Extracted from:

The Cucumber for Java Book

Behaviour-Driven Development for Testers and Developers

This PDF file contains pages extracted from *The Cucumber for Java Book*, published by the Pragmatic Bookshelf. For more information or to purchase a paperback or PDF copy, please visit http://www.pragprog.com.

Note: This extract contains some colored text (particularly in code listing). This is available only in online versions of the books. The printed versions are black and white. Pagination might vary between the online and printed versions; the content is otherwise identical.

Copyright © 2015 The Pragmatic Programmers, LLC.

All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

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

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and The Pragmatic Programmers, LLC was aware of a trademark claim, the designations have been printed in initial capital letters or in all capitals. The Pragmatic Starter Kit, The Pragmatic Programmer, Pragmatic Programming, Pragmatic Bookshelf, PragProg and the linking *g* device are trademarks of The Pragmatic Programmers, LLC.

Every precaution was taken in the preparation of this book. However, the publisher assumes no responsibility for errors or omissions, or for damages that may result from the use of information (including program listings) contained herein.

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at *https://pragprog.com*.

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)

For international rights, please contact *rights@pragprog.com*.

Copyright © 2015 The Pragmatic Programmers, LLC.
All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

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

Sketching Out the Domain Model

The heart of any object-oriented program is the domain model. When we start to build a new system, we like to work directly with the domain model. This allows us to iterate and learn quickly about the problem we're working on without getting distracted by user interface gizmos. Once we have a domain model that really reflects our understanding of the system, it's easy to wrap it in a pretty skin.

We're going to let Cucumber drive our work, building the domain model classes directly in the step definitions. As usual, we start by running `mvn clean test` on our scenario to remind us what to do next:

```
-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running RunCukesTest
Feature: Cash Withdrawal

  Scenario: Successful withdrawal from an account in credit
    Given I have deposited $100 in my account
    When I request $20
      cucumber.api.PendingException: TODO: implement me
        at nicebank.Steps.iRequest$(Steps.java:22)
        at *.When I request $20(cash_withdrawal.feature:4)
    Then $20 should be dispensed

1 Scenarios (1 pending)
3 Steps (1 skipped, 1 pending, 1 passed)
0m0.090s

cucumber.api.PendingException: TODO: implement me
        at nicebank.Steps.iRequest$(Steps.java:22)
        at *.When I request $20(cash_withdrawal.feature:4)
```

When we last worked on this scenario, we'd just reached the point where we had written the regular expressions for each of our step definitions and implemented the first one. Here's how our steps file looks:

step_definitions_inside/01/src/test/java/nicebank/Steps.java

```
package nicebank;

import cucumber.api.java.en.*;
import cucumber.api.PendingException;

public class Steps {

  class Account {
    public Account(int openingBalance) {
```

```
    }
  }

  @Given("^I have deposited \\$(\\d+) in my account$")
  public void iHaveDeposited$InMyAccount(int amount) throws Throwable {
      new Account(amount);
  }

  @When("^I request \\$(\\d+)$")
  public void iRequest$(int arg1) throws Throwable {
      // Write code here that turns the phrase above into concrete actions
      throw new PendingException();
  }

  @Then("^\\$(\\d+) should be dispensed$")
  public void $ShouldBeDispensed(int arg1) throws Throwable {
      // Write code here that turns the phrase above into concrete actions
      throw new PendingException();
  }
}
```

In that first step definition, we create an instance of our new `Account` class. Cucumber then tells us that we need to work on our second step definition, which is still marked as Pending. Before we do that, let's review the code in our step definition and see what we think. There are a few things we're not happy about:

- Some inconsistent language is creeping in; the step talks about *depositing* funds, but the code passes funds to the `Account` constructor.
- The step is lying to us! It says `Given I have deposited $100 in my account`, and it's passed. Yet we know from our implementation that nothing has been deposited anywhere.
- Bank balances don't always contain whole numbers of dollars, but our step definition uses an `int`. We should be able to deposit dollars and cents.

We'll work through each of these points before we move on to the next step in the scenario.

Getting the Words Right

We want to clarify the wording before we do anything else, so let's think about how we could make the code in the step definition read more like the text in the step. We could go back and reword the step to say something like `Given an Account with a balance of $100`. In reality, though, the only way that an account would have a balance is if someone deposited funds into it. So, let's change the way we talk to the domain model inside our step definition to reflect that:

• Click HERE to purchase this book now. discuss

step_definitions_inside/02/src/test/java/nicebank/Steps.java
```
class Account {
  public void deposit(int amount) {

  }
}
```

step_definitions_inside/02/src/test/java/nicebank/Steps.java
```
@Given("^I have deposited \\$(\\d+) in my account$")
public void iHaveDeposited$InMyAccount(int amount) throws Throwable {
    Account myAccount = new Account();
    myAccount.deposit(amount);
}
```

That seems better.

There's something else in the wording that bothers us. In the step, we talk about *my account*, which implies the existence of a protagonist in the scenario who has a relationship to the account, perhaps a Customer. This is a sign that we're probably missing a domain concept. However, until we get to a scenario where we have to deal with more than one customer, we'd prefer to keep things simple and focus on designing the fewest classes we need to get this scenario running. So, we'll park this concern for now.

Telling the Truth

Now that we're happier with the interface to our Account class, we can resolve the next issue from our code review. After we've deposited the funds in the account, we can check its balance with an assertion:

step_definitions_inside/03/src/test/java/nicebank/Steps.java
```
@Given("^I have deposited \\$(\\d+) in my account$")
public void iHaveDeposited$InMyAccount(int amount) throws Throwable {
    Account myAccount = new Account();
    myAccount.deposit(amount);

    Assert.assertEquals("Incorrect account balance -",
                        amount, myAccount.getBalance());
}
```

We've used a JUnit assertion here, but if you prefer another assertion library, feel free to use that. It might seem odd to put an assertion in a Given step, but it communicates to future readers of this code what state we expect the system to be in once the step has run. We'll need to add a balance method to the Account so that we can run this code:

step_definitions_inside/03/src/test/java/nicebank/Steps.java
```
class Account {
    public void deposit(int amount) {
```

```
    }

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

Notice that we're just sketching out the interface to the class, rather than adding any implementation to it. This way of working is fundamental to outside-in development. We try not to think about *how* the Account is going to work yet but concentrate on *what* it should be able to do.

Now when we run the test, we get a nice helpful failure message:

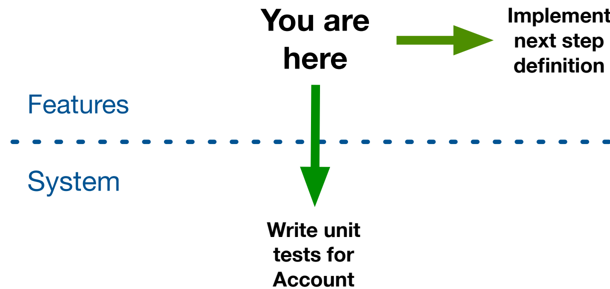
```
  Scenario: Successful withdrawal from an account in credit
    Given I have deposited $100 in my account
      java.lang.AssertionError: Incorrect account balance
                                              - expected:<100> but was:<0>
        at org.junit.Assert.fail(Assert.java:88)
        at org.junit.Assert.failNotEquals(Assert.java:743)
        at org.junit.Assert.assertEquals(Assert.java:118)
        at org.junit.Assert.assertEquals(Assert.java:555)
        at nicebank.Steps.iHaveDeposited$InMyAccount(Steps.java:29)
        at *.Given I have deposited $100 in my account(cash_withdrawal.feature:3)
    When I request $20
    Then $20 should be dispensed

1 Scenarios (1 failed)
3 Steps (1 failed, 2 skipped)
0m0.076s

java.lang.AssertionError: Incorrect account balance
                                        - expected:<100> but was:<0>
```

Now our step definition is much more robust, because we know it will sound an alarm bell if it isn't able to deposit the funds into the account as we've asked it to do. Adding assertions to Given and When steps like this means that if there's ever a regression later in the project, it's much easier to diagnose because the scenario will fail right where the problem occurs. This technique is most useful when you're sketching things out; eventually, we'll probably move this check further down the testing stack into a *unit test* for the Account class and take it out of the step definition.

Doing the Simplest Thing

We're at a decision point here. We've effectively finished implementing our first step definition, but we can't move on to the next one until we've made

some changes to the implementation of the Account class so that the step passes.



It's tempting to pause here, move the Account class into a separate file, and start driving out the behavior we want using unit tests. We're going to try to resist that temptation for now and stay on the outside of the Account class. If we can get a full tour through the scenario from this perspective, we'll be more confident in the design of the class's interface once we do step inside and start implementing it.

So, we'll keep working on our very simple implementation of the Account class that's obviously incomplete but just right enough to make this first step pass. Think of this like putting up scaffolding on a construction site: we're going to take it down eventually, but it will help things to stand up in the meantime.

Change Account to look like this, and now the first step should pass:

step_definitions_inside/04/src/test/java/nicebank/Steps.java

```
class Account {
    private int balance;

    public void deposit(int amount) {
        balance += amount;
    }

    public int getBalance() {
      return balance;
    }
}
```

Good. We still have one issue left on our list, which is our use of int as our balance. Now that our step is passing, we can do that refactoring with confidence.

Staying Honest with Transforms

Another issue we have with the first step definition is that our regular expression is capturing an integer, but we would expect to be able to deposit dollars and cents into the account. So let's change the feature to demonstrate this:

step_definitions_inside/05/src/test/resources/cash_withdrawal.feature

```
Feature: Cash Withdrawal
  Scenario: Successful withdrawal from an account in credit
    Given I have deposited $100.00 in my account
    When I request $20
    Then $20 should be dispensed
```

Now when we run mvn clean test it reports that we have an undefined step definition and tells us what regular expression we now need to use to match our feature:

```
Feature: Cash Withdrawal

  Scenario: Successful withdrawal from an account in credit
    Given I have deposited $100.00 in my account
    When I request $20
    Then $20 should be dispensed

1 Scenarios (1 undefined)
3 Steps (2 skipped, 1 undefined)
0m0.000s

You can implement missing steps with the snippets below:

@Given("^I have deposited \\$(\\d+)\\.(\\d+) in my account$")
public void iHaveDeposited$InMyAccount(int arg1, int arg2) throws Throwable {
    // Write code here that turns the phrase above into concrete actions
    throw new PendingException();
}

Tests run: 5, Failures: 0, Errors: 0, Skipped: 4, Time elapsed: 0.458 sec
```

Cucumber has recognized two numbers in the step and has generated a regular expression that is capturing each separately and passing them as two integers to our step definition. Rather than pass two integers around, we're going to use a Money class written specially for this example, which you can find in src/main/java/nicebank.

Seb says:

Do We Really Have to Reinvent Money?

You might think that a language like Java would have its own money class, but as of this writing it doesn't. There are a number of classes available, such as Joda Money, but we're still waiting for JSR 354 (which will define a Java Money class) to be released.

In our step definition we can now create an instance of the Money class:

step_definitions_inside/06/src/test/java/nicebank/Steps.java

```
@Given("^I have deposited \\$(\\d+)\\.(\\d+) in my account$")
public void iHaveDeposited$InMyAccount(int dollars, int cents) throws Throwable {
    Account myAccount = new Account();
    Money amount = new Money(dollars, cents);
    myAccount.deposit(amount);

    Assert.assertEquals("Incorrect account balance -",
                        amount, myAccount.getBalance());
}
```

And we change our implementation of Account to handle deposits of Money:

step_definitions_inside/06/src/test/java/nicebank/Steps.java

```
class Account {
    private Money balance = new Money();

    public void deposit(Money amount) {
        balance = balance.add(amount);
    }

    public Money getBalance() {
      return balance;
    }
}
```

This is fine, but it still means that we have to create an instance of Money in every step definition that works with dollars and cents. It would be much nicer if Cucumber could just pass a Money object directly to the step definition.

The first thing we need to do to make this happen is to change the step definition so that:

- the regular expression captures the whole amount in a single capture group
- its signature expects a Money parameter

Auto-Conversion Magic

Have you wondered how Cucumber knows what arguments to use in the step definition snippets it generates? First, it generates one argument per capture group in the regular expression. Then, for each capture group, if it matches only numbers it creates an int parameter; otherwise it creates a String parameter. For example:

```
@Given("^a (\\w+) amount \\$(\\d+)$")
public void aDollarAmount$(String arg1, int arg2) throws Throwable {
}
```

What if you wanted to manipulate digits as a String? No problem—these snippets are just a hint from Cucumber to you. If you'd like to work with a different type, then just change the signature of the step definition, like so:

```
@Given("^a (\\w+) amount \\$(\\d+)$")
public void aDollarAmount$(String arg1, String arg2) throws Throwable {
}
```

Under the hood, Cucumber represents each capture group in the regular expression as a String. Then, when calling the step definition it converts the String into the type expected. If it can't perform the conversion, it throws a cucumber.runtime.CucumberException, but otherwise the conversion happens automatically—as if by magic.

step_definitions_inside/07/src/test/java/nicebank/Steps.java

```
@Given("^I have deposited \\$(\\d+\\.\\d+) in my account$")
public void iHaveDeposited$InMyAccount(Money amount) throws Throwable {
    Account myAccount = new Account();
    myAccount.deposit(amount);

    Assert.assertEquals("Incorrect account balance -",
                              amount, myAccount.getBalance());
}
```

Now all we need to do is tell Cucumber how to convert a String object into a Money object. One approach would be to give our Money class a single argument constructor that takes a String. Cucumber would then automatically invoke this constructor when calling the step definition, passing in the original String that matched the regular expression in our capture group.

step_definitions_inside/07/src/main/java/nicebank/Money.java

```
public Money(String amount) {
    Pattern pattern = Pattern.compile("^[^\\d]*([\\d]+)\\.([\\d][\\d])$");
    Matcher matcher = pattern.matcher(amount);

    matcher.find();
    this.dollars = Integer.parseInt(matcher.group(1));
    this.cents = Integer.parseInt(matcher.group(2));
}
```

But what if the Money didn't have a String constructor and wasn't ours to modify? In that case, we're going to need to learn about another Cucumber feature, the Transformer class, which allows us to create the instances of Money that we want without giving it a new constructor.

Transformers work on captured arguments. Each transform is responsible for converting a captured String into something more meaningful. For example, we can use a Transformer to take a String argument that contains a monetary amount and turn it into an instance of our Money class. Let's create a MoneyConverter transformer and put it in a new folder, test/transforms:

step_definitions_inside/08/src/test/java/transforms/MoneyConverter.java

```
package transforms;

import cucumber.api.Transformer;

import nicebank.Money;

public class MoneyConverter extends Transformer<Money> {
     public Money transform(String amount) {
         String[] numbers = amount.split("\\.");

         int dollars = Integer.parseInt(numbers[0]);
         int cents = Integer.parseInt(numbers[1]);

         return new Money(dollars, cents);
     }
 }
```

Then we annotate the parameter in the step definition to tell Cucumber which Transformer to use:

step_definitions_inside/08/src/test/java/nicebank/Steps.java

```
@Given("^I have deposited \\$(\\d+\\.\\d+) in my account$")
public void iHaveDeposited$InMyAccount(
                              @Transform(MoneyConverter.class) Money amount)
                                                                 throws Throwable {
    Account myAccount = new Account();
    myAccount.deposit(amount);

    Assert.assertEquals("Incorrect account balance -",
                            amount, myAccount.getBalance());
}
```

Great! That code looks much cleaner and easier to read.

We can tidy this up a little further by moving the dollar sign into the capture group. This makes the code more cohesive, because we're bringing together

the whole regular expression statement for capturing the amount of funds deposited. It also gives us the option to capture other currencies in the future.

step_definitions_inside/09/src/test/java/nicebank/Steps.java

```
@Given("^I have deposited (\\$\\d+\\.\\d+) in my account$")
public void iHaveDeposited$InMyAccount(
                              @Transform(MoneyConverter.class) Money amount)
                                                              throws Throwable {
    Account myAccount = new Account();
    myAccount.deposit(amount);

    Assert.assertEquals("Incorrect account balance -",
                            amount, myAccount.getBalance());
}
```

Of course we have to make a corresponding change to MoneyConverter to ensure it handles the currency sign correctly. For the time being (since we're not handling multiple currencies) we'll just discard the dollar sign:

step_definitions_inside/09/src/test/java/transforms/MoneyConverter.java

```
String[] numbers = amount.substring(1).split("\\.");
```

Let's take another look at our to-do list. Using the transform has cleared up the final point from the initial code review. As we went along, we collected a new to-do list item: that we need to implement the Account properly, with unit tests. Let's leave that one on the list for now and move on to the next step of the scenario.