Tests Driving Towards SRP: An Example

Today I ran into a simple example of how easily one can violate the Single Responsibility Principle and how I can use tests to detect the violation. As with all tiny examples, it seems ridiculous on the surface to consider it risky. Even so, I think you can imagine that your legacy code consists of several thousand instances of code that looks as harmless as this.

I'm working on a version of my Point of Sale training example, which implements a kind of cash register that eventually takes on additional features, like managing inventory and generating sales reports. In order to integrate with my barcode scanner, CueCat, I need to write code that parses lines from `stdin`, then tokenizes these lines and fires the event `onBarcode` for each token, which is a single line of text trimmed of whitespace.

At first, I build a little cluster of objects that turns a `Reader` into `onBarcode` events, but then I isolate the *reading* behavior, leaving behind a smaller cluster of objects. This new cluster turns a `Stream` of lines into `onBarcode` events, but then I find that the tests tangle up removing whitespace with firing the events, so I isolate the *lexing* behavior, leaving behind an even smaller cluster of objects. This new new cluster turns a `Stream` of tokens (trimmed text) into `onBarcode` events, and that seems quite reasonable. The tests almost seem *too* simple.

```
package ca.jbrains.pos.test;

import io.vavr.collection.Stream;
import org.junit.Test;
import org.mockito.Mockito;

public class InterpretTextCommandsAsBarcodesTest {
    private final BarcodeScannedListener barcodeScannedListener =
Mockito.mock(BarcodeScannedListener.class);

    @Test
    public void oneBarcode() throws Exception {
        interpretCommands(barcodeScannedListener, Stream.of("::barcode::"));

        Mockito.verify(barcodeScannedListener).onBarcode("::barcode::");
        // REFACTOR I think this becomes "verify at most n commands (of any kind,
let alone 'barcode')".
        Mockito.verify(barcodeScannedListener,
Mockito.atMost(1)).onBarcode(Mockito.anyString());
    }

    @Test
    public void noBarcodes() throws Exception {
        interpretCommands(barcodeScannedListener, Stream.empty());

        Mockito.verify(barcodeScannedListener,
Mockito.never()).onBarcode(Mockito.any());
    }

    @Test
    public void threeBarcodes() throws Exception {
        interpretCommands(barcodeScannedListener, Stream.of(
                "::barcode 1::", "::barcode 2::", "::barcode 3::"));

        Mockito.verify(barcodeScannedListener).onBarcode("::barcode 1::");
        Mockito.verify(barcodeScannedListener).onBarcode("::barcode 2::");
        Mockito.verify(barcodeScannedListener).onBarcode("::barcode 3::");
        Mockito.verify(barcodeScannedListener,
Mockito.atMost(3)).onBarcode(Mockito.anyString());
    }

    // CONTRACT
    // assume that all commands are "valid" for whatever meaning of "valid"
matters to you.
    private void interpretCommands(BarcodeScannedListener barcodeScannedListener,
Stream<String> commands) {
        commands.forEach(barcodeScannedListener::onBarcode);
    }
}
```

Here I do "TDD as if you meant it", using the technique where I test-drive the production code as a *fully-explicit function*. This function isn't *pure*, but it can at least be `static`. I don't care (yet) about how to partially apply the function to turn it into an object. I can figure that out later. I describe this approach in "Injecting Dependencies, Partially Applying Functions, and It Really Doesn't Matter".

The test seems pretty simple: given a stream of commands, fire them all as `onBarcode` events, because the system doesn't have any other events yet. I can already imagine adding new commands for checking out, printing a receipt, generating a sales report, pausing a purchase, and resuming one. (The first time I saw this, it amazed me: while checking the price of an item I wanted to buy, the cashier paused my purchase by printing a ticket with a barcode on it, then she checked out the next two shoppers. When her co-worker gave her the price of my item, she scanned the ticket and resumed my purchase. Neat!) I digress. Although the test *seems* pretty simple, I find something not to like about this test: it tangles two bits of behavior together, namely interpreting an individual command and applying that interpreter to a `Stream` of commands.

How Reuse Happens

I routinely see programmers extract a reusable component 65% of the way. By this, I mean that they strip away most of a potentially-reusable piece of code, but then leave it tangled with one or two other bits of behavior only because this is the one way that they happen to want to use that potentially-reusable behavior right now, here, so far. This leads inexorably towards legacy code: I can only run *this* code in *that* context, and eventually *that* context grows into a horrible gelatinous blob full of network connections, file systems, and configuration settings. You can't run the tests? Weird. It works on my machine.

You can usually avoid this mess by going *one* or *two* steps farther! Pick a relatively small function and look to see whether it combines some purely abstract, generic, reusable behavior with some more concrete, specific, context- or domain-specific behavior. If you practise looking for it, then eventually you'll start finding it, even in ridiculously harmless-looking examples like the one in this article. Once you find it, then you can pull the reusable bit away from the rest, leaving both parts considerably easier to test. You might even decide that you trust the reusable bit so much that you don't need to test it! (Gasp!)

What's Really Happening Here?

In this example, `interpretCommands()` does two things:

1. It interprets a command as a barcode.
2. It `map` s this interpretation over a `Stream` of commands. (Strictly speaking, since interpreting a command returns nothing, we use `forEach` instead of `map`, but for our purposes here, the distinction matters not a whit.)

Wait. The first of these talks about domain concepts (barcode), while the second of these talks only about generic concepts (`map` / `forEach`, `Stream` and command). Surely we can isolate the generic parts and leave behind something simpler to test. Indeed we can! In fact, we leave behind something so simple that we don't even need to test it yet! Every command is a barcode—at least for now.

Even better, we also leave behind something so trustworthy that we understand so well, that we don't need to test it at all! I know how `forEach` works and I trust Vavr to have implemented correctly. If I didn't, then I could write learning tests for it.

Where Do I Compose These Functions?

The next level up the call stack will handle it. (I visualize the call stack top-down, so the entry point sits at the top.)

Where Will You Test That You Interpret All The Commands?

I won't. I don't need to test it. It'll Just Work^{SM}.

Really?

Really. I presume that my customer will play around with the system and check that it works, but I won't feel any anxiety about it. I know that It'll Just Work^{SM}. Of course, if something happened to prove me wrong, then I'd go to the box, two minutes by myself, and I'd feel shame. I would end up writing an integrated test or two for the offending area, then figure out how to redesign that part of the system so that I couldn't get it wrong. In the meantime, I have more than enough confidence about this particular part of the system.

Refactoring The Design

According to the documentation for `forEach()`, it accepts a `Consumer`, which has the contract of taking a single object and returning nothing. The method `onBarcode()` fulfils that contract trivially by having the right "shape": it takes a `String` and returns nothing. The contract for `Consumer` has no semantics, beyond the universal (and annoying) "don't throw an exception".

I can isolate the generic behavior by extracting a parameter for `barcodeScannedListener::onBarcode`. (If you don't speak Java, that's how they refer to the function so that one can pass that function as a parameter. It magically acts as a `Consumer` of `String` objects.) This leaves me with pretty silly-looking tests.

```
package ca.jbrains.pos.test;

import io.vavr.collection.Stream;
import org.junit.Test;
import org.mockito.Mockito;

import java.util.function.Consumer;

public class InterpretTextCommandsAsBarcodesTest {
    private final BarcodeScannedListener barcodeScannedListener =
Mockito.mock(BarcodeScannedListener.class);

    @Test
    public void oneBarcode() throws Exception {
        interpretCommands(Stream.of("::barcode::"),
barcodeScannedListener::onBarcode);

        Mockito.verify(barcodeScannedListener).onBarcode("::barcode::");
        // REFACTOR I think this becomes "verify at most n commands (of any kind,
let alone 'barcode')".
        Mockito.verify(barcodeScannedListener,
Mockito.atMost(1)).onBarcode(Mockito.anyString());
    }

    @Test
    public void noBarcodes() throws Exception {
        interpretCommands(Stream.empty(), barcodeScannedListener::onBarcode);

        Mockito.verify(barcodeScannedListener,
Mockito.never()).onBarcode(Mockito.any());
    }

    @Test
    public void threeBarcodes() throws Exception {
        interpretCommands(Stream.of(
                "::barcode 1::", "::barcode 2::", "::barcode 3::"),
                barcodeScannedListener::onBarcode);

        Mockito.verify(barcodeScannedListener).onBarcode("::barcode 1::");
        Mockito.verify(barcodeScannedListener).onBarcode("::barcode 2::");
        Mockito.verify(barcodeScannedListener).onBarcode("::barcode 3::");
        Mockito.verify(barcodeScannedListener,
Mockito.atMost(3)).onBarcode(Mockito.anyString());
    }

    // CONTRACT
    // assume that all commands are "valid" for whatever meaning of "valid"
matters to you.
    private void interpretCommands(Stream<String> commands, Consumer<String>
interpretCommand) {
        commands.forEach(interpretCommand);
    }
}
```

Here I can't tell who's testing whom: does `interpretCommands()` test `onBarcode()` or does `onBarcode()` test `interpretCommands()`? Either way, it seems like an awful lot of work just to verify that `forEach()` applies a `Consumer` to every item in a `Stream`. If I wanted to check that, then I could check it without any reference to barcodes. *And that would make this test entirely domain-neutral, which would make it very clear that we can reuse this behavior anywhere.*

I don't need to check `forEach()`. I trust it. That leaves `onBarcode()`. Wait, no. That leaves `onBarcode()` *as the production implementation of* `interpretCommand()`. We have a command interpreter coming to life in this design, so I decide to extract it now.

```java
package ca.jbrains.pos.test;

import io.vavr.collection.Stream;
import org.junit.Test;
import org.mockito.Mockito;

public class InterpretTextCommandsWithCommandInterpreterTest {
    private final InterpretCommand interpretCommand =
Mockito.mock(InterpretCommand.class);

    @Test
    public void oneCommand() throws Exception {
        interpretCommandsWith(Stream.of("::command::"), interpretCommand);

        Mockito.verify(interpretCommand).interpretCommand("::command::");
        // REFACTOR I think this becomes "verify at most n commands (of any kind,
let alone 'command')".
        Mockito.verify(interpretCommand,
Mockito.atMost(1)).interpretCommand(Mockito.anyString());
    }

    @Test
    public void noCommands() throws Exception {
        interpretCommandsWith(Stream.empty(), interpretCommand);

        Mockito.verify(interpretCommand,
Mockito.never()).interpretCommand(Mockito.any());
    }

    @Test
    public void threeCommands() throws Exception {
        interpretCommandsWith(Stream.of(
                "::command 1::", "::command 2::", "::command 3::"),
                interpretCommand);

        Mockito.verify(interpretCommand).interpretCommand("::command 1::");
        Mockito.verify(interpretCommand).interpretCommand("::command 2::");
        Mockito.verify(interpretCommand).interpretCommand("::command 3::");
        Mockito.verify(interpretCommand,
Mockito.atMost(3)).interpretCommand(Mockito.anyString());
    }

    // CONTRACT
    // assume that all commands are "valid" for whatever meaning of "valid"
matters to you.
    private void interpretCommandsWith(Stream<String> commands, InterpretCommand
interpretCommand) {
        commands.forEach(interpretCommand::interpretCommand);
    }
}
```

I just don't see the point in these tests, so I delete them.

You can imagine the simple implementation of the new interface `InterpretCommand`.

```
package ca.jbrains.pos.test;

public class InterpretPointOfSaleCommand implements InterpretCommand {
    private BarcodeScannedListener barcodeScannedListener;

    public InterpretPointOfSaleCommand(BarcodeScannedListener
barcodeScannedListener) {
        this.barcodeScannedListener = barcodeScannedListener;
    }

    @Override
    public void interpretCommand(String command) {
        barcodeScannedListener.onBarcode(command);
    }
}
```

Eventually, this will become a dispatcher, a kind of routing table, firing an event for each command to whoever wants to listen. For now, it fires one kind of event to one, mandatory listener. Good enough for now.

One responsibility. Dead simple.

When it becomes a routing table, I can apply this entire microtechnique again, extracting a generic routing table, putting abstractions in code and details in metadata, and leaving behind a collection of routing rules. Each rule is a function that turns a regular expression into firing an event. Or maybe just an event. I don't know yet. I'll figure that out when I get there.

Putting It Together

Since you might be wondering, I put the pieces of this little system together as follows.

```
package ca.jbrains.pos;

import ca.jbrains.java.ReaderBasedTextSource;
import ca.jbrains.pos.test.*;
import ca.jbrains.pos.test.FindPriceInMemoryCatalogTest.InMemoryCatalog;

import java.io.InputStreamReader;
import java.util.HashMap;

public class PointOfSaleTerminal {
    public static void main(String[] args) {
        CommandLexer commandLexer = new RemovingWhitespaceCommandLexer();

        InterpretCommand interpretCommand = new InterpretPointOfSaleCommand(
                new SellOneItemController(
                        new InMemoryCatalog(
                                new HashMap<String, Price>() {{
                                    put("12345", Price.euroCents(495));
                                    put("23456", Price.euroCents(750));
                                }}
                        ),
                        new PrintStreamDisplay(System.out)
                )
        );

        new ReaderBasedTextSource(new InputStreamReader(System.in))
                .parseIntoLines()
                .flatMap(commandLexer::tokenize)
                .forEach(interpretCommand::interpretCommand);
    }
}
```

Once I have the pieces in place, I simply wire them all together and It Just Works^{SM}.

References

J. B. Rainsberger, "Injecting Dependencies, Partially Applying Functions, and It Really Doesn't Matter". Take advantage of the equivalence between constructors and partially-applied functions to simplify your tests, and maybe also to simplify your design.

J. B. Rainsberger, "How Reuse Happens". Reuse happens when we decide to make it happen.

J. B. Rainsberger, "Stop. Write a Learning Test.". An example of when I write learning tests for other people's code.

J. B. Rainsberger, "Demystifying the Dependency Inversion Principle". An overview of various ways you can think of and use this fundamental design principle.