Extracted from:

The Cucumber for Java Book

Behaviour-Driven Development for Testers and Developers

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

The Cucumber For Java Book

Behaviour-Driven Development for Testers and Developers

Seb Rose, Matt Wynne, and Aslak Hellesøy

edited by Jacquelyn Carter

The Cucumber for Java Book

Behaviour-Driven Development
for Testers and Developers

Seb Rose

Matt Wynne

Aslak Hellesøy

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The team that produced this book includes:

Jacquelyn Carter (editor)
Potomac Indexing, LLC (indexer)
Liz Welch (copyeditor)
Dave Thomas (typesetter)
Janet Furlow (producer)
Ellie Callahan (support)

Printed in the United States of America.
ISBN-13: 978-1-941222-29-4
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—February 2015

Handling Failure

We've delivered a working scenario and shown it to our stakeholders. They're happy with what we've done so far, but they want to discuss how we're going to handle some common error cases. Once we've had these discussions, it'll be time to extend our web app, and we'll see some examples of how to write scenarios that force an error to happen so that we can drive out our error handling functionality.

Our running scenario describes the situation where a customer who has money in his account successfully withdraws some of it using the ATM. What happens if the mechanism that counts and dispenses the money malfunctions? Or if the ATM doesn't contain enough money to fulfill the customer's request? We're sure you can think up plenty of other situations that need to be considered, but that's enough to give you an idea.

Now we'll implement the scenarios that you captured while working with your stakeholders. Since Spring is such a popular DI container, we'll start from the code that we wrote in *Spring in Your Steps*, on page ?. We won't spend a lot of time describing the code, because you learned most of this earlier in the book, but we will describe new Selenium functionality as we use it. You can always jump ahead to *Reusing the Browser*, on page 10 if you want to skip the development phase.

A Faulty ATM

The first scenario we decide to handle is where the ATM develops a fault after the user tries to withdraw money but before that money is dispensed. Working with our stakeholders, we capture this scenario as follows:

fast/01/src/test/resources/cash_withdrawal.feature
```
Scenario: Unsuccessful withdrawal due to technical fault
  Given my account has been credited with $100.00
  But the cash slot has developed a fault
  When I withdraw $20
  Then I should see an out-of-order message
  And $0 should be dispensed
  And the balance of my account should be $100.00
```

There are two new steps in this scenario: one that injects a fault into the cash slot and another that checks that the correct error message is displayed to the user. We'll look at both of these before seeing if we can do anything to improve how the scenario reads.

Injecting a Fault

We'd like to make sure that our software behaves correctly when something goes wrong with the ATM mechanism, represented by our CashSlot. We wouldn't want to change our production code, so instead we create a TestCashSlot (that extends CashSlot) to allow us to simulate a fault:

fast/01/src/test/java/support/TestCashSlot.java

```java
package support;

import nicebank.CashSlot;

public class TestCashSlot extends CashSlot {
    private boolean faulty;

    public void injectFault() {
        faulty = true;
    }

    public void dispense(int dollars){
        if (faulty) {
            throw new RuntimeException("Out of order");
        } else {
            super.dispense(dollars);
        }
    }
}
```

Since we're using Spring, we can make a simple change to the configuration file cucumber.xml to inject our test class into our application:

fast/01/src/test/resources/cucumber.xml

```xml
<bean class="support.AtmUserInterface" scope="cucumber-glue" />
<bean class="support.TestCashSlot" scope="cucumber-glue" />
```

Now when our scenarios run, Spring will create a single instance of TestCashSlot and inject it anywhere that we need a TestCashSlot or CashSlot. When we try to withdraw cash from a faulty ATM, dispense will throw an exception. For the time being we're using Java's RuntimeException, but we'd want to replace this with something more meaningful if this was more than an example.

We can inject a fault into our TestCashSlot at any time using the injectFault method. We call this from our new step "But the cash slot has developed a fault":

fast/01/src/test/java/nicebank/CashSlotSteps.java

```java
@Given("^the cash slot has developed a fault$")
public void theCashSlotHasDevelopedAFault() throws Throwable {
    cashSlot.injectFault();
}
```

Remember that by the time this step definition gets executed, the TestCashSlot will already have been created by Spring and wired into our application. We're simply changing a flag to indicate that for all future calls we want it to behave as if it was faulty.

Checking for Text

If the ATM develops a fault, we want a helpful message to be displayed to the user. We'll use Selenium WebDriver to check that the correct message is being displayed. To do this we add the following step definition:

fast/01/src/test/java/nicebank/TellerSteps.java
```
@Then("^I should see an out-of-order message$")
public void iShouldSeeAnOutOfOrderMessage() throws Throwable {
    Assert.assertTrue(
        "Expected error message not displayed",
        teller.isDisplaying("Out of order"));
}
```

All this step definition is doing is delegating responsibility to the Teller implementation to check that the required "Out of order" message is being displayed. We don't do any complicated work in the step definition itself, because as we've already explained we want to keep this layer of glue code as thin as possible. We do the actual work of checking the text displayed on the UI in AtmUserInterface:

fast/01/src/test/java/support/AtmUserInterface.java
```
public boolean isDisplaying(String message) {
    List<WebElement> list = webDriver
        .findElements(By.xpath("//*[contains(text(),'" + message + "')]"));
    return (list.size() > 0);
}
```

Of course, the first time we run this it'll fail, because we haven't made any change to our production code yet. To handle the exception thrown when we try to use a faulty CashSlot, we've modified the doPost on WithdrawalServlet. It now extracts the message from the caught exception and displays it to the user:

fast/01/src/main/java/nicebank/WithdrawalServlet.java
```
    protected void doPost(HttpServletRequest request, HttpServletResponse response)
                                            throws ServletException, IOException
    {
        Teller teller = new AutomatedTeller(cashSlot);
        int amount = Integer.parseInt(request.getParameter("amount"));

        try {
            teller.withdrawFrom(account, amount);

            response.setContentType("text/html");
            response.setStatus(HttpServletResponse.SC_OK);
            response.getWriter().println(
                "<html><head><title>ATM</title></head>" +
                "<body>Please take your $" + amount + "</body></html>");
        }
        catch (RuntimeException e) {
            response.setContentType("text/html");
            response.setStatus(HttpServletResponse.SC_OK);
            response.getWriter().println(
                "<html><head><title>ATM</title></head>" +
                "<body>" + e.getMessage() + "</body></html>");
        }
}
```

At this point we can run mvn clean test and our new scenario will pass.

Rewriting the Scenario

Take a look at our new scenario. Does it read well to you? Are there any details in it that are *incidental* to the behavior being described? Remember, this scenario is about handling a technical fault. We aren't interested in how much money is in the account to start with, how much money the customer is trying to withdraw, or what the balance is after the fault occurs. What we want to be sure of is that the correct error message is displayed, that no money is dispensed, and that the customer's balance is not affected.

After we discuss this with our stakeholders, we rewrite the scenario as:

fast/02/src/test/resources/cash_withdrawal.feature

```
Scenario: Unsuccessful withdrawal due to technical fault
  Given my account is in credit
  But the cash slot has developed a fault
  When I request some of my money
  Then I should see an out-of-order message
  And $0 should be dispensed
  And the balance of my account should be unchanged
```

This scenario has been stripped of several incidental details, allowing us to use natural language to focus our attention on the things that are really important.

Insufficient Funds in ATM

The next scenario we'll tackle is where the ATM contains less cash than the user tries to withdraw. Working with our stakeholders, we capture this scenario as follows:

fast/03/src/test/resources/cash_withdrawal.feature

```
Scenario: Unsuccessful withdrawal due to insufficient ATM funds
  Given my account is in credit
  And the ATM contains $10
  When I withdraw $20
  Then I should see an ask-for-less-money message
  And $0 should be dispensed
  And the balance of my account should be unchanged
```

This scenario introduces the idea that an ATM contains a limited amount of money. We'll need to implement a way to specify how much money to load into the ATM, and check that we have sufficient funds before we attempt to dispense money to the customer:

fast/03/src/main/java/nicebank/CashSlot.java
```
public void load(int dollars){
    available = dollars;
}

public void dispense(int requested){
    if (available >= requested) {
        contents = requested;
        available -= requested;
    } else {
        throw new RuntimeException("Insufficient ATM funds");
    }
}
```

We also need to make sure that our original, successful withdrawal scenario keeps working. We could do this by adding an extra step to it to load the ATM, but this would be an incidental detail. Instead we add a constructor to our TestCashSlot that makes sure there's plenty of cash available for any scenarios that aren't specifically interested in how much money is in the machine:

fast/03/src/test/java/support/TestCashSlot.java
```
public TestCashSlot() {
    super.load(1000);
}
```

We can now run mvn clean test and see all three scenarios pass.

We added only two extra scenarios, but already our feature takes longer to run. If we don't do something, it won't be long before the team stops running the scenarios. Next we'll take a look at the simplest way to speed things up.

Reusing the Browser

Each of our scenarios is exercising our application through our web UI and so needs to use a browser. At the moment we start a new instance of Firefox for each scenario, which takes a fair amount of time. Is it really necessary, or could our scenarios all use the same instance of Firefox?

It's important that each scenario is *isolated* from all other scenarios, but the browser itself holds very little context. In most situations it is quite safe to reuse the same browser instance for all your scenarios as long as you clear out any cookies. In this example it's even simpler—we have no cookies.

Sharing a Browser Using Spring

We're using a Spring configuration file, cucumber.xml, to define our EventFiringWebDriver. It just takes a tiny modification to keep the browser instance alive for the whole run of Cucumber. In the XML that follows, we've simply removed the attribute scope="cucumber-glue" from the definition of the bean:

fast/04/src/test/resources/cucumber.xml

```
<bean class="org.openqa.selenium.support.events.EventFiringWebDriver"
                                                   destroy-method="quit">
    <constructor-arg>
        <bean class="org.openqa.selenium.firefox.FirefoxDriver" />
    </constructor-arg>
</bean>
```

Now when Spring creates an instance of the web driver, it's associated with the default scope that lives for the entire Cucumber run. Try running mvn clean test and you'll see that this makes the feature run much quicker overall, because we don't have to wait for the browser to start up for each scenario.

If you needed to delete cookies as well, you'd need to write a hook that does this programmatically, which we'll see next.

Using a Shutdown Hook

There's a more general way to do some cleanup when a JVM is closing down that doesn't rely on Spring: using Java's addShutdownHook. The following code comes from the Webbit-Websockets-Selenium example in the Cucumber project:[3]

```
public class SharedDriver extends EventFiringWebDriver {
    private static final WebDriver REAL_DRIVER = new FirefoxDriver();
    private static final Thread CLOSE_THREAD = new Thread() {
        @Override
        public void run() {
            REAL_DRIVER.close();
        }
    };

    static {
        Runtime.getRuntime().addShutdownHook(CLOSE_THREAD);
    }

    public SharedDriver() {
        super(REAL_DRIVER);
    }
```

3. https://github.com/cucumber/cucumber-jvm/blob/master/examples/java-webbit-websockets-selenium/src/test/java/cucumber/examples/java/websockets/SharedDriver.java

```
 -
 -      @Override
 -      public void close() {
20          if (Thread.currentThread() != CLOSE_THREAD) {
 -              throw new UnsupportedOperationException(
 -                "You shouldn't close this WebDriver. " +
 -                " It's shared and will close when the JVM exits.");
 -          }
25          super.close();
 -      }
 -
 -      @Before
 -      public void deleteAllCookies() {
30          manage().deleteAllCookies();
 -      }
 -  }
```

We create a single driver instance and store it in a static variable (line 2). This will be shared by all instances of SharedDriver. We also create a static Thread (line 3) and add it to the JVM's list of hooks that should be called when the JVM is shutting down (line 11).

If you try to close the driver manually, SharedDriver will check whether the call is coming from the registered CLOSE_THREAD (line 20). If it isn't, SharedDriver will report the error through an exception. This protects you from inadvertently closing the browser midway through executing your Cucumber features.

This example also shows how you can use a hook to clear cookies before each scenario runs (line 29). The method manage is part of the EventFiringWebDriver API provided by Selenium.

While driving out some fault handling features in our ATM, we've learned a general way to call cleanup code at the end of a Cucumber run. We've also used Selenium to ensure that each scenario cleans up any cookies that the previous scenario might have saved. Now we need to show what we've done to our users and see whether they like it!