasMessage("no message"));
    }
}

public interface IConsole {
    public void display(String message);
}
```

```
public class MockConsole implements IConsole {
    HashSet messages = new HashSet();
    public void display(String message) {
        messages.add(message);
    }

    public int numberOfMessages() {
        return messages.size();
    }

    public boolean hasMessage(String s) {
        return messages.contains(s);
    }
}
```

“OK, that’s good.” said Jasper. “Now let’s see if we can get this acceptance test to pass.”

The code for this article can be located at:

http://www.objectmentor.com/resources/articles/CraftsmanCode/Craftsman_39_DoSageTrackingSystem.zip

To be continued...

The Craftsman: 40

Dosage Tracking XVII

Non-trivial trivialities

Robert C. Martin
21 July 2005

...Continued from last month.

In the winter of 1943, and the spring of 1944 the American spy satellites watched as the axis powers assembled a huge armada on the west coast of Africa. The Americans kept very close tabs on this build-up, and the scouting missions that were related to it. The Axis scouts were paying a lot of attention to the Yucatan peninsula. Their intent was obvious. The Axis powers were planning a major invasion of Mexico; probably in the hopes of establishing a base from which to directly assault the heartland of the U.S.

General MacArthur smiled as he perused the endless flow of satellite photos and intelligence reports. The straightforward actions of the enemy told him that they had no knowledge of eyes that were watching from above. And if they didn't know about the orbiting cameras, then they certainly didn't know that Von Braun's missiles could deliver atomic fire anywhere in the world. MacArthur's smile deepened. The options were endless! He didn't know what he would do next, but he would think of something.

22 Feb 2002, 0900

“Getting this test to pass shouldn't be to hard.” I said. “Let's look at the first failure.” I pulled the test up on the screen.

Do Fixture			
start	dtrack.fixtures.DTrackFixture		
set suit	314159	as registered	
check	register suit	314159	false expected
			true actual
check	was a message sent to manufacturing	false expected	
		true actual	
check	count of registered suits is	1	
check	error message	Suit 314159 already registered.	true expected
			false actual

“OK, the first line to fail is the `check register suit` line. The registration should fail because 314159 is a duplicate suit. So we need to put the duplication check in to the function that the fixture is calling.”

Jasper nodded sagely, and didn't say a word for a change. So I pulled up the `DTrackFixture` to look

at the `registerSuit` method.

```
public boolean registerSuit(int barcode) {  
    return Utilities.manufacturing.registerSuit(barcode);  
}
```

I stared at this function for a few seconds, trying to piece together what it implied. Finally I said: “OK, `Utilities.manufacturing.registerSuit` is the function that sends the message to Manufacturing to validate that the suit is authentic, and that it was approved for outside maintenance.”

“It doesn’t seem to have the right name, does it?” asked Jasper.

“No, it doesn’t. It should probably be called something like `requestApprovalForRegistration`. I’ll make the change.” So I quickly changed the name of the function.

“Let’s follow that function and see if there are any other name changes to make.” suggested Jasper. I agreed so I opened up the `Manufacturing` class.

```
public abstract class Manufacturing {  
    public boolean requestApprovalForRegistration(int barCode) {  
        SuitRegistrationMessage msg = new SuitRegistrationMessage();  
        msg.sender = "Outside Maintenance";  
        msg.argument = barCode;  
        send(msg);  
        return true;  
    }  
  
    protected abstract void send(SuitRegistrationMessage msg);  
}
```

Jasper pointed to the screen. “Yeah, that `SuitRegistrationMessage` really ought to be `SuitRegistrationApprovalRequest`, shouldn’t it?”

“I agree.” And I quickly made the change. “Gee, we didn’t pick our names well, did we?”

A big grin spread across Jasper’s face. “Aw, Alphonse, don’t sweat it! You’ve only been working on this code for a day. Look how quickly we spotted this naming issue, and how easy it was to resolve.”

“Yeah, but if we’d thought about it a little more before we chose those names...”

“We wouldn’t have as many tests passing as we do! And we wouldn’t have been as sure about the names as we are now! C’mon Al! You know this!”

Jasper’s grin was wider than ever. His eyes were sparkling. I held up my hand and said: “OK, OK, you’re right. Cool down Jasper! And remember, my name is *Alphonse*.”

Jasper caught himself, nodded, and then turned back to the screen. “Right. Sorry. OK, let’s look inside the `SuitRegistrationApprovalRequest` class.”

```
public class SuitRegistrationApprovalRequest {  
    public String id;  
    public int argument;  
    public String sender;  
    final static String ID = "Suit Registration";  
  
    public SuitRegistrationApprovalRequest() {  
        id = ID;  
    }  
}
```

I looked over the class and then said: “Yeah, that ID string isn’t well named. It should be “`Suit Registration Approval`”.

I was about to make the change but Jasper stopped me. “I rather doubt that. I don’t think it should be

a string at all.”

“What do you mean? What should it be?”

Jasper scratched his head for a second and said. “It could be a unique integer, or an enum, or, I suppose, it could even be a string. It just shouldn’t be tied to the name of the class. I don’t want to change the ID code just because we decide to change the name of the string.”

“Oh yeah, SRP!” I said.

“Right! Single Responsibility Principle. *Good Alphonse!*”

I shot him a glance, and he wiped the growing grin off his face.

“We could still use a string.” I said. “How about this.” And I grabbed the keyboard.

```
final static String ID = SuitRegistrationApprovalRequest.class.getName();
```

Jasper looked at that for a second and said: “Hmmm. Clever. But it still changes whenever the class name changes.”

“Yeah, but *we* don’t have to make an explicit change to the source code!” I said enthusiastically.

“True, but since the string changes, it means that a cosmetic change to the source code could break the protocol with `Manufacturing`. The `Manufacutring` system would have to be recompiled and redeployed.”

“Hmmm. Yeah, but if we’re making changes to our code, won’t `Manufacturing` have to recompile their system anyway?”

Jasper froze in his seat. A funny puzzled expression passed over his face. “Wait a darn second! We can’t be sending *objects* to the `Manufacturing` system! We must be sending a packet based on XML or something!” Jasper sprang out of his seat and called over to Carole. “Carole, what is the format of the messages that we are sending back and forth to `Manufacturing`?”

Carole was working with Jean on something. She looked up and said “They are XML based. I put a description of the messages on the wiki yesterday. Check there.” And then she resumed work with Jean.

Jerry and Avery were busy working on something just across the table from us. Apparently Jerry had heard Jasper’s question because he looked up and said: “Yeah, I just read Carole’s wiki entry. The XML messages are pretty straightforward Jasper. No attributes or anything; just one tag per field. The packet that’s sent between the two systems is just raw text.”

Jasper nodded his thanks and sat back down. “OK, that’s good news. We can just forget about this issue for the time being.”

I was puzzled. “Why can we forget about it? Don’t we have to build the XML string?”

“Yes, but not right now. We’ll write a translation layer that takes our message objects and converts them to XML later.”

I thought about that for a few seconds and then said: “Oh, OK, so we’ll implement that abstract `send` method in the `Manufacturing` class to translate the request into XML.”

“Yeah, something like that.”

I nodded. That made sense. Some derivative of the `Manufacturing` class could worry about XML later. We didn’t have to worry about it now.

“OK, so now back to the original problem. How do we get this test to pass?”

Jasper said: “Go back to the original `DTrackFixture` class and look at the `registerSuit` method.”

```
public boolean registerSuit(int barcode) {  
    return Utilities.manufacturing.requestApprovalForRegistration(barcode);  
}
```

Jasper continued: “This fixture should not be calling `requestApprovalForRegistration`, should it? That function shouldn’t be called by a fixture at all. Asking for approval is *part* of the registration process. We need some object to handle the *whole* registration process.”

“Avery and I created just such an object yesterday afternoon! We called it the Registrar.”

“Yeah, I know.” Jasper blurted. “Jerry told me about it after dinner last night. That’s when we decided that you and I should work together today.”

I didn’t care for the sound of that. These journeymen were talking about Avery and me behind our backs, making plans for us. I suppose that’s part of their job, but I didn’t like it. If they are going to make plans that concern me, they should *involve* me!

I shook off that thought and filed it for later. “OK, so I have an idea.” I grabbed the keyboard and began to type.

```
public class DTrackFixture {
    ...

    public boolean registerSuit(int barcode) {
        return Registrar.attemptToRegisterNewSuit(barcode);
    }

    ...
}

```

```
public class Registrar {
    ...

    public static boolean attemptToRegisterNewSuit(int barcode) {
        return Utilities.manufacturing.requestApprovalForRegistration(barcode);
    }
}

```

Jasper half-smiled and half-smirked. The way he was shifting his eyes and jerking his head I knew that something was itching him. He was trying to keep from saying something. It occurred to me that I needed to get Jasper into a poker game. “Jasper, what are you smiling about?”

He actually burst out with a giggle, looked at me with that impossibly wide grin, and said: “Fonse....Alphonse, how long has it been since you ran a test?”

Ooops. It has been awhile. I’d done all those name changes. But those changes were trivial! I ran the tests.

One of the unit tests failed, and every acceptance test failed.

“Ouch!” I said.

“Ouch indeed!” Jasper said with an even bigger grin.

The code for this article can be located at:

http://www.objectmentor.com/resources/articles/CraftsmanCode/Craftsman_40_Do_sageTrackingSystem.zip

To be continued...

The Craftsman: 41

Dosage Tracking XVIII

Yeah, Sorta.

Robert C. Martin
23 August 2005

...Continued from last month.

In April of 1944, the three leaders of the Axis, Hitler, Stalin, and Tojo (Mussolini's status had declined of late) traveled to the Canary Islands to witness the launch of "Operation Underbelly", the Mexican Invasion. They were expecting to make very short work of Mexico and then to plow through the breadbasket of the United States.

At exactly 8pm on the eve of the launch, the sky directly above the Canary Islands suddenly blazed forth with actinic fury. For a fraction of a second the landscape was bathed in noon-day light. Anyone who happened to be looking straight up at the time was dazzled as though they had looked at the sun. There was no sound, no thunder, indeed, the silence was eerie. A second or so after the flash the sky took on a strange green glow that faded into a blood-red aura spreading across the horizon for several minutes.

In the hotel where the three Axis leaders were enjoying an evening meal, the electrical power failed, and telephone and radio communications were disrupted for a few minutes. When communications were restored the leaders learned that similar events had occurred directly above Berlin, Tokyo, and Moscow.

22 Feb 2002, 1000

"Damn, when did we break that unit test?" I was pretty embarrassed that I had allowed the tests fail. I made a mental note to run the tests more frequently.

Jasper's big toothy grin and his condescending demeanor continued to grate on me. "Why don't you check the error, Alphonse?"

The error said:

```
Expected:Suit Registration  
Actual   :dtrack.messages.SuitRegistrationApprovalRequest
```

I clicked on the error and it took me to the failing test. (I've marked the failing line with an arrow.)

```
public class ManufacturingTest extends TestCase {  
    public void testRegisterSuitSendsMessageToMfg() throws Exception {  
        MockManufacturing mfg = new MockManufacturing();  
        mfg.requestApprovalForRegistration(7734);  
    }  
}
```

```

    SuitRegistrationApprovalRequest message =
        (SuitRegistrationApprovalRequest) mfg.getLastMessage();
-> assertEquals("Suit Registration", message.id);
    assertEquals("Outside Maintenance", message.sender);
    assertEquals(7734, message.argument);
}
}

```

“Oh, OK.” I said. “When I changed the ID of the message, I forgot to change the test. That’s easy to fix.” So I changed the test as follows.

```

assertEquals("dtrack.messages.SuitRegistrationApprovalRequest",
    message.id);

```

I ran the unit tests and they all passed.

But Jasper didn’t like it, and he said so: “I don’t like it Alphonse.”

“Why not, Jasper? It works.”

“Sure it works, Alphonse, but if the message id changes again, the test will break again.”

“What do you suggest we do about it, Jasper?”

“How about this?” And with that, Jasper took the keyboard and changed the test to this:

```

assertEquals(SuitRegistrationApprovalRequest.ID, message.id);

```

This didn’t compile because the `ID` field wasn’t public. Jasper made than simple change than then ran all the tests. They all passed.

Then Jasper passed the keyboard back to me, locked my gaze with those sparking eyes of his, and gave me a quick double eyebrow raise. He clearly thought that his change was very clever. I was beginning to wonder just what it took to become a Journeyman around here. If Jasper was any indication, Mr. C’s standards must be pretty low.

I looked back at the screen and tried to focus on my job. “OK, I see your point. It’s better not to embed constants in the code if you can avoid it. Now, let’s see why so many of the acceptance tests are failing.”

“I’ll bet it’s for the same reason.” Jasper quipped.

He was probably right. The acceptance tests probably also mentioned the message ID by name.

“Yeah.” I said, and brought up the first failing acceptance test.

Message sent to manufacturing message id?	message argument?	message sender?
Suit Registration expected	314159	Outside Maintenance
dtrack.messages.SuitRegistrationApprovalRequest actual		

“You see!” Cried Jasper. “It’s the same issue.”

“Yes, I see Jasper. You are right. It’s the same issue.” I gave a deep sigh and shook my head.

Jasper noticed my mood and hung his head. “I’m sorry Alphonse, I’m doing it again aren’t I.”

“Yeah, sorta.” I said, without looking at him.

“OK, look, I’ll try to back off. I guess I’m just easily excited.”

I gave a snort, looked over at him with a grin and said: “Yeah, sorta!”

He smiled back, still embarrassed, and then we both turned back to the code.

“OK, Jasper, I think we need to apply your solution again. Instead of just changing the string in the test table, we need to see if the ID in the message equals the contents of the static ID variable in the class.”

“Yeah, I agree. Here’s something Jean showed me a few months back.” And he took the keyboard and began to modify the test tables.

```
!|Message sent to manufacturing|  
|Suit registration approval request?|message argument?|message sender?|  
|true|314159|Outside Maintenance|
```

“OK, I see.” I said. “We’re simply asking whether or not the message is the right kind. We aren’t even mentioning the notion of an ID.”

“Right. There’s no point in exposing a detail like the message ID, especially if it’s likely to change.”

Jasper saved the page and ran the test. Of course it complained.

“OK, now we have to add the `SuitRegistrationApprovalRequest` method to the fixture.” Said Jasper.

“I’ll do it.” And I grabbed the keyboard and made the appropriate changes to the fixture.

```
public class MessageSentToManufacturing extends ColumnFixture {  
    ...  
  
    public boolean suitRegistrationApprovalRequest() {  
        return message.id.equals(SuitRegistrationApprovalRequest.ID);  
    }  
  
    ...  
}
```

I saved this and ran the test, and it passed.

For a second Jasper’s eyes showed the same old excitement, but he reigned it back in and simply said: “OK, good. Now how about the rest of the acceptance tests?”

We found another that was failing for the same reason, and quickly fixed it. That left one more failing acceptance test: `RejectDuplicateRegistration`.

“Oh yeah!” I said happily. “That’s what we started working on this morning. This one should be failing because we haven’t finished getting it working yet.”

fitlibrary.DoFixture			
start	dtrack.fixtures.DTrackFixture		
set	314159	as registered	
suit			
check	register suit	314159	false expected <hr/> true actual
check	was a message sent to manufacturing	false expected <hr/> true actual	
check	count of registered suits is	1	
check	error message	Suit 314159 already registered.	true expected <hr/> false actual

“OK.” I said. “This is failing because the `Registrar` is not detecting that suit# 314159 has already been registered. We should be able to fix that pretty easily.”

“Be my guest.” Said Jasper, still somewhat subdued.

So I brought up the code for the Registrar and found the appropriate method.

```
public static boolean attemptToRegisterNewSuit(int barcode) {  
    return Utilities.manufacturing.requestApprovalForRegistration(barcode);  
}
```

“Yeah, see, it’s just passing the request to manufacturing without checking.”

So I modified the code as follows:

```
public static boolean attemptToRegisterNewSuit(int barcode) {  
    if (SuitGateway.isSuitRegistered(barcode))  
        return false;  
    return Utilities.manufacturing.requestApprovalForRegistration(barcode);  
}
```

I talked while I typed. “OK, now we need to write the IsSuitRegistered method.”

Jasper stopped me. “No, Alphonse. Now we need to write the unit test for IsSuitRegistered.”

“Right! I almost forgot.” So I looked for the appropriate unit test to modify, but couldn’t find one.

“Wow, it looks like we wrote those gateways without any unit tests!” I brought up the InMemorySuitGateway that we’d been using.

```
public class InMemorySuitGateway implements ISuitGateway {  
    private Map suits = new HashMap();  
    public void add(Suit suit) {  
        suits.put(new Integer(suit.barCode()), suit);  
    }  
  
    public int getNumberOfSuits() {  
        return suits.size();  
    }  
  
    public Suit[] getArrayOfSuits() {  
        return (Suit[]) suits.values().toArray(new Suit[0]);  
    }  
}
```

Jasper looked at the code and said: “OK, this is pretty simple code. I can see why the unit tests didn’t get written, but it’s time to write one now.” And he grabbed the keyboard and started to write the test.

```
public class InMemorySuiteGatewayTest extends TestCase {  
    public void testIsSuitRegistered() throws Exception {  
        InMemorySuitGateway g = new InMemorySuitGateway();  
        assertFalse(g.isSuitRegistered(314159));  
        g.add(new Suit(314159, new Date()));  
        assertTrue(g.isSuitRegistered(314159));  
    }  
}
```

Then, step by step, he wrote the implementation.

```
public boolean IsSuitRegistered(int barcode) {  
    return suits.containsKey(barcode);  
}
```

The unit tests all passed.

“OK, now we have to connect that to the SuitGateway class.”

```

public interface ISuitGateway {
    ...
    boolean isSuitRegistered(int barcode);
}

public class SuitGateway {
    public static ISuitGateway instance = null;

    ...

    public static boolean isSuitRegistered(int barcode) {
        return instance.isSuitRegistered(barcode);
    }
}

```

We ran the unit tests, and they all passed. Then we ran the `RejectDuplicateRegistration` acceptance test.

fitlibrary.DoFixture			
start	dtrack.fixtures.DTrackFixture		
set	314159	as registered	
suit			
check	register suit	314159	false
check	was a message sent to manufacturing	false	expected
		true	actual
check	count of registered suits is	1	
check	error message	Suit 314159 already registered.	true
			expected
			false
			actual

“Great!” I said. “That first failing cell of the test is now passing. Only two more to go.”

Jasper looked up at me and said: “I need a break. Let’s go watch the starbow for awhile.”

“Sounds like a plan.” Working with Jasper was tiring. Perhaps he found working with me to be tiring too.

The code for this article can be located at:

http://www.objectmentor.com/resources/articles/CraftsmanCode/Craftsman_40_Do_sageTrackingSystem.zip

To be continued...

The Craftsman: 42

Dosage Tracking XIX

Devious Thoughts

Robert C. Martin
2 October 2005

...Continued from last month.

The High Altitude Nuclear Explosions (HANES) went off with clockwork precision. General MacArthur nodded with quiet satisfaction as he received the reports of the detonations. Better still, reports from covert agents confirmed that preparations for the invasion appeared to have ground to a halt. MacArthur eagerly anticipated the satellite photos in the coming days. He expected them to show that the Axis powers were conducting a mass withdrawal and skulking back to their bases.

But as the days and weeks wore on, no photos arrived. To his growing dismay, MacArthur learned that virtually every spy satellite in orbit during the HANES and those that had been launched days, and even weeks later, quickly stopped operating. Eventually he was told that it had something to do with high energy radiation belts created by the explosions.

General MacArthur did not like being blind.

22 Feb 2002, 1030

“Jasper is driving me nuts!” I said to Jerry in the break room, once I had found a way to escape from Jasper.

Jerry smiled and nodded knowingly. “Yeah, I know what you mean. He’s a bit intense isn’t he?”

“A bit? I’ve confronted him two or three times, and yet he keeps on being condescending. It’s like he wants to be my big brother or something.”

“Jasper is a good programmer, Alphonse. There’s a lot you can learn from him. So try to get passed his personality quirks. OK?”

“If you say so. I haven’t been impressed so far. He makes me crazy!”

“Well, you’ve been working with him for a couple of hours today. Perhaps it’s time to switch partners. If you like, I’ll ask him to help Avery, while I work with you.”

That sounded like a really great idea. Getting away from Jasper, even for just a few hours, had become an imperative. “I’d be in your debt!”

“OK, I’ll take care of it. Besides, it should be fun to see how *those* two get along.” Jerry gave me a wink; and I realized that putting Avery and Jasper together could indeed light off some fireworks.

I wandered over to a window to watch the starbow make it’s lazy circular trek around our ship. The starbow was a hoop of colored stars that encircled our bow, positioned as though the ship were trying to fly through it. It’s leading edge was a pale blue that grew in intensity to white at the centerline and then faded away into red. At slower speeds, the colors would fade and the starbow would widen to become the stars of

the nighttime sky. I knew the effect was caused by a combination of Doppler shift, and optical aberration, but that didn't keep me from being mesmerized by it's stark beauty.

Jerry came by a few minutes later and said: "OK, it's all set. This is actually a good breaking point for Avery and I, since we just finished the story we were working on. I've asked Jasper to work with Avery on the story that the two of you were doing. You can help me write some more acceptance tests."

This was good news. I could use a break from the code; and I was tired of that Suit Registration story too. But I had a question.

"Isn't Carole supposed to be writing the acceptance tests, Jerry? After all, she's our customer, isn't she?"

"Oh, she *is* writing them. You'll see her working with Jean and I on them. But she can't write them all, so we help her."

We started walking back to the lab as we talked. Jerry was thoughtful for a second and then said:

"The normal process is for the customer, or analysts that report to the customer, to write the primary acceptance tests, and for QA to write the alternate acceptance tests."

"What's a primary acceptance test?"

"Oh, yeah, you haven't met Ivar yet. It's a term that Ivar uses to describe a test that describes the 'happy path' of a feature."

"Happy path?"

"Yeah, when everything goes right. You know, no invalid entries, no failures, no exceptions thrown, no cockpit error."

"Oh, ok, so that first Suit Registration acceptance test we did was 'Primary' because it tested that a suite would be registered if everything went OK. But the second one we were working on showed that registration failed if the suit was already registered, so it would be – er – Alternate?"

"Yeah, you got it. Anyway, QA are the folks who are trained to think of all the things that can go wrong. They explore the boundary conditions and the failure scenarios. So they usually write the 'Alternate' acceptance tests."

"Usually?"

"Yeah. Have you seen any QA folks in the lab?"

"No, it's just been you, me, Avery, Jasper, Carole, and Jean. We see Jasmine and Adelaide from time to time, but they are working on SMC."

Jerry nodded. "Right. We don't have QA on our team yet. Jean's trying to get one or two QA folks assigned. Until then, it's up to us."

"OK, I see. So you and I are going to write some alternate acceptance tests?"

"Yeah, that's the plan. We'll look at the primary tests that Carole has written, and then try to imagine everything that could go wrong with them and write tests for those cases."

We walked in silence until we reached the lab, and then sat down where Jerry and Avery had been working. Jerry pulled up the FitNesse site and navigated to the `StoryDescriptions` page. It looked like this:

```
!path C:\DosageTrackingSystem\classes

^RegisterSuit
^UnRegisterSuit
^AlertManagerIfOutsideUserPastDosageLimit
^SuspendUser
^SuspendedUserAttemptsToCheckOutSuit
^AddUser
^DeleteUser
^UserDosageReport
^UserHistoryReport
^SuitHistoryReport
^SuitInventoryReport
```

```
^CheckOutSuit
^CheckInSuit
^AttemptToCheckOutSuitThatRequiresInspection
^AttemptToCheckOutSuitWhenUserOverMonthlyLimit
^SuitInspection
```

“Wow!” I said. “Somebody’s been busy.”

“Yeah, Carole and Jean have been entering primary acceptance tests since yesterday. They’ve got a lot of them done. Let’s look at `UnRegisterSuit`.”

Jerry clicked on the link, and the screen showed the following:

```
^UnRegisterRegisteredSuit
^UnRegisterCheckedOutSuit
```

“OK, I’ll bet that first page simply unregisters a registered suit.”

Was this a joke? I flashed Jerry a big Jasper grin and said: “Given the name, I’d say that’s a good bet.”

Jerry smirked and clicked the link.

Unregister a suit.				
Users can unregister a suit that has been properly registered. This will remove the suit from inventory.				
set suit	314159	as registered		
check	unregister suit	314159	true	
check	count of registered suits is	0		
check	message	Suit 314159 unregistered.	was printed	true

We both examined the page for a few seconds. Then Jerry said: “OK, that’s what I thought. She’s loading a registered suit into the database and then unregistering it. Then she makes sure that the count of suits has been decremented, and that the appropriate message was displayed.”

“Wait!” I said. “Jasper and I just wrote some of those fixture function in the last hour or so. How did Carole and Jean know to use them?”

“Didn’t you tell them?”

“No, should we have?”

“Well, it would have been polite. They do have a lot of acceptance tests to write, and I’m sure they’d appreciate any help they can get.”

“Uh…”

“Anyway, these tests *are* on a wiki. They must have seen what you and Jasper did, and made use of your ideas.”

“Really?”

“What? Don’t you think your ideas are worth using?”

“I, uh…”

“We re a team, Alphonse. We’re not a group of disconnected developers. We look at each other’s work and learn from it.”

“OK, sure, I just didn’t expect it so soon, that’s all.”

“So let’s look at that `UnRegisterCheckedOutSuit` acceptance test. I’ll bet it tries to unregister a suit that’s been checked out, and makes sure the unregistration fails.”

“Checked ou?”

“Yeah...being worn by someone outside the ship.”

“OH! Yeah, it wouldn’t be a good idea to unregister a suit that somebody was using.”

Jerry grimaced in agreement and clicked on the link.

You cannot unregister a suit that is checked out.

If a user has checked out a suit, then we cannot allow the suit to be unregistered.

set	suit	314159	as registered		
set	suit	314159	as checked out to user	Bob	
check	unregister suit	314159	false		
check	count of registered suits is	1			
check	message	Could not unregister suit 314159. Checked out to Bob.	was printed	true	

I looked it over and beat Jerry to the punch. “Yeah, she’s just forcing the suit to be checked out, and then showing that unregistration fails.”

“Right. OK, so what has she missed?”

“You mean on this page?”

“No, I mean, what other tests has she missed. Think deviously!”

“I, uh...” What did it mean to think deviously?

“For example, what would happen, Alphonse, if you tried to unregister a suit that was already unregistered? What *should* happen?”

“Oh! I see what you mean. Well, clearly we’d want the unregistration request to fail, and display some appropriate message.”

“Good. So write the test for that!”

I thought about it for a second and then created a new page named `UnRegisterUnRegisteredSuit`. I smiled when I wrote that name. It thought it was clever. Jerry rolled his eyes, but remained silent. So I wrote the following.

You can't unregister a suit that's not already registered.

Any attempt to do so should be ignored with an appropriate error message.

check	unregister suit	314159	true		
check	count of registered suits is	0			
check	message	Suit 314159 was not registered.	was printed	true	

Jerry looked it over and said: “Yeah, that looks about right.” Then he looked up and called over to Carole: “Hay Carole, look at the `UnregisterUnregisteredSuit` page. Does that look right?”

Carole nodded and spend a few seconds apparently navigating to the page. Then she gave us a smile and a thumbs-up and got right back to work.

“Great.” Said Jerry. “Now, Alphonse, any other devious thoughts?”

I thought for a bit and said: “No, I can’t think of any other alternate tests.”

“OK, then. let’s look at the next story.”

The code for this article can be located at:

http://www.objectmentor.com/resources/articles/CraftsmanCode/Craftsman_42_DoSageTrackingSystem.zip

To be continued...

The Craftsman: 43 Dosage Tracking XX Language Lawyers

Robert C. Martin
27 October 2005

...Continued from last month.

May, 1944. Henry Wallace had a brass paperweight in the form of a globe of the Earth. He kept it on his desk in the Oval Office. Embossed on the bottom was: "29 Apr, 1959, 0943 GMT". "Fifteen years." He mumbled to himself. "Fifteen years." He wondered how he could save a world that was so hell bent on destroying itself. He reckoned that he had two bad options: collaborate or conquer. The first seemed infeasible given the belligerence of the axis leaders; they yearned to conquer the Americas, not collaborate with them. The second option was feasible, but distasteful. He could, in fact, conquer the world, and he could do it quickly. But could he keep it dominated for the necessary fifteen years? Not likely.

One thing was certain. Option two would not last long. Having seen that nuclear explosions were possible; Hitler, Stalin, and Tojo had to be pushing their remaining scientists to the breaking point.

22 Feb 2002, 1100

Just as we were about to start on the next test case, Jasper walked over with that big depressing grin plastered all over his face.

"Hey there Jerry! Can I borrow you for a second or three?"

Jerry looked up at Jasper without a flinch and said: "Sure, Jasper. Alphonse, I'll be back in a few minutes." Jerry got up and walked away with Jasper as if nothing were wrong; but then he quickly turned around, rolled his eyes, and winked at me.

I smiled back at both of them and said: "OK see you soon. I'll go see what Avery is up to."

I sauntered over to where Avery was sitting and said: "Hi."

Avery looked up and gave me a desperate grimace. "Why did you leave me with that moron?"

"He's not a moron. Actually he's pretty smart. But he can be trying."

"Trying? If he smiles at me one more time I'm going to shove this keyboard into his mouth. And it'll fit too."

We both laughed, making sure no one could overhear us. Then Avery said: "Hey, have you looked at the new features in Java 5?"

"Yeah, you mean autoboxing, scanners, generics, and stuff like that?"

"Yeah, have you used any of it yet?"

"Not so far, although we could have used generics yesterday when we wrote a mock database. I just didn't think of it then."

Avery's eye's sparkled. "Oh yeah? Show me!"

This sounded like fun. So I sat down next to him and brought up the `InMemorySuitDatabase`.

```
public class InMemorySuitGateway implements ISuitGateway {
    private Map suits = new HashMap();
    public void add(Suit suit) {
        suits.put(new Integer(suit.barCode()), suit);
    }

    public int getNumberOfSuits() {
        return suits.size();
    }

    public Suit[] getArrayOfSuits() {
        return (Suit[]) suits.values().toArray(new Suit[0]);
    }

    public boolean isSuitRegistered(int barcode) {
        return suits.containsKey(barcode);
    }
}
```

We both looked at this for a second and then Avery said: "Yeah, we can change that `HashMap` real easily." He grabbed the keyboard and made the following change:

```
public class InMemorySuitGateway implements ISuitGateway {
    private Map<Integer, Suit> suits = new HashMap<Integer, Suit>();
    public void add(Suit suit) {
        suits.put(new Integer(suit.barCode()), suit);
    }
    ...
}
```

The tests all still ran.

"Cool!" I said. Now we should be able to use autoboxing in the `add` method! So I grabbed the keyboard and did this:

```
public class InMemorySuitGateway implements ISuitGateway {
    private Map<Integer, Suit> suits = new HashMap<Integer, Suit>();

    public void add(Suit suit) {
        suits.put(suit.barCode(), suit);
    }
    ...
}
```

The tests all still ran!

"Way cool! That's much cleaner than before!" I said.

"Yeah! And now we can get rid of that ugly cast in `getArrayOfSuits`." Avery grabbed the keyboard and deleted the cast.

```
public Suit[] getArrayOfSuits() {
    return suits.values().toArray(new Suit[0]);
}
```

The tests ran just fine.

"Sweet!" I said. Too bad we don't have anything else to clean up with generics. This is fun!

“Well...” said Avery. “Have you considered whether or not there will be more than one kind of Suit?”

“What? You mean like different derivatives of Suit?”

“Yeah. Imagine that you have two derivatives like `MensSuit` and `WomensSuit`.

“You mean like this?” And I grabbed the keyboard and typed the classes in a junk package.

```
public class WomensSuit extends Suit {
    public WomensSuit(int barCode, Date nextInspectionDate) {
        super(barCode, nextInspectionDate);
    }
}

public class MensSuit extends Suit {
    public MensSuit(int barCode, Date nextInspectionDate) {
        super(barCode, nextInspectionDate);
    }
}
```

Avery watched as I typed, and then said: “Yeah. Now create a class that has two lists. One a list of `MensSuit` and the other a list of `WomensSuit`.”

“OK.” I said, and I typed the following:

```
public class SuitInventory {
    private List<MensSuit> mensSuits = new ArrayList<MensSuit>();
    private List<WomensSuit> womensSuits = new ArrayList<WomensSuit>();
}
```

Avery nodded and said: ‘Great! Now let’s assume we want to inspect suits. We can use a method named `inspect` that takes a list of suits.’”

“You mean like this?” And I typed the following:

```
void inspect(List<Suit> suits) {
    for (Suit s : suits) {
        // inspect s.
    }
}
```

“Impressive, Alphonse! You made fine use of the new iteration syntax.”

I had missed the old formal banter. Sometimes working with Avery was a lot of fun. Especially after working with Jasper. “Indeed I did, Avery, you are quite observant!”

“No more observant than you are creative; but shall we continue?”

“Indeed, let’s.”

“Well then, suppose that we now decide to create a method for inspecting only `WomensSuit` objects?”

“I imagine I can write such a method. Shall I?”

“Please do.”

And so I typed the following:

```
void inspectWomensSuits() {
    inspect(womensSuits);
}
```

“Hay!” I said, forgetting the formal banter for the moment. “That doesn’t compile!”

“As, indeed, it should not!” Quipped Avery. He maintained his formal composure.

“But I don’t understand, it should compile!”

“No, Alphonse, if you look carefully you’ll notice that we are attempting to pass a `List<WomensSuit>` to an argument defined to take a `List<Suit>`.”

“Sure, but a `List<WomensSuit>` *is* a `List<Suit>`!”

“Would you like to reconsider that statement Alphonse?”

“Uh...” I thought for a moment and realized that that wasn’t quite right. “OK, technically you’re right, a `List<WomensSuit>` does not derive from `List<Suit>`, but a list of womens suits is still a list of suits.”

“Not a all! You can add an instance of `MensSuit` to a `List<Suit>`, but you can’t add a `MensSuit` to a `List<WomensSuit>`.”

“OK, that’s true, but so what?”

“So, my dear Alphonse, a `List<WomensSuit>` is not substitutable for a `List<Suit>`, and is therefore not a subtype of `List<Suit>`. You are of course aware of the Liskov Substitution Principle¹, are you not?”

That’s when it clicked. “Of course. Of course! Substitutability is the basis of subtyping. Therefore a list of derivatives is not a derivative of the list of bases.”

“Precisely Alphonse!”

“Yes. Precisely! But that leaves us with a problem, doesn’t it my dear Avery?”

“Indeed it does! If a list of derivatives is not substitutable for a list of superclasses, what is?”

“And is there an answer to this connundrum?” I asked?

“Indeed there is. Shall I demonstrate?”

“Please do!”

So Avery took the keyboard and made the following change:

```
void inspect(List<? extends Suit> suits) {  
    for (Suit s : suits) {  
        // inspect s.  
    }  
}
```

I looked at the strange syntax that Avery had written. “Now that is one strange syntax; but I think I understand the intent. The question mark indicates that the type is unknown, and the remainder of the clause tells the compiler that the unknown type is a derivative of `Suit`.”

“Indeed, indeed. That is the correct interpretation Alphonse. Notice also that the whole program now compiles.”

“Yes, I see that. This is all fascinating! There is great power and expressiveness in these new features.”

“Yeah, and they’re fun too!” Avery said, switching modes.

I gave him a poke in the ribs, and he bopped me on the head with a one of the books on the table.

“Now, now boys, don’t go breaking anything.” It was Jean. She must have come by to see what we were up to. “Goodness! You boys certainly have a lot of energy.” She looked at the screen and exclaimed: “Oh! I see you boys are having fun with the rather odd new syntax for generics. For my money it’s all a bit involved, isn’t it. So much syntax for so little effect. I fear the language lawyers may be leading us into a deep and unnecessary complexities. Such a shame too. But then I suppose youngsters like you revel in such intricacies, don’t you. Well then, carry on, carry on. You are such fine young men.”

Jean waddled off, while Avery and I suppressed giggles.

The code for this article can be located at:

¹ See <http://www.objectmentor.com/resources/articles/lsp.pdf>

http://www.objectmentor.com/resources/articles/CraftsmanCode/Craftsman_43_DosageTrackingSystem.zip

To be continued...

The Craftsman: 44

Brown Bag I

Java Generics 2

Robert C. Martin
22 November 2005

...Continued from last month.

June, 1944.

Intelligence analyst Jennifer Kohnke scanned the satellite photos again and again. Since the birds had begun flying again last month, her workload had become nearly unbearable. Her supervisors kept on handing her photos of some kind of complex in the eastern part of the Reich. They wanted to know what the Germans were doing there. The heavy train traffic implied that the complex was industrial. So did the large number of barracks for the workers. The problem was that she didn't see any kind of significant factory. There was just one small building with big chimneys. It was too small for any real industrial purpose. Yet huge volumes of smoke continuously poured from those chimneys.

Avery and I learned a lot from experimenting with Java generics last Friday. So we decided to make such explorations a daily ritual from now on. We planned to meet at 11:00 every morning for a few minutes to discuss and show off, some new thing we had learned. Today it was my turn.

Monday, 25 Feb 2002, 1100

Japer and I had been working all morning on the Suspend User story, making good progress on it. At 11:00, Avery walked up and said: "Ready?"

I looked at Jasper and said: "Would you mind if I took a break for 15 min?"

He gave me that bright-eyed toothy grin of his and said: "Sure, Alphonse, have a blast!"

"Thanks. I'll see you shortly."

Avery and I found a conference room. We shut the door, and I set up my laptop. The two of us huddled around it. And I began.

"OK, now suppose I have a function that looks like `getSuitsOverdueForInspection`:

```
public class SuitInventory {
    private List<MensSuit> mensSuits = new ArrayList<MensSuit>();
    private List<WomensSuit> womensSuits = new ArrayList<WomensSuit>();

    public void getSuitsOverdueForInspection(List<Suit> overdueSuits) {
        for (Suit s : mensSuits)
            if (s.overdueForInspection())
                overdueSuits.add(s);

        for (Suit s : womensSuits)
            if (s.overdueForInspection())
```

```

        overdueSuits.add(s);
    }
}

```

“Ack!” said Avery. “Duplicate code! You have to refactor that!” So Avery grabbed the keyboard and reduced the two loops into a single loop:

```

public void getSuitsOverdueForInspection(List<Suit> overdueSuits) {
    getSuitsOverdueForInspectionFromList(mensSuits, overdueSuits);
    getSuitsOverdueForInspectionFromList(womensSuits, overdueSuits);
}

private void getSuitsOverdueForInspectionFromList(
    List<Suit> suitList, List<Suit> overdueSuits) {
    for (Suit s : suitList)
        if (s.overdueForInspection())
            overdueSuits.add(s);
}

```

I stopped him right there. “Yes, Avery, Right. But notice that this doesn’t compile!”

Avery looked carefully at the screen and noticed that the two calls to `getSuitsOverdueForInspectionFromList` were complaining about the type of the first argument.

“Right!” He said. “This is just what we talked about on Friday. A list of derivatives, like `List<MensSuit>` cannot be passed as a List of base classes like `List<Suit>`. And we know what to do about this.”

So Avery continued typing, fixing the compile errors as follows:

```

private void getSuitsOverdueForInspectionFromList(
    List<? extends Suit> suitList, List<Suit> overdueSuits) {
    for (Suit s : suitList)
        if (s.overdueForInspection())
            overdueSuits.add(s);
}

```

“Precisely!” I said. “This makes the `getSuitsOverdueForInspectionFromList` method more generic than either of the two earlier loops. So in that sense `? extends Suit` is more generic than `Suit`. However, I have a problem. I would like to call the first function, `getSuitsOverdueForInspection`, from a number of different places. Let me show you. Let’s say that I have a `SuitInspector` class like this:”

```

public class SuitInspector {
    private SuitInventory inventory = new SuitInventory();

    public void inspectSuits() {
        List<Suit> overdueSuits = new ArrayList<Suit>();
        inventory.getSuitsOverdueForInspection(overdueSuits);
        for (Suit overdueSuit : overdueSuits) {
            overdueSuit.inspect();
        }
    }
}

```

Avery looked at this for a second or two and said: “OK, This class just finds all the suits that are overdue for inspection, and inspects them.”

“Right!” I said. “But I also have a `GeneralInspector` class that looks like this:”

```

public class GeneralInspector {
    private SuitInventory suits = new SuitInventory();
    private ValveInventory valves = new ValveInventory();

    public List<Object> getOverdueItems() {
        List<Object> overdueItems = new ArrayList<Object>();
        suits.getSuitsOverdueForInspection(overdueItems);
        valves.getValvesOverdueForInspection(overdueItems);
        return overdueItems;
    }

    public void PrintOverdueInspectionReport() {
        List<Object> overdueItems = getOverdueItems();
        for (Object overdueItem : overdueItems) {
            System.out.printf("%s is overdue\n", overdueItem);
        }
    }
}

```

Again, Avery stared and the screen for a few seconds and then said: “OK, sure. you are just gathering up the overdue Suit and Valve objects and then printing them. Hay! What’s that printf statement?”

“That?” I said with a smirk. “That’s actually pretty retro. It’s sort of like the old C printf statement. Maybe you can talk about that one tomorrow. Or maybe I will. For now, the more interesting thing is that the two getXXXOverdueForInspection lines don’t compile.”

“Well of course they don’t” Avery Sneered. “You are passing a List<Object> to them, when they respectively take List<Suit> and List<Valve>”.

He was falling into my trap. Good. “So how should we fix this?”

“You need to change the arguments of the getXXXOverdueForInspection functions to take List<? extends Object>.” And Avery grabbed the keyboard and began to type.

Before he was finished I said: “I’d like to point out that widening the type of the argument to getSuitsOverdueforInspection is a bit strange. It should be taking a List<? extends Suit>!” But Avery pretended not to hear as he pounded out the following code:

```

public void getSuitsOverdueForInspection(
    List<? extends Object> overdueSuits) {
    getSuitsOverdueForInspectionFromList(mensSuits, overdueSuits); // broken
    getSuitsOverdueForInspectionFromList(womensSuits, overdueSuits); // broken
}

```

“Oh no!” Avery shrieked. “This made GeneralInventory compile, but broke the calls to getSuitsOverdueForInspectionFromList.

“Right!” I said, because you can’t pass a List<? extends Object> to an argument that expects a List<Suit>.”

“OK.” Avery said. “So we just have to widen the argument of getSuitsOverdueForInspectionFromList like this:”

```

private void getSuitsOverdueForInspectionFromList(
    List<? extends Suit> suitList, List<? extends Object> overdueSuits) {
    for (Suit s : suitList)
        if (s.overdueForInspection())
            overdueSuits.add(s); // broken
}

```

“Aarrgghh!” Avery cried. Now the ‘add’ statement is broken.”

“Right again!” I said. “That’s because you can’t add something to a list of unknown type. ?

extends Object is a truly unknown type!”

Avery looked near panic. I could see the fear of casts and `instanceof`s in his face. But then he seemed to suddenly realize that I had sprung a trap on him. He visibly relaxed; put a sheepish smile on his face, and said: “OK, so I’m sure you have a solution to this conundrum.”

“Indeed I do!”

“Play on, Maestro! Play on!”

“It turns out, my dear Avery, that you can widen the type of a list by using `? extends X`, only if you plan on *reading* from that list. If, however, you plan on writing to that list; as our `getSuitsOverdueForInspection` functions do, then you have to use a different form. You have to use `<? super X>`.

Avery drew breath in an audible hiss.

I continued: “Allow me to show you how this works.” And I made the following changes to the `SuitInventory` class.

```
public void getSuitsOverdueForInspection(List<? super Suit> overdueSuits) {
    getSuitsOverdueForInspectionFromList(mensSuits, overdueSuits);
    getSuitsOverdueForInspectionFromList(womensSuits, overdueSuits);
}

private void getSuitsOverdueForInspectionFromList(
    List<? extends Suit> suitList,
    List<? super Suit> overdueSuits) {
    for (Suit s : suitList)
        if (s.overdueForInspection())
            overdueSuits.add(s);
}
```

“This settles the issue.” I said. “Now the functions take a `List<? super Suit>`, which is a list of unknown types that are known to be lists of something *no more derived* than `Suit`.”

“No more derived...” echoed Avery. “That makes my head hurt.”

“Think of it this way, Avery.” When you put an object into a list, all you care about is that the object is of a type that is compatible with the type held by the list. So you want the list to take the type of that object, or one of the superclasses of that object.”

“Ow!, Ow! Ow!. My head is hurting”

“On the other hand, when you *read* from a list, you want the type you are reading to be the type contained by the list, or a derivative of that type.”

“Ow!”

“So when you read from a list, you use `<? extends X>`, but when you write to a list you use `<? super X>`. See?

“Ow!”

“Don’t you think that’s cool?”

“Ow!”

This was fun! Avery is usually the superior one, and he just couldn’t keep up with me on this one. I’m sure it would be my turn next. But that would be fun too.

To be continued...

The Craftsman: 45

Brown Bag II

Java Generics 2

Robert C. Martin
6 December 2005

...Continued from last month.

July, 1944.

Werner Von Braun tipped his head back and gave a loud belly-laugh. It was just dumb luck that the paper written by Stanislaw Ulam, a member of the contingent, and part of project Nimbus, had crossed his desk. It wasn't a paper about rockets or anything related to rocketry. Indeed, Ulam proposed that instead of using expensive and unreliable rockets to propel an intercontinental missile to its target, you could drop tiny atomic bombs out the back end and blow them up behind a protective pusher plate. Von Braun laughed and laughed, because he realized that Ulam was thinking far too small.

Tuesday, 26 Feb 2002, 1100

Avery walked into the conference room just after me. He had a cocky look on his face, so I knew he had something good for me.

“OK, Alphonse”, he began, “Suppose you must print an inspection report for each spacesuit in the inventory. The report contains a line for each suit. Men’s suits have three inspection points labeled A, B, and C, whereas women’s suits have two, A and B. A typical report might look like this:” And he showed me an index card with the following.

```
314159 MS A:3, B:4, C:2
749445 WS A:2, B:6
```

“This report describes two suits” he went on. “The first is a men’s suit, and the second is a woman’s suit, as shown by the MS and WS. The values of each inspection point is printed following its name.” Avery stopped and looked at me expectantly.

“OK”, I said, “That’s clear. So what’s your point?”

“How would you design the program that prints this report?”

“I’ll show you.” I said, and I began to type.

```
public class SuitInspectionReport {
    public String generate(SuitInventory inventory) {
        StringBuffer report = new StringBuffer();
        for (Suit s : inventory.getSuits()) {
            report.append(s.getInspectionReportLine()).append("\n");
        }
    }
}
```

```

        return report.toString();
    }
}


---


public abstract class Suit {
    protected int barcode;

    public Suit(int barcode) {
        ...
    }

    public abstract String getInspectionReportLine();
}


---


public class MensSuit extends Suit {
    private int ipa;
    private int ipb;
    private int ipc;

    public MensSuit(int barCode) {
        super(barCode);
    }

    public String getInspectionReportLine() {
        return String.format("%d MS A:%d, B:%d, C:%d",
                               barcode, ipa, ipb, ipc);
    }
}


---


public class WomensSuit extends Suit {
    private int ipa;
    private int ipb;
    public WomensSuit(int barCode) {
        super(barCode);
    }

    public String getInspectionReportLine() {
        return String.format("%d WS A:%d, B:%d",
                               barcode, ipa, ipb);
    }
}

```

“Very good!” said Avery. “I thought you might do something like that. But don’t you think something smells about that design?”

“You mean the short variable names?”

“No, I mean something about the structure.”

I looked the code over, but didn’t see anything wrong with it. So I shrugged and looked at Avery expectantly.

Avery signed and said: “Think about design principles, Alphonse.”

“Oh, right! SRP¹!”

Avery gave me a condescending smile and said: “Yeah, right!” He was enjoying this. “The Single Responsibility Principle says that you shouldn’t put code that changes for different reasons into the same class.”

“Right, right.” I said, trying to get ahead of Avery. “The report format is going to change for different reasons than the rest of the code in the `Suit` derivatives.”

“Right, the `Suit` derivatives will contain business rules for managing suits, and should not be coupled to something as transient as report formats. And another thing: what if there are a dozen different kinds of reports for suits? Do we really want a dozen `getXXXReportLine` methods in `Suit`?”

¹ <http://www.objectmentor.com/resources/articles/srp>

“OK, you got me. OCP² is violated too.”

“Bingo, Alphonse. The Open Closed Principle implies that you shouldn’t set up a class so that it will require continuous modification. Every time there is a new kind of report, the `Suit` classes will have to be changed.”

I sighed and said: “OK, Avery, you’ve made your point. The `getInspectionReportLine` method doesn’t belong in `Suit`. That’s a shame since it was so nicely polymorphic.”

“Yeah, but it with two major principle violations...”

“Yeah. So what do we do about it?”

Avery smiled and said: “You were right when you said that the `getInspectionReportLine` method was nicely polymorphic. If possible we’d like to preserve that polymorphism, but remove the method from the `Suit` hierarchy. It turns out that there is a simple way to add polymorphic methods to hierarchies without modifying those hierarchies.”

“OK, you’ve got my interest. How do you add polymorphic methods to a hierarchy, without modifying that hierarchy?”

“Watch this.” Avery said with a smirk. And he began to type.

```
public abstract class Suit {
    protected int barcode;

    public Suit(int barcode) {
        this.barcode = barcode;
    }

    public abstract void accept(SuitVisitor v);
}

public interface SuitVisitor {
    public void visit(MensSuit ms);
    public void visit(WomensSuit ws);
}

public class MensSuit extends Suit {
    int ipa;
    int ipb;
    int ipc;

    public MensSuit(int barCode) {
        super(barCode);
    }

    public void accept(SuitVisitor v) {
        v.visit(this);
    }
}

public class WomensSuit extends Suit {
    int ipa;
    int ipb;

    public WomensSuit(int barCode) {
        super(barCode);
    }

    public void accept(SuitVisitor v) {
        v.visit(this);
    }
}

public class SuitInspectionReportVisitor implements SuitVisitor {
```

² <http://www.objectmentor.com/resources/articles/ocp>

```

    public String line;

    public void visit(MensSuit ms) {
        line = String.format("%d MS A:%d, B:%d, C:%d",
                               ms.barcode, ms.ipa, ms.ipb, ms.ipc);
    }

    public void visit(WomensSuit ws) {
        line = String.format("%d WS A:%d, B:%d",
                               ws.barcode, ws.ipa, ws.ipb);
    }
}

public class SuitInspectionReport {
    public String generate(SuitInventory inventory) {
        StringBuffer report = new StringBuffer();
        SuitInspectionReportVisitor v = new SuitInspectionReportVisitor();
        for (Suit s : inventory.getSuits()) {
            s.accept(v);
            report.append(v.line).append("\n");
        }
        return report.toString();
    }
}

```

This was a lot to take in. I had to read the code over several times because at first it was hard to see what was going on. Eventually, though, it became clear. The `accept` method knows which derivative of `Suit` it is running in, and so it knows which `visit` method to call. Simple, once you know the trick.

Then I took a closer look. All the dependencies were right. `SuitInspectionReport` had no idea what type of `Suits` it was dealing with. The `Suit` derivatives were completely decoupled from the report format. Indeed, I could add any new report I wanted simply by creating a new derivative of `SuitVisitor`. This really was like adding a new polymorphic method to the `Suit` hierarchy; without having to change the `Suits` at all!

“That’s really cool, Avery!” I said appreciatively. “Where did you learn that trick?”

“Have you heard of John Vlissides?” Avery asked?

“Of course, who hasn’t?”

“Right. I attended a talk he gave a few weeks ago. This was one of the techniques he talked about. He called it the *Visitor*³ pattern.”

“Wow, how did I miss a talk by Vlissides? You should have told me!”

“I didn’t know you then. Besides it was a private talk for my graduating class.”

“Well, I envy you. He’s a great speaker. So, he called this the Visitor pattern eh? I wonder why?”

“I don’t know. The name doesn’t seem to make a lot of sense to me.”

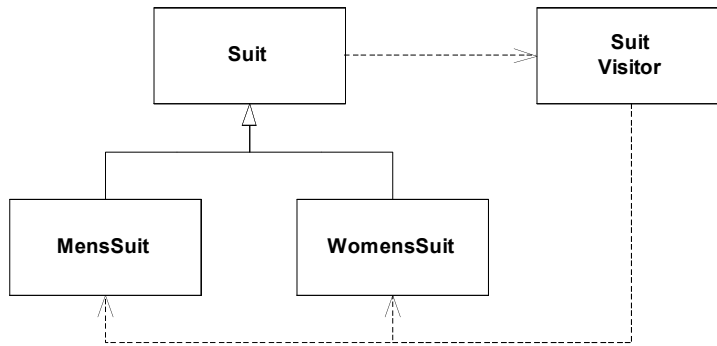
“Regardless of the name, it sure looks like a useful technique. I love the way it manages the dependencies in this situation. The report logic is so nicely decoupled from the `Suit` hierarchy.”

Avery paused for a second and then said: “It’s a good technique, but it’s not perfect. There is a nasty dependency cycle at its heart.”

“A dependency cycle?”

“Yeah. `Suit` depends upon `SuitVisitor`, which depends upon both derivatives of `Suit`. And of course the derivatives of `Suit` depend upon `Suit`.” Avery drew a little picture on his sketch pad while he talked.

³ <http://www.objectmentor.com/resources/articles/visitor>



I looked at this diagram for a few seconds and then said: “OK, I see the cycle. What I don’t see is why the cycle is a problem.”

“Do you agree that its a general rule of OO that base classes should not depend upon their derivatives?”

“Sure, everyone knows that!”

“OK, so at very least *Suit* *transitively* depends upon its own derivatives.”

“Sure, I see that. But what does that harm?”

“For one thing, it confuses build order. Imagine you are the Java compiler. You have been told to build *Suit*. Suppose this is a clean compile, so there are no *.class* files around from previous compiles. As you begin to compile *Suit* you see a reference to *SuitVisitor*, so you decide to compile *SuitVisitor* before you finish compiling *Suit*. But as you compile *SuitVisitor* you see references to *MensSuit* and *WomensSuit*, so you decide to compile them first. But as you compile them you see references to *Suit*, which you are trying to compile. What do you do?”

I thought about this for several seconds and realized that there was no way out. “I see, so there is no correct order to build these classes. No matter what order you build them in, the order must be wrong.”

“Exactly! Now in cases like *SuitVisitor*, build order isn’t very important since the classes are simple enough for the compiler to get right despite the ambiguous build order. But in more complicated cases, cycles of dependencies can cause some really nasty problems. The symptom of these problems is that you have to build the system more than once before it works. The first time you build it, the build order is wrong, and something breaks at runtime. The second, or third, or fourth time you build it, without deleting the class files in between builds, you finally, accidentally, get the order right enough to run.”

“That’s kind of scary.”

“Yes, it is. You have to be very careful with dependency cycles in Java, and make sure you eliminate all but the most trivial.”

“OK, but the *SuitVisitor* cycle isn’t harmful.”

“I didn’t say that. All I said was that the ambiguous build order wasn’t harmful. There are other reasons to avoid such cycles.”

“Whew! It’s time to get back to work; but let’s talk more about this tomorrow.”

.....

This article is dedicated to the memory of John Vlissides, co-author of *Design Patterns* and trusted friend and colleague.

To be continued...

The Craftsman: 46

Brown Bag III

Squaring the Circle

Robert C. Martin
5 January 2006

...Continued from last month.

Aug, 1944.

“How do you push thirteen quadrillion tons of loosely packed gravel and snow to one side?” Linus Pauling asked.

“I don’t think you do.” Replied Jan Oort. “It would be like trying to push six thousand cubic kilometers of feathers. There’s nothing to push against. The pusher just penetrates the pile without exerting any useful force against it.”

“That’s exactly the problem.” Pauling agreed. “If we had 50 years we could pull it aside with the gravity of a massive ship. But Clyde will be here in fifteen years. We can’t push it aside, we can’t dodge it, and we can’t destroy it.”

Wednesday, 27 Feb 2002, 1100

I was already waiting in the conference room when Avery walked in. He had Jerry with him. We greeted each other, and then got down to business.

“I hear you guys have been investigating dependency cycles.” Jerry blurted.

“Right.” I acknowledged. “Avery showed me the VISITOR pattern yesterday, and that led to a discussion of dependency cycles. He described to me how such cycles make the build order ambiguous, but he also mentioned that there were other costs.”

Jerry nodded. “Sure, there are all kinds of problems with dependency cycles. For example, if you have two components in two different jar files, and there is a dependency cycle between those components, then you cannot deploy those two jar files independently.”

Both Avery and I nodded. That made perfect sense.

“Also, any class that depends on a member of a cycle, depends upon *every* member of the cycle.”

I thought about this for a second, and then said: “You mean transitively?”

“Of course.” Jerry replied. “But transitive dependencies are dependencies. If A depends on B, and B depends on C, then A depends on C.”

“OK, I see that.” I said. “And I understand that all them members of a cycle must be compiled and deployed together. But is there any other harm?”

“The number of dependencies in a cycle” Jerry continued “is related to the square of the number of classes in the cycle.”

“Uh...” I said intelligently.

Jerry took a professorial stance. “Consider a cycle of A, B, and C. A depends on B which depends on C which depends back on A. How many dependencies are there?”

“I counted three in your description.” I said.

“Yes, but there are actually six. AB, BC, CA, and the transitive dependencies: BA, CB, AC.”

“OK, I guess.” I frowned.

“Perhaps you don’t think A really depends on C?” Jerry challenged.

“Well, it’s indirect at best.” I hedged.

“Consider that C has a method `f(int a)`, and you change it’s signature to `f(double a)`. Is it possible, even likely, that you’ll have to change the source code of B?”

“Sure.” I said. “B probably calls `C.f`, and so B is likely to need some changing. Probably some `int` inside of B will have to change to a `double`.”

“Right. Now what about A?”

“What about it?”

“Is there some `int` inside of A that might need to change to a `double`?”

“Er... Maybe.”

“In fact, this kind of backwards chaining between modules is very common! Changes made at leaves of a dependency tree often propagate back along the branches.”

“OK.” I sighed. “For the sake of this argument, let’s say that changes to C will likely cause changes to A.”

Jerry smiled, while Avery remained oddly silent. “OK, now what if there are four modules, A, B, C, and D. How many dependencies are there?”

I drew a square and connected the vertices, and counted. “Twelve, I said.”

“And for five modules?”

I imagined the pentagon in my head and said: “Twenty.”

Avery jumped in: “The general formula is $n^2 - n$.”

Jerry nodded. “Both right. So the number of dependencies in a cycle is $O(n^2)$.”

“OK, I see that, but why is that significant? Why do I care that the number of dependencies in a cycle is related to the square of the number of modules in that cycle?”

Jerry smiled. “What is a software system?”

I rolled my eyes. This topic was leaving the ship and going to some other galaxy. “A software system is a collection of classes.”

Jerry smirked and said: “That will do for the moment. Now how do you understand that system?”

A little itch started at the back of my brain. “You understand that system by learning what the classes do, and how they interrelate – Oh!”

Jerry’s smile got as wide and condescending as Jasper’s. “Yeah! Cycles are *hard* to understand! And the bigger the cycle, the harder they are. And to make matters worse, every class that depends on a cycle, depends upon every class in that cycle; so they are hard to understand too.”

“Hmmm. I said. I understand the math, but I’m not sure I agree with the premise. Is the difficulty in understanding a software system really based on the number of dependencies?”

“It’s certainly a component.” said Jerry. “I have seen systems that started out simple, but that became virtually un-maintainable in a very short time. Every time anyone tried to make a change, they broke the system in unexpected ways and in unexpected places. And for an unexpected module to break, it had to be dependent on the module that changed. I have seen development efforts start fast and furious only to grind to a near halt in a matter of months because the developers lost control of the dependencies. So yes, Alphonse, dependencies really are a significant factor in the understanding and maintainability of a system.”

We all sat there looking at each other. Jerry was smiling, Avery was withdrawn, and I was pensive. Finally, I said: “If this is true, then it is a guilt-edged priority to keep dependency cycles out of our systems.

Jerry's smile slowly faded. "In Mr. C's organization, it *definitely* is. Especially cycles between packages!"

"How do you prevent them?" I asked.

"There are tools that find them for you. JDepend¹ will hunt through your whole Java project and find dependency cycles. And there is a plugin² for FitNesse that uses JDepend so that you can make acceptance tests that check for cycles."

I thought about that for a minute. "Cool! That means the build will fail if you have a dependency cycle. If you do that, you'll simply never get cycles because you'll have to fix them as soon as they happen!"

"Yeah. We haven't set that up yet on Dosage Tracking, but we will."

"OK, OK." Said Avery, suddenly getting into the conversation, "But what about the VISITOR? the VISITOR creates a cycle of dependencies. Is that bad?"

Jerry visibly gathered his thoughts into this new direction and then said: "Well, usually the cycles from the VISITOR pattern are so small that they don't cause much harm."

"But what if you wanted to break them anyway?" Avery said.

"Well, you could use the ACYCLIC VISITOR."

Avery and I both said: "The what?"

Jerry laughed and said: "Here, let me show you." And he pulled up the example of the `SuitInspectionReport` that we had used yesterday. He made a few changes to it, and then said: "There. That's the acyclic version of the VISITOR."

The first thing to notice is that the `SuitVisitor` has become a *degenerate interface* having no methods or variables. Interfaces like this are sometimes called *marker interfaces*.

```
public interface SuitVisitor {}
```

Next, notice that there are two new interfaces, one for the `MensSuit`, and one for the `WomensSuit`.

```
public interface MensSuitVisitor {  
    void visit(MensSuit ms);  
}
```

```
public interface WomensSuitVisitor {  
    void visit(WomensSuit ws);  
}
```

The `SuitInspectionReportVisitor` is the same as before, except that it now implements all three visitor interfaces.

```
public class SuitInspectionReportVisitor implements SuitVisitor,  
                                                    MensSuitVisitor,  
                                                    WomensSuitVisitor {  
  
    public String line;  
  
    public void visit(MensSuit ms) {  
        line = String.format("%d MS A:%d, B:%d, C:%d",  
                             ms.barcode, ms.ipa, ms.ipb, ms.ipc);  
    }  
  
    public void visit(WomensSuit ws) {  
        line = String.format("%d WS A:%d, B:%d",  
                             ws.barcode, ws.ipa, ws.ipb);  
    }  
}
```

¹ www.clarkware.com

² <http://butunclebob.com/ArticleS.UncleBob.StableDependenciesFixture>

```

    }
}

```

Finally, the `MensSuit` and `WomensSuit` classes implement the `accept` method a little differently.

```

public class MensSuit extends Suit {
    int ipa;
    int ipb;
    int ipc;

    public MensSuit(int barCode) {
        super(barCode);
    }

    public void accept(SuitVisitor v) {
        if (v instanceof MensSuitVisitor)
            ((MensSuitVisitor)v).visit(this);
    }
}

public class WomensSuit extends Suit {
    int ipa;
    int ipb;

    public WomensSuit(int barCode) {
        super(barCode);
    }

    public void accept(SuitVisitor v) {
        if (v instanceof WomensSuitVisitor)
            ((WomensSuitVisitor)v).visit(this);
    }
}

```

The `SuitInspectionReport` uses the `SuitInspectionReportVisitor` in exactly the same way as before.

```

public class SuitInspectionReport {
    public String generate(SuitInventory inventory) {
        StringBuffer report = new StringBuffer();
        SuitInspectionReportVisitor v = new SuitInspectionReportVisitor();
        for (Suit s : inventory.getSuits()) {
            s.accept(v);
            report.append(v.line).append("\n");
        }
        return report.toString();
    }
}

```

I looked at this code for awhile and then said: “This breaks the cycle all right; and it retains all the benefits of the original VISITOR. So why wouldn’t we always use this?”

Jerry scratched his head and said: “Well in a case like this the cycle in the regular VISITOR just isn’t very harmful. Moreover, the VISITOR is a bit faster and simpler than the ACYCLIC VISITOR. So I’d probably opt for the simpler case. However, if there were a lot of classes and dependencies involved; I’d be tempted to break the cycle.”

Our time was up, so we walked out of the conference room and back to our workstations. Jerry strode quickly ahead of us. When he was out of earshot I heard Avery murmur: “He ruined everything!”

.....

To be continued...

The Craftsman: 47

Brown Bag IV

Jinx!

Robert C. Martin
7 February 2006

...Continued from last month.

Sept, 1944.

President Henry Wallace sat in private and buried his face in his hands. "Perhaps we deserve to be destroyed," he said to an uncaring universe. The extermination of the human race seemed assured on several fronts. Clyde was coming in fifteen years, and there didn't seem to be anything to be done about it. The Third Reich was systematically exterminating whole races of people. Stalin continued his purges against any hint of contrary ideologies. The American monopoly on atomic weapons could not last much longer -- and then there would be war and destruction on a scale that only Clyde could rival. Indeed, perhaps Clyde was simply God's undertaker; sent to bury an Earth that was already dead.

And then there was a knock at the door. "Mr. President? Dr. Von Braun is here to see you."

Thursday, 28 Feb 2002, 1100

Avery and I arrived together at the small conference room that we used for our daily discussions. I assumed that the scowl on his face had something to do with the fact that Jerry and Jasmine were following right behind. We all got seated and then Jerry said: "OK, whose turn is it today?" Avery's scowl deepened, but he remained silent.

"Usually Avery and I take turns." I said, "But Avery didn't get his turn yesterday, so..."

Avery sent a cold glance at Jerry, but then said: "Never mind Alphonse, it's your turn. Go ahead. I don't have anything prepared."

"OK, actually I don't have much prepared either, but I do have a question. Has anyone seen the new `enum` feature in Java 5? Apparently you can define `enum` constants that have variables and methods."

Avery suddenly became very alert. "Can you write an example on the wall, Alphonse?"

"Sure", and so I used my finger to write the following code on the wall panel.

```
public class Taco {  
    enum Color {red, green, burntumber}  
}
```

"OK", I said, "we all know that Java 1.5 has `enums` that look like this. They are essentially integer constants."

Avery jumped to the wall and started writing and talking at the same time. "Yeah, before 1.5 we

would have done this.”

```
public class Taco {
    public static final int RED = 0;
    public static final int GREEN = 1;
    public static final int BURNT_UMBER = 2;
}
```

The wall panel recognized Avery’s writing and made sure his code was in a separate sandbox from mine, so that they didn’t conflict.

“Right.” I said. But that’s where the similarity ends. In Java 1.5 an enumeration is actually a class, and each of the enumerators is a derived class. These classes can have variables. For example:” And I modified the code on the screen with my finger.

```
public class Taco {
    enum Color {red, green, burntumber}

    enum Salsa {
        mild(10, Color.red),
        medium(1000, Color.green),
        hot(100000, Color.burntumber);

        public final int scovilles;
        public final Color color;

        Salsa(int scovilles, Color color) {
            this.scovilles = scovilles;
            this.color = color;
        }
    }
}
```

“Hot damn!” said Jasmine. “That’s pretty cool. Look, Jerry, the `enum` has a constructor and instance variables!”

Jerry looked at the wall for a second and said, “Yeah, that’s pretty useful. This means that a constant can have more than one value. So consider this representation for a Red/Green/Blue color.” Jerry started writing on the wall too.

```
public enum RGB {
    red(0xff, 0x00, 0x00),
    green(0x00, 0xff, 0x00),
    blue(0x00, 0x00, 0xff),
    white(0xff,0xff,0xff);

    public final int r;
    public final int g;
    public final int b;

    RGB(int r, int g, int b) {
        this.r = r;
        this.g = g;
        this.b = b;
    }
}
```

I muscled my way back up to the wall and said: “Right, and this allows you to do interesting things like this.”

```
public class Taco {
    enum Color {red, green, burntumber}

    enum Salsa {
        mild(10, Color.red),
        medium(1000, Color.green),
        hot(100000, Color.burntumber);

        public final int scovilles;
        public final Color color;

        Salsa(int scovilles, Color color) {
            this.scovilles = scovilles;
            this.color = color;
        }
    }

    public static boolean isTooHotForMe(Salsa s) {
        return s.scovilles > 30000;
    }
}
```

“Oh wow!” Said Jasmine. “That means we can write code that makes decisions about enums without knowing any of the enumerations!”

“Huh?” said Jerry. “What do you mean?”

“Well, just look at what Alphonse wrote!” She urged. “The `isTooHotForMe` method will work for any enumerator within the Salsa enumeration. Even if we add new enumerators later, the `isTooHotForMe` method will continue to work unchanged.”

“The Open Closed Principle!” I blurted.

“Yeah!” Said Avery. “We can add, change, or remove enumerations, without affecting the `isTooHotForMe` method!”

Jerry looked intently at the wall for a few seconds, and then said: “Hmmm, that implies a whole new way of thinking about constants and enums. Sometimes we want to NOT know what enumerators there are, and rather depend on the internal structure. For example, we could write an `isBlue` function as follows:

```
class SomeClass {
    public static boolean isBlue(RGB color){
        return color.b > 2*(color.r + color.g);
    }
}
```

Jasmine rolled her eyes and jabbed Jerry in the ribs with her elbow. “You’ve got color on the brain, don’t you buddy-boy.” Then they smiled at each other just a little too long.

I shook my head and said: “That method might be better off in the enumeration itself. Look at this.”

```
public enum RGB {
    red(0xff, 0x00, 0x00),
    green(0x00, 0xff, 0x00),
    blue(0x00, 0x00, 0xff),
    white(0xff,0xff,0xff);

    public final int r;
    public final int g;
```

```

public final int b;

RGB(int r, int g, int b) {
    this.r = r;
    this.g = g;
    this.b = b;
}

    public boolean isBlue(){
    return b > 2*(r + g);
    }
}

```

“Does that work?” Avery demanded.

“Sure, I said, and wrote a test to prove it.”

```

public class RGBTest extends TestCase {
    public void testIsBlue() throws Exception {
        assertTrue(RGB.blue.isBlue());
        assertFalse(RGB.red.isBlue());
        assertFalse(RGB.green.isBlue());
    }
}

```

“Wow.” Said Jerry.

“Wow.” Said Jasmine.

“Jinx!” They both said, giggling.

“So we could do the same thing to the `Taco.Salsa` enum too, right?”, asked Avery. “Let me try.” And he started changing things on the wall.

```

enum Color {
    red, green, burntumber}

enum Salsa {
    mild(10, Color.red),
    medium(1000, Color.green),
    hot(100000, Color.burntumber);

    public final int scovilles;
    public final Color color;

    Salsa(int scovilles, Color color) {
        this.scovilles = scovilles;
        this.color = color;
    }

    public boolean isTooHotForMe() {
        return scovilles > 30000;
    }
}

```

“And here are the tests.” He continued.

```

public class TacoTest extends TestCase {
    public void testIsTooHot() throws Exception {
        assertTrue(Taco.Salsa.hot.isTooHotForMe());
        assertFalse(Taco.Salsa.medium.isTooHotForMe());
        assertFalse(Taco.Salsa.mild.isTooHotForMe());
    }
}

```

```
}
```

Avery stood with a silly grin as he stared at the green test indicator bar on the wall. “That’s really cool.” He finally said.

“Yes.” I agreed. “But it’s better than that. Those methods can be polymorphic.”

“What?” said Jerry.

“What?” said Jasmine.

“Jinx!” (giggle).

Avery and I rolled our eyes at each other.

“Yeah.” Look at this.

```
public class Taco {
    enum Color {
        red, green, burntumber}

    enum Salsa {
        mild(10, Color.red) {
            public int sips() {
                return 0;
            }
        },
        medium(1000, Color.green){
            public int sips() {
                return 1;
            }
        },
        hot(100000, Color.burntumber){
            public int sips() {
                return 5;
            }
        }
    };

    public final int scovilles;
    public final Color color;
    public abstract int sips();

    Salsa(int scovilles, Color color) {
        this.scovilles = scovilles;
        this.color = color;
    }

    public boolean isTooHotForMe() {
        return scovilles > 30000;
    }
}
```

“Does that work?!” Avery demanded once again.

Everyone looked at him for a second, and then started to laugh. After a few seconds, even Avery laughed.

“Sure, here are the tests.” I said.

```
public void testSips() throws Exception {
    assertEquals(0, Taco.Salsa.mild.sips());
    assertEquals(1, Taco.Salsa.medium.sips());
    assertEquals(5, Taco.Salsa.hot.sips());
}
```

What happened next I can only describe in code:

```
wow() * 3;  
jinx() * 3;  
laugh() * 4;
```

“OK.” Jerry said with some finality. “This has gotten out of control. I think it’s time we got back to work.”

There were nods of agreement, and everyone headed for the door.

“But wait!” I said. “I still haven’t asked my question.”

“OK, Hotshot.” Jasmine responded. “Hurry it up. What’s your question.”

I stood my ground and looked at the three of them. Then I said: “What would you use this for?”

The Craftsman: 48

Brown Bag V

Statenum

Robert C. Martin
11 March 2006

...Continued from last month.

Sept, 1944.

"Dr. Oppenheimer, General Groves, thank you for coming." The morning sunlight streamed through the southeast windows of the Oval Office, providing a cheery glow that matched President Wallace's mood. "Yesterday I had a most interesting meeting with Dr. Von Braun. It is his opinion that we can build a fleet of huge atomic powered interplanetary space ships to rescue a remnant of humanity from the impact of Clyde." Oppenheimer squirmed uncomfortably in his chair, but Groves kept his attention squarely on the President. "Gentlemen, I don't pretend to know if he is right. What I do know is that we need more bold ideas like that. I want you to scour project Nimbus and find them; the bigger and bolder the better!"

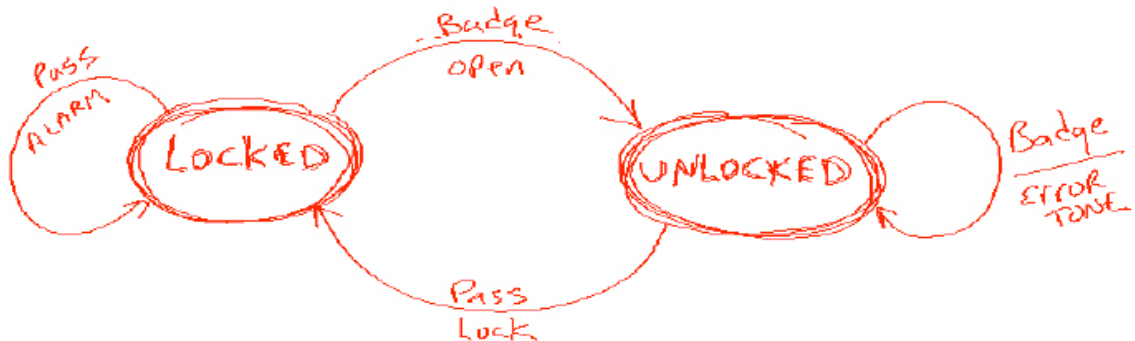
Friday, 1 Mar 2002, 1102

I was running a couple of minutes late for our regular 11:00 meeting. When I walked in the room there were quite a few people in that little conference room. Jerry, Avery, Adelaide, and Jasper were watching Jasmine who was just starting to draw something on the wall.

Jasmine fixed me with those deep green eyes of her and said: "It's about time, Hotshot. I was just about to answer that question you asked yesterday."

"Sorry." I murmured as I squeezed in beside Avery. He rolled his eyes meaningfully while subtly cocking his head towards Jasmine. I gave him a pained smile and turned my attention back to her.

She spoke while she drew the following picture on the wall. "OK, here is the finite state machine for the security gate up on the bridge."



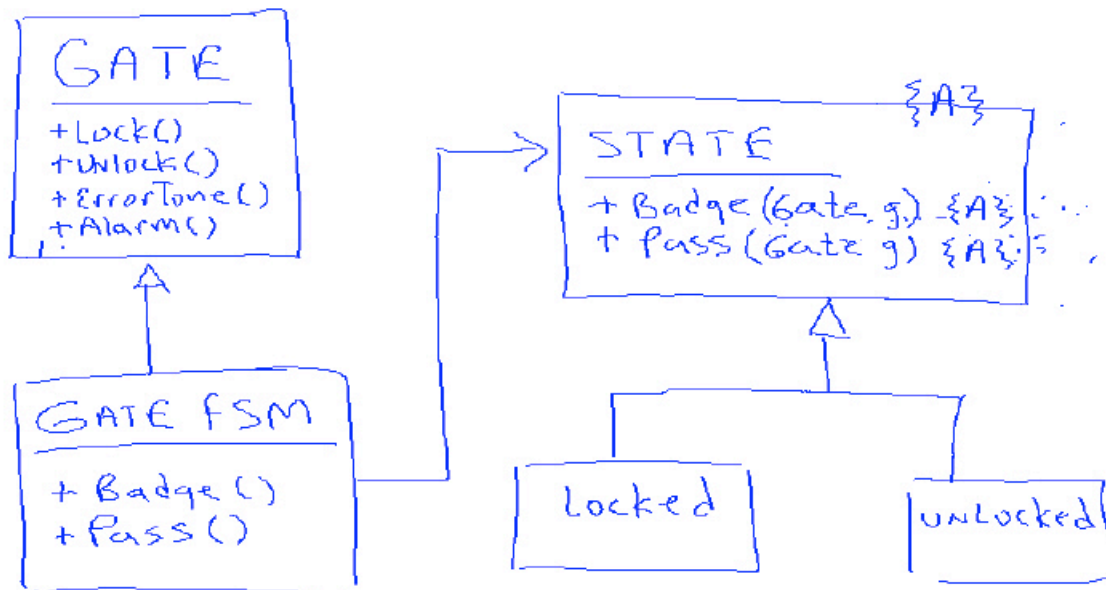
Avery said: "Excuse me, but that's not really how it works, it..."

“Yeah, yeah, I know.” Jasmine replied testily, “But I don’t have enough wall space or time to draw the complete FSM. Just bear with me. This *simplified* machine starts out in the LOCKED state. If you wave a valid badge at it, the gate opens and the FSM enters the UNLOCKED state. Once you pass through the gate, it automatically closes and locks and the FSM goes back to the LOCKED state.

“The two side transitions show the error cases. The one on the right shows the case of someone continually waving their badge at an open gate. The gate sounds an error tone in an attempt to get the nitwit to stop waving his badge and go through the gate. The transition on the left shows the case where someone forces entry onto the bridge. An alarm sounds.”

Jasmine spun away from the wall to look at her audience. Her long black hair flared out and then fell perfectly around her shoulders. “Everybody got it?” She demanded rather than asked. We all nodded meekly.

“OK,” she continued, “now we all know that SMC¹ would generate this using the State pattern. It would create this nice class hierarchy.” And she drew the following:



We were all familiar with this so we all nodded meekly as she swung around. She frowned playfully and then went on. “One problem has always been that we had to manage the instances of the Locked and Unlocked derivatives in static variables to prevent us from having to continuously create and destroy them. Another problem has been that we could not make the methods and variables of GateFSM private since they are manipulated by the derivatives of State.”

Jasmine then pulled up some code that she had written previously for this talk.

```

public class Gate {
    void lock() {}
    void unlock() {}
    void errorTone() {}
    void alarm() {}
}

public class GateFSM extends Gate {
    State state = State.locked;

    public void badge() {
        state.badge(this);
    }
}
  
```

¹ <http://www.objectmentor.com/resources/downloads/index>

```

    public void pass() {
        state.pass(this);
    }
}

```

```

public abstract class State {
    public abstract void badge(GateFSM g);
    public abstract void pass(GateFSM g);

    public static State locked = new Locked();
    public static State unlocked = new Unlocked();
}

```

```

public class Locked extends State {
    public void badge(GateFSM g) {
        g.unlock();
        g.state = State.unlocked;
    }

    public void pass(GateFSM g) {
        g.alarm();
    }
}

```

```

public class Unlocked extends State {
    public void badge(GateFSM g) {
        g.errorTone();
    }

    public void pass(GateFSM g) {
        g.lock();
        g.state = State.locked;
    }
}

```

“Notice the public static variables in the State class, and the non-private methods and variables of Gate and GateFSM.

“Excuse me,” wheedled Avery, “But you could make those methods and variables private by using inner classes. Here, let me show...”

“SIT DOWN! Yes, I know, I know. I’m just trying to make a point about the code that SMC generates. Bear with me please!”

Avery sat down and started to sulk.

Jasmine gave us all a weak smile, actually it was more like a grimace, and then continued. “Here’s how we can implement it using the new enum syntax that Alphonse showed us yesterday.

```

public class Gate {
    private enum State {
        LOCKED {
            void badge(Gate g) {
                g.unlock();
                g.itsState = UNLOCKED;
            }

            void pass(Gate g) {
                g.alarm();
            }
        },
        UNLOCKED {
            void badge(Gate g) {
                g.errorTone();
            }
        }
    }
}

```

```

        void pass(Gate g) {
            g.lock();
            g.itsState = LOCKED;
        }
    };

    abstract void badge(Gate g);
    abstract void pass(Gate g);
}

private State itsState = State.LOCKED;

private void lock() {}
private void unlock() {}
private void errorTone() {}
private void alarm() {}

public void badge() {
    itsState.badge(this);
}

public void pass() {
    itsState.pass(this);
}
}

```

“First, you’ll notice that it all fits in one class. It’s small, expressive, we don’t have any `static` variables to manage, and all the methods of `Gate` can be `private`. I think this is a pretty cool use for polymorphic enums don’t you?

We all nodded meekly; except for Avery who kept on sulking.

Jerry stood up and said: “Yes, that’s very interesting. The enumeration is like a class, and each enumerator is like a static instance of a derivative of that class. It’s kind of hard to get your head around, but it’s definitely interesting.”

Then Jasper got into the act. “Yes ma’am, that’s pretty interesting Jazzie.” Jasmine’s face reddened, and her jaw clenched, but she held her peace as Jasper gushed on. “But I think the aviator’s got a point. The problems with the SMC code can be solved without having to resort to polymorphic enums. Aviator suggested inner classes; but I think nested switch-case statements would work better.” And he quickly wrote the following code on the wall.

```

public class Gate {
    private enum State {LOCKED, UNLOCKED}
    private enum Event {BADGE, PASS}

    private State state = State.LOCKED;

    private void lock() {}
    private void unlock() {}
    private void errorTone() {}
    private void alarm() {}

    public void badge() {
        processEvent(Event.BADGE);
    }

    public void pass() {
        processEvent(Event.PASS);
    }

    private void processEvent(Event event) {

```

```

switch (event) {
  case BADGE:
    switch (state) {
      case LOCKED:
        unlock();
        state = State.UNLOCKED;
        break;
      case UNLOCKED:
        errorTone();
        break;
    }
    break;
  case PASS:
    switch (state) {
      case LOCKED:
        alarm();
        break;
      case UNLOCKED:
        lock();
        state = State.LOCKED;
        break;
    }
    break;
}
}
}
}

```

“See,” said Jasper smugly, “no static variables, no private methods, nice and simple.”

“You call that simple?” Jerry bantered. “Those nested switch statements make my head hurt. Man, I’ve maintained big FSMs written that way, and they are no fun at all, let me tell you.”

“Yeah,” Jasmine chimed in, “And all that logic is sitting there in one method that’s bound to grow and grow and grow. Ick!”

“Jazzie...did you just say Ick?”

“Stop calling me Jazzie, Jasper. I’ve told you before. Yeah, ick! That code is going to rot before you know it.”

They kept on like this for a minute or two. Avery left the room in a huff. I finally stood up and tapped on the wall until I had everyone’s attention.

“Guys,” I said, grinning, “I haven’t seen one test. How do we know that any of these FSMs actually work?”

They all groaned and declared the meeting over.

On the way back, Jerry walked up next to me and, shaking his head, said: “I don’t know what we’re going to do about Avery.”

The Craftsman: 49

Brown Bag VI

Abstract Factory

Robert C. Martin
18 October 2006

...Continued from last month.

October, 1944.

"Dr Ulam's idea is simplicity itself!" Dr. Von Braun said to Dr. Robert Goddard as they ate lunch at the Nimbus facility. "A nuclear powered spaceship can be propelled simply by dropping atomic bombs out the back and blowing them up!"

"I see," said Goddard, "That seems a bit drastic. I presume you'd use a shield, a kind of big pusher-plate to absorb the momentum of the blast and transmit it to the whole ship."

"Precisely, Robert! We use one-kiloton bombs at a rate of one per second or so. A few thousand such bombs could propel the entire Nimbus facility, and all it's inhabitants, to Saturn and back!"

"Yes, I suppose that when you use atomic bombs as a propellant lifting power is not a problem. Are you really thinking about going to Saturn?"

"Oh, no that's just something we could do if we wanted to. However we are going to try to land on Mars by this time next year."

Goddard was taken aback. He stared at Von Braun in disbelief. After a few seconds he said: "That's quite, er..., aggressive, isn't it?"

Von Braun just looked soberly at Goddard and said nothing. Goddard finally conceded: "Yes, I suppose we are in a bit of a rush." Von Braun just nodded.

"Why Mars?"

"It's a reasonable test of the ship's capabilities, and it might just be our new home."

Goddard nodded. "Yes, canals, vegetation, perhaps even an ancient civilization. I hope they don't resent us just showing up like this."

"There's nobody there, Robert, they all died centuries ago, millennia ago. And if someone is there, we'll just have to deal with that at the time. You can bet we'll be prepared."

"Hmmp." Goddard acknowledged. Then he looked thoughtful for a moment and said: "Have you considered, Werner, that if you can propel a ship with atomic bombs, that you might be able to shift the orbit of Clyde in the same manner?"

Von Braun began to shake his head but then he stopped and stared at Goddard, and his eyes got very large.

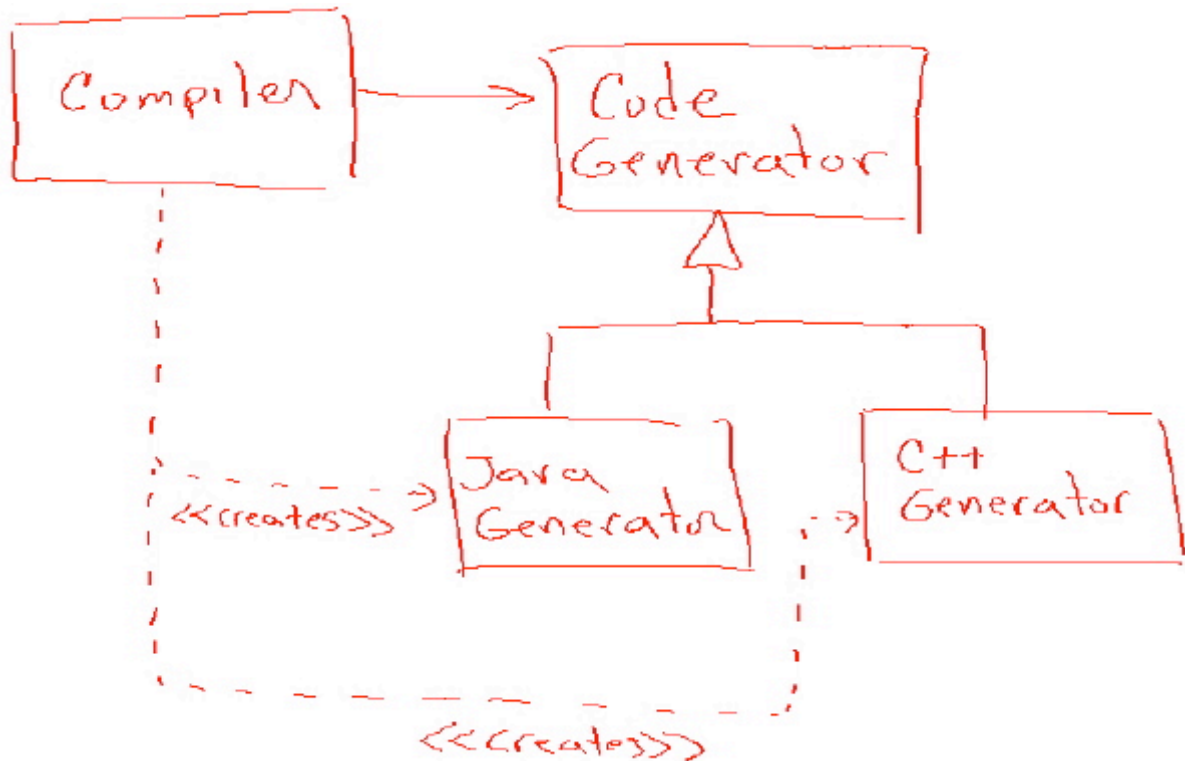
Monday, 4 Mar 2002, 1100

I was determined not to repeat last Friday's late entrance; so I walked into conference room along with everyone else, and noted that Adelaide was getting ready to present. Our little group had grown quite a bit.

There were people here I had never met before. I counted 25 people in the room. Even Jean was there.

Adelaide rapped for attention, and the crowd settled down. She was clearly nervous, but wasn't in a mood to back down. She steeled herself and began.

"OK everybody, Jasmine asked me to talk about the ABSTRACT FACTORY pattern. She and I just used this pattern in the SMCRemote project, and so I guess I can say a few words about it. Here is what I know:" Adelaide began to draw on the wall with her finger.



"Here is the old design of The SMC compiler, except that we have added a feature that allows it to generate both Java and C++ code. As I'll show you in a second, we are proposing a plug-in architecture so that new languages can be added to the compiler simply by creating new derivatives of the CodeGenerator class. Unfortunately, as you can see in the current design, in order for the Compiler class to create the appropriate instance, it has to have a direct dependency on the derivatives.

"So we propose to use the ABSTRACT FACTORY pattern to break this dependency and insulate the Compiler class from any new derivatives of the CodeGenerator."

It was clear that Adelaide had rehearsed this pitch a few times. She was a bit too smooth, and going a bit too fast. Jerry must have noticed the same thing because as Adelaide turned around to erase the diagram he stopped her with an obvious question."

"Adelaide, why is it a problem that Compiler knows about JavaGenerator and C++Generator? It seems to me that those dependencies are perfectly appropriate. After all, the Compiler class just calls new on those classes and then uses them through the CodeGenerator interface, doesn't it?"

Adelaide glanced uneasily at Jasmine, but then took a breath and answered Jerry directly.

"We don't want any kind of dependency from Compiler to the derivatives of CodeGenerator because we don't want to have to recompile Compiler any time one of the derivatives changes. The CodeGenerator class is there to insulate Compiler from such changes. It would be a shame to waste that insulation simply because of the new keyword."

I wasn't sure I bought that argument, so I asked:

"Why would changes to the derivatives force a recompile of the Compiler class?"

Adelaide looked a little flustered and glanced meaningfully at Jasmine. So Jasmine stood up to rescue her and gave me a stern look.

“OK, Hotshot, we talked about this during the VISITOR discussion last week. So I know you know the answer. But for the benefit of everyone else in the room, our build system recompiles modules based upon the modification dates of files. Since `Compiler` depends on `JavaGenerator`, if `JavaGenerator.java` has a later date than `Compiler.class`, the build system will automatically rebuild `Compiler`. OK Alphonse?”

“Sure, er, thanks.”

One of my old classmates, Alex, stood up and said: “OK, sure, but you could fix that by just using the name of the class and `class.forName` in the `Compiler`.”

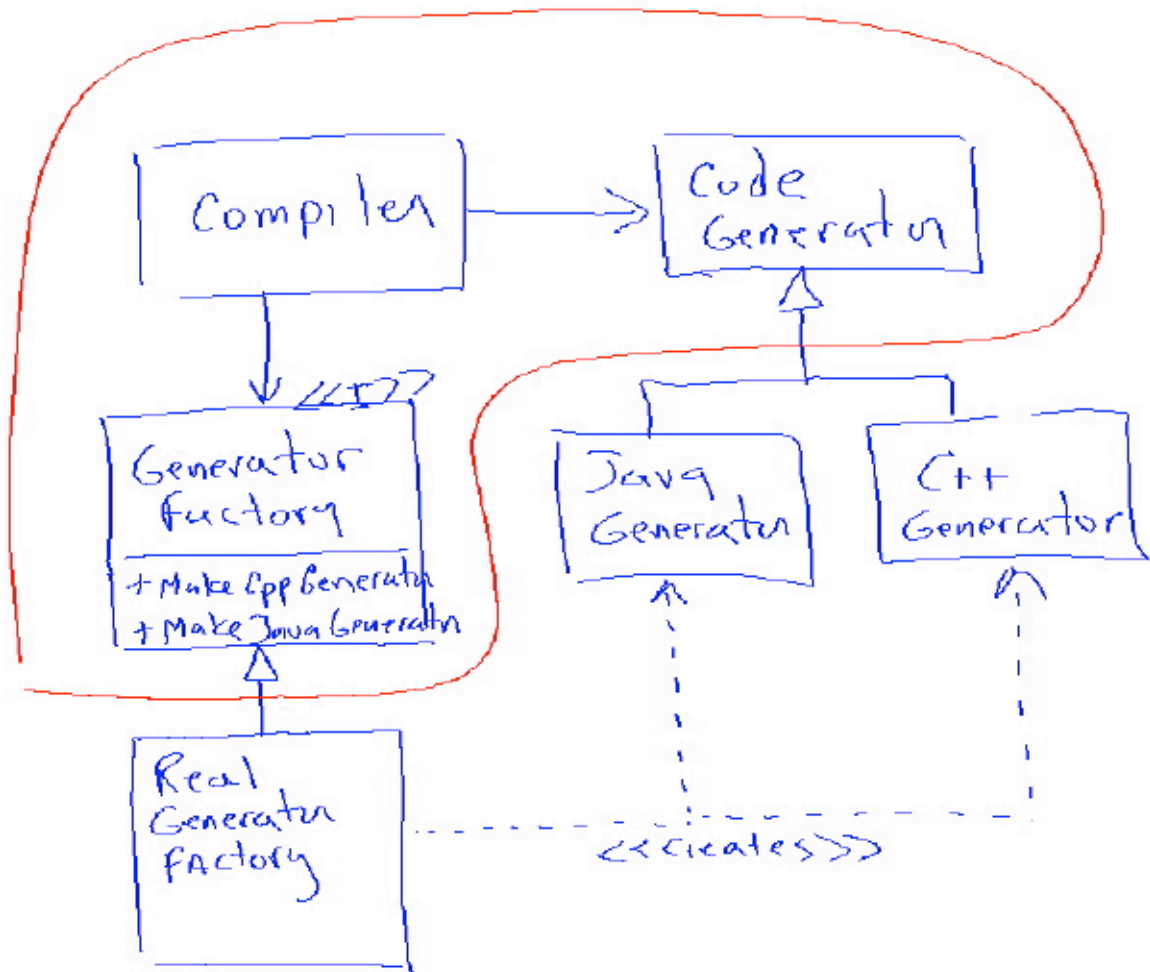
“Yeah, we could.” Replied Jasmine. “But we don’t want that kind of detailed mechanism inside the `Compiler` class. We are trying to *separate concerns*.” Jasmine stretched those last two words out with a sarcasm that forced Alex to sit down again.

Joseph stood up and said: “Jasmine, Alex asked a reasonable question, you don’t need to shut him down like that.”

Jasmine sighed and her shoulders drooped a bit. Then she said: “OK, you’re right Joseph. Alex, I’m sorry I was abrupt. Our goal here is to keep `Compiler` from knowing how the derivatives are created, or, in fact, anything detailed about the derivatives. We want this because we think the mechanism for creating the derivatives may change over time, and we don’t want those changes to impact on the `Compiler` class. We want the `Compiler` to just be a compiler, and nothing else.”

Jasmine sat down and I caught a glimpse of her rolling here eyes at Jerry.

Adelaide stood back up, waved at the wall to erase the last diagram, and then drew the following:



Then she started her practiced banter again. “This is our proposed solution. It is classic example of the ABSTRACT FACTORY pattern. Notice the GeneratorFactory interface. The Compiler class calls one of the two makeXXXGenerator methods of this interface. In response the RealGeneratorFactory creates the appropriate instance and returns it to the Compiler. Now notice the red line...”

“I’m sorry dear.” Jean interrupted. “But I’m afraid that all this UML is giving me a bit of a headache. I guess I’m just old-fashioned enough to want to see some code. You’re a terribly sweet girl, would you mind showing me the code?”

By the look on Adelaide’s face, I think it was becoming clear to her that she was never going to get through her canned presentation. To her credit, she didn’t panic, or even glance over at Jasmine for more help. Instead, she paused for an instant, and then opened up a TextMate window on the wall. Using the wall’s virtual keyboard she typed:

```

public interface GeneratorFactory {
    public CodeGenerator makeCppGenerator();
    public CodeGenerator makeJavaGenerator();
}

public class RealGeneratorFactory implements GeneratorFactory {
    public CodeGenerator makeCppGenerator() {return new CppGenerator();}
    public CodeGenerator makeJavaGenerator() {return new JavaGenerator();}
}
  
```

“Does that help?” she asked politely.

Jean didn’t even take a second to look at the code. She just smiled indulgently and said, “Yes, dear,

that's just fine."

Avery whispered in my ear: "Jean just wanted to be sure that Adelaide knew how to write the code."

I tended to agree, but did not say anything in response.

"OK, now look at the red line," Adelaide continued. "The red line delimits our plug-in architecture. Everything outside the red line is a plug-in. Everything inside the red bubble is our core engine. Notice how all the dependencies that cross the red line point inwards! This means that changes to elements outside the bubble don't affect the elements inside the bubble. For example, if one of the code generator derivatives changes, the bubble does not need to be recompiled."

"A fat lot of good that does!" Avery had stood up and was speaking loudly and forcefully. "This structure has the same problem that the VISITOR has. There's a dependency cycle between the `GeneratorFactory` and the `CodeGenerator` derivatives. Every time you add a new generator derivative you have to add a new `make` method to the `GeneratorFactory` interface, which means your red-lined bubble *does, in fact*, need to be recompiled. So this whole pattern is just worthless!"

Avery stood there glaring while Adelaide looked to be on the verge of tears. Jasmine shot to her feet. "Avery, you little..."

But before Jasmine could make a move, Jean stood up stiffly and said: "Avery dear, would you please come with me." Jasmine's eyes widened and her eyebrows did a quick raise, and then she sat down again. Avery's face went white. Jean beckoned him towards the door, and the two left in uneasy silence.

The uneasy tension remained for a few moments, and then Jasper said: "Gee, wasn't it just last week when these sessions were fun!" The tension broke and everyone laughed and relaxed...a little.

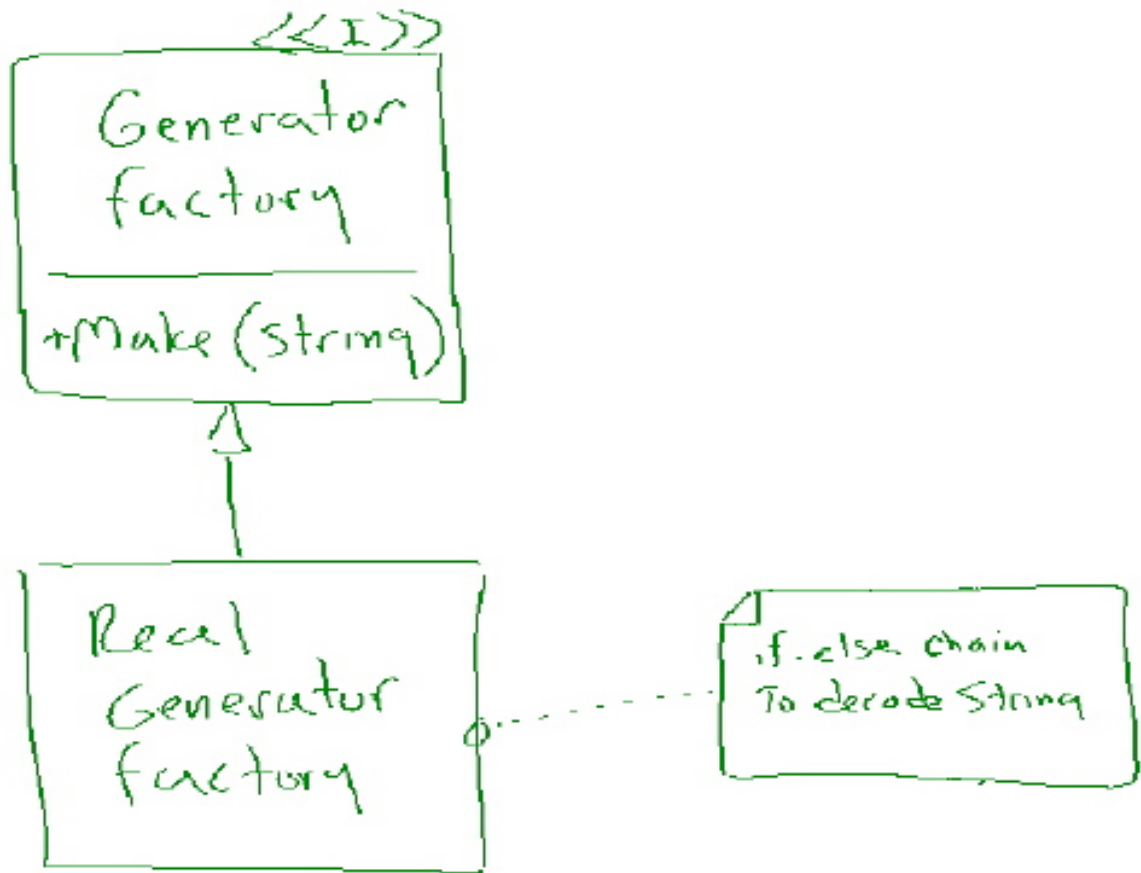
Jerry stood up. "Avery could have said it better, but he had a point, didn't he? You *do* have to change and recompile the core engine every time a new code generator is added."

Adelaide was still staring at the deck, biting back tears, and couldn't seem to answer. Jasmine eventually stood up and said: "Yes, that was an intentional part of the design. We expect that new languages won't be frequent enough to bother protecting ourselves from. What we are trying to insulate the core from are the maintenance changes to the existing code generators."

Jared stood up and said: "OK, that's fair enough, but what if you did want to protect yourself from new languages? What would you do then? How could you keep from recompiling the core engine every time someone wrote a new code generator derivative?"

There was silence in the room for a few seconds, and then I stood up.

"I've been thinking about this for the last few days because we have a similar problem in the Dtrack project. We don't want to recompile our core engine every time a new kind of spacesuit is created." I waved at the wall to erase it, and then drew the following:



“And just in case Jean come back in I’ll write the code.”

```

public interface GeneratorFactory {
    public CodeGenerator makeGenerator(String type);
}

public class RealGeneratorFactory implements GeneratorFactory {
    public CodeGenerator makeGenerator(String type) {
        if (type.equals("C++")) return new CppGenerator();
        if (type.equals("Java")) return new JavaGenerator();
        return null;
    }
}
  
```

“Now imagine that the strings “C++” and “Java” are command line arguments to the Compiler class. As you can see, we can add new code generators without ever touching, recompiling, or in any other way impacting the core engine.”

Jasmine said: “But, Alphonse, that’s not type safe! What if you misspell the string?”

“Wouldn’t your unit tests catch that misspelling?”

“Uh…”

Jerry said: “I think Jasmine’s point is that we’d like the compiler to check this for us.”

“Why?” I replied? “Again, won’t you have unit tests that make sure you aren’t misspelling?”

“OK, but this is a command line argument!”

“True, but if you misspell a command line argument shouldn’t you expect to get an error? In this case

the Compiler engine could say: 'Ruby is not a supported language.'

“OK, fine, but what if you call the Compiler from a script or another program, and there was a spelling error in one of those programs. The compiler wouldn't warn you.”

“True, but again, your unit tests would.”

“Er...”

This was fun. I seem to have stumbled onto an idea that neither Jerry nor Jasmine had thought about. Indeed, all of the journeymen in the room had intent looks on their faces.

“Why is this such an issue?” I asked.

Jasper spoke up. “Well, Fonze, er... Alphonse. I don't know about the others, but you've made me wonder whether or not type safety is as valuable as we had been thinking. This unit testing stuff we've been doing over the last couple of years may make type safety redundant.”

I saw a few heads nod, and several others shake. But as we left the room the mood remained thoughtful.

The Craftsman: 51

Brown Bag VIII

Ruby Visitor

Robert C. Martin
2 Aug, 2007

December 1, 1944

The November elections resulted in a landslide victory for Henry Wallace against Dewey. The surging economy, and the near bloodless thwarting of the Axis invasion of Mexico were huge factors in his popularity. Dewey had tried to use the frightening growth of the national debt as a platform for fiscal restraint, but the people weren't in a mood to hear about belt-tightening.

The general public was not yet aware of the Turing's conclusion about Clyde. They did not know that they had a little over 14 years to live. But there had been enough leaks, rumors, and innuendos that it was clear that something big was up. The atmosphere in the nation felt like the hour before a big storm.

Wednesday, 6 Mar 2002, 11:00

I'm not sure why they call these meetings "Brown bags". Apparently it's something from the old days at Los Alamos. In any case, over 30 people filed into the conference room for this one. The speaker was someone I hadn't met before. His name was Dave Chelimsky.

As I walked into the room I saw Avery and Jean sitting together in the back. Avery's expression was flat, and he made no eye-contact. Jasmine and Jerry were on the other side of the room, also in the back, talking intensely and happily about something. Jasper was close to the front, clearly anticipating this talk. Not surprisingly there was an empty seat next to him. I considered leaving it empty, but I had a notion that sitting next to him could be interesting today.

The rear wall was populated by about a half dozen virtual participants. Their faces stared out from the windows that held their virtual session. This was a first. We hadn't had virtual participants at any of our previous brown bags.

The room grew silent and the speaker began. Chelimsky's style might be described as intense nonchalance. It was clear that he cared deeply about his topic; but his demeanor was almost apologetic for his passion – as if to say: "I care about this, but I understand if you don't."

"I understand that you have been talking about Ruby lately." He stared out at us a smiled as if Ruby were a swear word that everyone was shocked to hear. "I thought it would be interesting if we looked at a common design pattern in both Java and Ruby, and compared the differences. And what better design pattern to study than Visitor?" Again, that smile, as though he was sharing some inside joke with us.

"The Visitor pattern can be really useful when you want the decoupling of polymorphism without the coupling of inheritance. Consider, for example, a simply hierarchy of employees. Perhaps we have an `Employee` base class that holds the name of the employee. Perhaps we also have two derivatives named `HourlyEmployee` and `SalariedEmployee`. `HourlyEmployee` holds a list of `TimeCard` objects, whereas `SalariedEmployee` holds just the salary. Now let's say we want to generate a report, one line per employee. For every hourly employee we want that line to read 'Hourly Bob worked 8 hours.', whereas

for every salaried employee we'd like it to read 'Salaried Bill earns \$350.'"

"Clearly we could put an abstract function named `reportLine` into the `Employee` base class and implement it appropriately in the two derivatives. However..." And with this he stared out at us meaningfully, "as you all know this would violate the Single Responsibility Principle!" He continued to stare meaningfully.

I understood him. I looked around and could tell that Jasper, Jerry, Jasmine, Jean, Adelaide, and even Avery did too. But there were quite a few people in that room who looked uncomfortable. Was it possible that they did not know about the SRP? Was the SRP something related to the tight group under the sway of Jean?

David reared back a bit and nodded in a way that made it clear that he had not really expected everyone to know about the SRP. "It violates the SRP because the `Employee` hierarchy should not know about the format of a report. Classes that deal with business rules should not also be coupled to things like report formats!"

"So, then, what do we do? How do we get the polymorphic behavior we want without coupling it to the `Employee` hierarchy? The answer is the Visitor pattern!"

And with that he began to type on the front wall. Or rather he wiggled his fingers as *though* he had told the wall to track his typing. It was clear, however, as the characters rapidly flew in from the left and one-by-one took their place on the wall, that his was all prepared, and just a bit of theatre. The whole typing charade took less than 5 seconds.

```
public class ReportVisitorTest {
    private HourlyEmployee hal;
    private SalariedEmployee sam;
    private EmployeeReportVisitor v;

    @Before
    public void createEmployees() {
        hal = new HourlyEmployee("Hal");
        hal.addTimeCard(new TimeCard(8));

        sam = new SalariedEmployee("Sam", 500);
        v = new EmployeeReportVisitor();
    }

    @Test
    public void hourlyEmployeeReportsHours() {
        Employee e = hal;
        e.accept(v);
        assertEquals("Hourly Hal worked 8 hours.", v.getReportLine());
    }

    @Test
    public void salariedEmployeeReportsSalary() {
        Employee e = sam;
        e.accept(v);
        assertEquals("Salaried Sam earns $500.", v.getReportLine());
    }
}
```

"Consider this unit test." He said. "It clearly exhibits the polymorphism we want. And it uses a visitor to make sure that the polymorphic behavior is decoupled from the `Employee` hierarchy. Here are the rest of the classes."

With that he waved his hand at the screen, and the rest of the code flashed into place on the screen with a resounding clash of cymbals the opening guitar lead from *Layla*.

I looked around and caught a few eyes that were saying to me, and to each other "What a ham."

```
public abstract class Employee {
```

```

    protected String name;

    public Employee(String name) {
        this.name = name;
    }

    public abstract void accept(EmployeeVisitor v);

    public String getName() {
        return name;
    }
}

```

```

public class SalariedEmployee extends Employee {
    private int salary;

    public SalariedEmployee(String name, int salary) {
        super(name);
        this.salary = salary;
    }

    public void accept(EmployeeVisitor v) {
        v.visit(this);
    }

    public int getSalary() {
        return salary;
    }
}

```

```

public class HourlyEmployee extends Employee {
    private List<TimeCard> timeCards;
    public HourlyEmployee(String name) {
        super(name);
        timeCards = new ArrayList<TimeCard>();
    }

    public void addTimeCard(TimeCard timeCard) {
        timeCards.add(timeCard);
    }

    public void accept(EmployeeVisitor v) {
        v.visit(this);
    }

    public int getHours() {
        int hours = 0;
        for (TimeCard tc : timeCards)
            hours += tc.getHours();
        return hours;
    }
}

```

```

public class TimeCard {
    private int hours;

    public TimeCard(int hours) {
        this.hours = hours;
    }

    public int getHours() {
        return hours;
    }
}

```

```

public interface EmployeeVisitor {

```

```

    void visit(HourlyEmployee hourlyEmployee);
    void visit(SalariedEmployee salariedEmployee);
}

public class EmployeeReportVisitor implements EmployeeVisitor {
    private String reportLine;

    public String getReportLine() {
        return reportLine;
    }

    public void visit(HourlyEmployee hourlyEmployee) {
        int hours = hourlyEmployee.getHours();
        String name = hourlyEmployee.getName();
        reportLine = String.format("Hourly %s worked %d hours.", name, hours);
    }

    public void visit(SalariedEmployee salariedEmployee) {
        String name = salariedEmployee.getName();
        int salary = salariedEmployee.getSalary();
        reportLine = String.format("Salaried %s earns $%d.", name, salary);
    }
}

```

I read through the code for a few seconds and saw the unmistakable structure of the Visitor pattern.

David spun around to face the audience and said: “Notice how the report formatting code is sequestered nicely in the `EmployeeReportVisitor` class. Notice also that the `Employee` hierarchy knows nothing about it. This is nice, and conforms well with the SRP. “ But then he took on that apologetic demeanor of his, he let his shoulder’s drop and his arms go limp. He said: “But, there’s a problem. See how the `EmployeeVisitor` names the two derivatives of `Employee` as arguments of its two `visit` functions? This creates a cycle of dependencies that makes it hard to add new `Employee` derivatives. “

I’d been through this before, so it wasn’t a surprise to me; but others seemed puzzled.

“Look!” he said with faux frustration, “The `Employee` class depends on `EmployeeVisitor` which depends upon both `SalariedEmployee` and `HourlyEmployee`. If you add a new derivative of `Employee`, like `CommissionedEmployee`, you have to add a new `visit` function to `EmployeeVisitor`. Since `Employee` depends on `EmployeeVisitor`, this change will affect `Employee` and anyone who uses `Employee`!”

Other than Jerry, Jasmine, and my other immediate co-workers, I didn’t see any lightbulbs go on. Did these people *really* not understand transitive dependencies?

David stood there for a minute, obviously hoping to get some feedback from the audience that they understood the issue. I saw Jean flash him a meaningful look, and then he dropped his hands and went on.

“There have been a number of solutions to this problem with Visitor.” He said. We can sometimes use a Decorator, or an Extension Object, or even an AcyclicVisitor. But each of those have their downsides. But now look at the same problem in Ruby.”

This time David brandished his arms at the screen as if invoking a magic spell.

```

describe EmployeeReportVisitor do
  before do
    @hal = HourlyEmployee.new("Hal")
    @hal.addTimeCard(TimeCard.new(8))
    @sam = SalariedEmployee.new("Sam", 500)
    @v = EmployeeReportVisitor.new
  end

  it "should generate hourly report for hourly employee" do
    @hal.accept(@v)
    @v.getReportLine.should == "Hourly Hal worked 8 hours."
  end
end

```

```

end

it "should generate salaried report for salaried employee" do
  @sam.accept(@v)
  @v.getReportLine.should == "Salaried Sam earns $500."
end
end

```

“Once again, here are the unit tests – written in *rspec*. You can clearly see the similarity. You can see that we are expecting polymorphic behavior through the agency of a visitor. Now here is the rest of the code!” The wall filled with Ruby code while the last few strains of the *Also Sprach Zarathustra* overture blared out.

```

class Employee
  def initialize(name)
    @name = name;
  end

  def getName
    @name
  end
end

class HourlyEmployee < Employee
  def initialize(name)
    super(name)
    @timeCards = []
  end

  def addTimeCard(timeCard)
    @timeCards << timeCard;
  end

  def accept(visitor)
    visitor.visitHourly(self)
  end

  def getHours
    hours = 0;
    @timeCards.each {|tc| hours += tc.getHours}
    hours
  end
end

```

```

class SalariedEmployee < Employee
  def initialize(name, salary)
    super(name)
    @salary = salary
  end

  def getSalary
    @salary
  end

  def accept(visitor)
    visitor.visitSalaried(self)
  end
end

```

```

class TimeCard
  def initialize(hours)

```

```

    @hours = hours
end

def getHours
    @hours
end
end

class EmployeeReportVisitor
  def visitHourly(hourlyEmployee)
    name = hourlyEmployee.getName
    hours = hourlyEmployee.getHours
    @reportLine = "Hourly #{name} worked #{hours} hours."
  end

  def visitSalaried(salariedEmployee)
    name = salariedEmployee.getName
    salary = salariedEmployee.getSalary
    @reportLine = "Salaried #{name} earns $#{salary}."
  end

  def getReportLine
    @reportLine
  end
end

```

“Again, you should be able to see the similarity. But notice! There is no `EmployeeVisitor` base class! The `accept` methods simply call `visitHourly` or `visitSalaried` on some object of unknown type. In our case it just happens to be an `EmployeeReportVisitor` object. The ruby runtime works this out. So there is no `EmployeeVisitor` to change if we add a new derivative of `Employee`. And even if there were, notice that `Employee` does not depend upon `EmployeeVisitor`. Indeed, there is no `accept` method declared in the `Employee` base!”

Jerry raised his hand. David stopped and called on him.

“This makes my head hurt.” Said Jerry. How can the Ruby compiler generate any code? How can it allow `HourlyEmployee.accept` to call `visitHourly` on some visitor object of unknown type?”

“First, Jerry, there is no Ruby compiler! The language is interpreted. So there is no code to generate. But even if there were, the compiler wouldn’t have any trouble. It would simply ask the visitor object to execute the `visitHourly` method. If that visitor object did not have that method, it would throw an exception.”

“That makes my head hurt even more.” Said Jerry as he sat down again.

David continued. “Can you see that the dependency cycle is broken? This is one of the great benefits of dynamically typed languages. The dependency knots created in programs written in statically typed languages simply don’t exist. They are resolved at runtime instead of compile time!”

Predictably, Jasmine stood up and said: “Yes, but isn’t this inherently unsafe? How can you defer your errors to runtime without risk that your system will crash in production?”

“How could it be any less safe if we use Test Driven Development? If we write our unit tests according to Mr. C’s rules, then we’ll find the type errors that the Java compiler would have found.” He stopped for a second and then continued. “Let me put it this way. You have to admit that if we are using TDD, then the risk of a type error getting through is very small. Yet the benefits of dynamic typing are huge. Code is smaller, simpler, and more flexible. You have fewer dependency knots. It is easier to make changes. So, if you are using TDD, the benefits vastly outweigh the risks.”

Jasmine looked skeptical. Jerry was befuddled. And Jean was glancing at her watch.

“I think it’s time to go, Dears.” She said. And so we all gathered up our things and filed out.

As I walked out of the room I realized that Jasper had remained utterly silent the whole time. I looked back and saw him behind me. He beamed at me with that huge toothy smile of his.

The Craftsman: 52

Clean Code: C1

Inappropriate Information

Robert C. Martin
20 January 2009

December 31, 1944, 23:55:00

Wallace and Truman celebrated the New Year with the White House staff. Though the music was good, and the party was lively, both men kept nervously looking at their watches. If the other party-goers thought the men were steeling themselves for the inevitable speeches they'd soon be making, they were mistaken.

As the clock struck midnight, a new sun blossomed in the Nevada desert. Riding that fervent impulse was a vessel the size of the Lincoln Memorial. A second later, and a 100' in the air, another sun blared forth, followed by another, and then another.

The shock absorbers between the cabin and the pusher plate damped the 1-kiloton blasts. The twenty crewmen aboard felt the G forces rise and fall as though they were swinging high on a playground swing set.

They were on their way to see Clyde.

Thursday, 7 Mar 2002, 14:00

The "Brown-Bag" was boring today. One of the virtual attendees gave a talk about some big framework. I didn't get the name, but I think it rhymed with "sore".

I left before it was over. Jasmine followed me out and faked stifling a yawn. We both smirked. Then she asked me if I'd mind pairing a bit with Adelaide. So after lunch I put the finishing touches on a module, made sure all the tests passed, pushed the module to github, fired off a Hudson build, and then sauntered over to where Adelaide was working.

"Hi Alphonse, thanks for stopping by." she murmured, never taking her eyes off the screen.

"Sure." I said, as I looked at her screen. She had a new module up, and it looked like she was just typing it from scratch. The only thing on the screen was:

```
/**
 * Farvle -- a class that performs 'Farvle' operations
 *          on 'Arvadents' and other slithy toves.
 * @Date: 7 Mar, 2002
 * @Ship: Dyson
 * @Guild: Mr. C
 * @Author: Adelaide
 * Version History:
 * 20020307 adl -- initial version.
 */
public class Farvle {
}
```

“Adelaide, why did you type all of that?”

She looked a question at me, but did not answer.

“Did Jasmine tell you to do that?”

“N-no,” she stammered, “This is how they taught us to start a module in CS.”

“You’ve been working with Jasmine for a few days now, have you ever seen anything like this at the front of the modules that you and she have worked on together?”

“N-no, I had wondered about that.” She mewed. “It seemed to me that those modules weren’t very well documented.”

“Are you familiar with Mr. C’s *Clean Code* rules?”

She hung her head and mumbled: “I haven’t had time to read through them all yet.”

“OK,” I said. “The first rule of comments (C1) is: ‘Avoid Inappropriate Information’”.

She looked at me strangely for a split second, and then stared at her lap. I thought she wasn’t going to respond, but a moment later she gritted her teeth and forcefully said: “I don’t see anything inappropriate about that information.”

I realized I needed to tread gently. After all, I was only a few days more senior than she. So I carefully held out my hands for the keyboard and said: “May I?”

She looked at me from the corner of her eye, and passed the keyboard my way.

“Thank you. Now have you created a git repository for this yet?”

“No,” she said. “Not yet.”

“Then please allow me.” And so I saved the module and typed the commands that would create a git repository and commit her module into it.

```
$ git init
$ git add src/Farvle.java
$ git commit -m "initial commit"
```

I handed the keyboard back to her and said: “OK, now type `git log`”.

She did as I asked, and up on the screen came the following message:

```
$ git log
commit a4c3868b8386ce5f7eeae2a449f8ef8c89081af5
Author: adelade <adelade@mrc.dyson.gld>
Date: Thu Mar 7 14:06:06 2002

    initial commit
```

“All that information that you had put into your source file is in the git log. There’s nothing inappropriate about the information itself; but the source file is the wrong place for it.”

She was staring at her lap again, and I could see that she was building up to a protest. So I quickly brought the *Clean Code* rules up on the screen.

C1: Inappropriate Information

It is inappropriate for a comment to hold information better held in a different kind of system such as your source code control system, your issue tracking system, or any other record-keeping system. Change histories, for example, just clutter up source files with volumes of historical and uninteresting text. In general, meta-data such as authors, last-modified-date, SPR number, and so on should not appear in comments. Comments should be reserved for technical notes about the code and design.

She looked at the screen for several minutes, and then sat back in her seat and looked over at me.

“OK, I get it.” She said. “But then why did my CS teachers tell me to put all that stuff in my source files?”

I sighed, the way I’d seen Jerry sigh when I asked him questions like that. Then I gave her the answer that he gave me: “They teach you a lot of goofy things in school, Adelaide.”