Extracted from:

The Cucumber Book

Behaviour-Driven Development for Testers and Developers

This PDF file contains pages extracted from *The Cucumber 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 printer versions; the content is otherwise identical.

Copyright © 2010 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

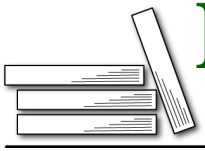
The Pragmatic Programmers

The Cucumber Book

Behaviour-Driven Development for Testers and Developers

Matt Wynne
and Aslak Hellesøy

edited by Jacquelyn Carter



Pragmatic Bookshelf

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 *http://pragprog.com*.

The team that produced this book includes:

Jackie Carter (editor)
Potomac Indexing, LLC (indexer)
Kim Wimpsett (copyeditor)
David J Kelly (typesetter)
Janet Furlow (producer)
Juliet Benda (rights)
Ellie Callahan (support)

Copyright © 2012 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: 978-1-934356-80-7
Printed on acid-free paper.
Book version: P2.0—August 2012

It's time to pull together everything that you've learned in the first part of the book and use it in practice. There are a few advanced concepts left about Cucumber that we want to explain to you, and they'll be much easier to demonstrate with an example. A lot of what we'll do in this part of the book will blur the line between testing and development. If you're more of a tester than a developer, don't let that worry you: the Ruby code we'll build is just about as simple as it gets. By following along, you'll get a good sense of how we like to work, as well as pick up some new knowledge about working with Cucumber.

At the end of Section 4.5, *Returning Results*, on page ?, we'd just started work on a greenfield project to build the software for an ATM. We had a single scenario for the most important behavior of the system: letting someone walk up to the machine and withdraw cash.

Download step_definitions_inside/01/features/cash_withdrawal.feature

```
Feature: Cash Withdrawal
  Scenario: Successful withdrawal from an account in credit
    Given I have deposited $100 in my account
    When I request $20
    Then $20 should be dispensed
```

Now we're going to pick up this scenario and work outside-in, designing the system as we go, just as we would on a real project. In this chapter, we'll get the scenario to pass by driving out a simple domain model for our ATM. Then, in the next chapter, we'll get a nasty surprise when we discover that there's a missing step in our scenario. Finally, we'll demonstrate the benefits of well-engineered test code by introducing a user interface around the domain model.

By the end of this chapter, you'll have learned about Cucumber's `World` and how you can use it to contain state that's shared between step definitions. We'll write some custom helper methods that will introduce a layer of decoupling between our step definitions and the application we're building. We'll show you how to use *transforms* to reduce duplication in your step definitions and make their regular expressions more readable. Finally, we'll show you how we like to organize the files in our projects so that they're easy to work with and maintain.

7.1 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 cucumber on our scenario to remind us what to do next:

```
Feature: Cash Withdrawal

  Scenario: Successful withdrawal from an account in credit
    Given I have deposited $100 in my account
      uninitialized constant Account (NameError)
      ./features/step_definitions/steps.rb:2
      features/cash_withdrawal.feature:3
    When I request $20
    Then $20 should be dispensed

Failing Scenarios:
cucumber features/cash_withdrawal.feature:2

1 scenario (1 failed)
3 steps (1 failed, 2 skipped)
0m0.005s
```

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 started to implement the first one. Here's how our steps file looks:

Download step_definitions_inside/01/features/step_definitions/steps.rb

```
Given /^I have deposited \$(\d+) in my account$/ do |amount|
  Account.new(amount.to_i)
end

When /^I request \$(\d+)$/ do |arg1|
  pending # express the regexp above with the code you wish you had
end

Then /^\$(\d+) should be dispensed$/ do |arg1|
  pending # express the regexp above with the code you wish you had
end
```

In that first step definition, we've sketched out a call to an imaginary class called Account. Ruby has given us an error that tells us that the next thing we need to do is define the Account class. Let's go ahead and do that:

Download step_definitions_inside/02/features/step_definitions/steps.rb

```
class Account
  def initialize(amount)
  end
end
```

```
Given /^I have deposited \$(\d+) in my account$/ do |amount|
  Account.new(amount.to_i)
end
```

Notice that we're defining the class right here in our steps file. Don't worry—it's not going to stay here forever, but it's most convenient for us to create it right here where we're working. Once we have a clear idea of how we're going to work with the class, then we can refactor and move it to a more permanent home. We're also converting the amount captured from the Gherkin step as a string into a number before we pass it into the domain model.

Let's run `cucumber` again and see what it thinks of that:

```
Feature: Cash Withdrawal

  Scenario: Successful withdrawal from an account in credit
    Given I have deposited $100 in my account
    When I request $20
      TODO (Cucumber::Pending)
      ./features/step_definitions/steps.rb:13
      features/cash_withdrawal.feature:4
    Then $20 should be dispensed

1 scenario (1 pending)
3 steps (1 skipped, 1 pending, 1 passed)
0m0.002s
```

Well, that was easy! Perhaps…too easy? Let's review the code in our step definition and see what we think. There are a couple of things we're not happy about:

- There's some inconsistent language creeping in; the step talks about *depositing* funds into the account, but the code passes the funds to the `Account` class's 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.
- Having to convert the amount to an integer is messy. If we have a variable called `amount`, we should expect it to already be a number of some kind, not the string captured by the regular expression.

We'll work through each of these points before we move onto 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:

Download step_definitions_inside/03/features/step_definitions/steps.rb

```
class Account
  def deposit(amount)
  end
end

Given /^I have deposited \$(\d+) in my account$/ do |amount|
  my_account = Account.new
  my_account.deposit(amount.to_i)
end
```

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:

Download step_definitions_inside/04/features/step_definitions/steps.rb

```
Given /^I have deposited \$(\d+) in my account$/ do |amount|
  my_account = Account.new
  my_account.deposit(amount.to_i)
  my_account.balance.should eq(amount.to_i),
    "Expected the balance to be #{amount} but it was #{my_account.balance}"
end
```

We've used an RSpec 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:

Download step_definitions_inside/04/features/step_definitions/steps.rb

```
class Account
  def deposit(amount)
  end

  def balance
  end
end
```

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.

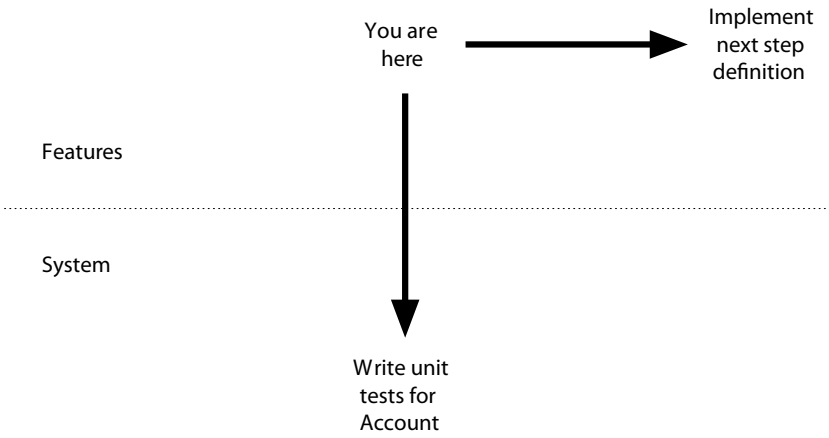
Now when we run the test, we get a nice helpful failure message:

```
Feature: Cash Withdrawal

  Scenario: Successful withdrawal from an account in credit
    Given I have deposited $100 in my account
      Expected the balance to be 100 but it was
      (RSpec::Expectations::ExpectationNotMetError)
      ./features/step_definitions/steps.rb:15
      features/cash_withdrawal.feature:3
    When I request $20
    Then $20 should be dispensed

Failing Scenarios:
cucumber features/cash_withdrawal.feature:2

1 scenario (1 failed)
3 steps (1 failed, 2 skipped)
0m0.003s
```

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 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 make a very simple implementation in the Account class that's obviously incomplete but is 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 the first step should pass:

Download step_definitions_inside/05/features/step_definitions/steps.rb

```
class Account
  def deposit(amount)
    @balance = amount
  end

  def balance
    @balance
  end
end
```

Good. We still have one issue left on our list, which is the duplication of the calls to to_i. Now that our step is passing, we can do that refactoring with confidence.

7.2 Removing Duplication with Transforms

Another issue we have with this step definition is that we have to convert the string captured by the regular expression into an integer. In fact, now that we've added an assertion, we've had to do it twice. As our test suite grows, we can imagine these calls to to_i littering our step definitions. Even these four characters count as duplication, so let's stamp them out.

To do this, we're going to learn about a new Cucumber method, called Transform.

Transforms work on captured arguments. Each transform is responsible for converting a certain captured string and turning it into something more meaningful. For example, we can use this transform to take a matched argument that contains a number and turn it into a Ruby Fixnum integer:

Download step_definitions_inside/06/features/step_definitions/steps.rb

```
Transform /^\d+$/ do |number|
  number.to_i
end
```

We define the transform by giving Cucumber a regular expression that describes the argument we're interested in transforming. Notice that we've used the ^ and $ to anchor the transform's regular expression to the ends of the captured string. This is really important, because we want our transform to match only captures that are numbers, not just captures that contain a number somewhere in them.

When Cucumber matches a step definition, it checks for any transforms that match each argument. When an argument matches a transform, Cucumber passes the captured string to the transform's block, and the result of running the block is what's then yielded to the step definition. This is shown in Figure 6, *Transforms: how do they work?*, on page 12.

With the transform in place, we can now remove the duplicated calls to to_i in our step definition:

Download step_definitions_inside/06/features/step_definitions/steps.rb

```
Given /^I have deposited \$(\d+) in my account$/ do |amount|
  my_account = Account.new
  my_account.deposit(amount)
  my_account.balance.should eq(amount),
    "Expected the balance to be #{amount} but it was #{my_account.balance}"
end
```

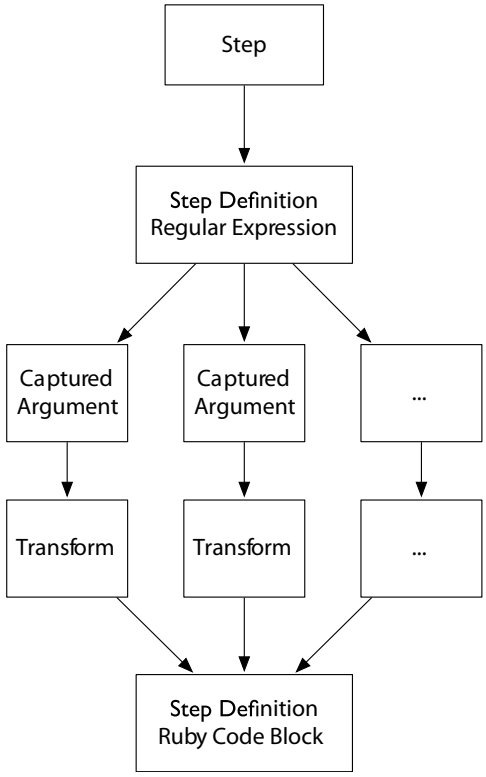


Figure 6—Transforms: how do they work?

Great! That code looks much cleaner and easier to read. Introducing the transform has brought in a new kind of duplication, though. The regular expression that we use to capture the number is now duplicated: we have \d+ both in the step definition and in the transform's definition. This could be a problem if we wanted, for example, to start using cents as well as dollars in our features; we'd have to change the regular expression in the step definition and in the transform. Fortunately, Cucumber allows us to define the regular expression once, in the transform, and then reuse it in the step definition, like this:

Download step_definitions_inside/07/features/step_definitions/steps.rb

```
CAPTURE_A_NUMBER = Transform /^\d+$/ do |number|
  number.to_i
end

Given /^I have deposited \$(#{CAPTURE_A_NUMBER}) in my account$/ do |amount|
  my_account = Account.new
  my_account.deposit(amount)
```

```
  my_account.balance.should eq(amount),
    "Expected the balance to be #{amount} but it was #{my_account.balance}"
end
```

We store the result of calling Transform in the constant CAPTURE_A_NUMBER and then use that constant as we build the regular expression in the step definition. As well as making it easier to change and reuse this capturing regular expression in the future, this refactoring makes it more obvious to someone reading the step definition that this argument will be transformed.

We can tidy this up a little further by moving the dollar sign into the transform's capture. 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.

Download step_definitions_inside/08/features/step_definitions/steps.rb

```
CAPTURE_CASH_AMOUNT = Transform /^\$(\d+)$/ do |digits|
  digits.to_i
end

Given /^I have deposited (#{CAPTURE_CASH_AMOUNT}) in my account$/ do |amount|
  my_account = Account.new
  my_account.deposit(amount)
  my_account.balance.should eq(amount),
    "Expected the balance to be #{amount} but it was #{my_account.balance}"
end
```

Notice that we've used a capture group inside the transform to separate the numbers from the currency symbol. This is how we tell Cucumber that we're interested in transforming only that part of the capture we've been passed, so that's all that will be passed into our block. If we wanted to capture the currency symbol as well, we could put another capture group around it, and it would be yielded to the transform's block as another argument:

```
# encoding: utf-8"
CAPTURE_CASH_AMOUNT = Transform /^(£|\$|€)(\d+)$/ do | currency_symbol, digits |
  # Obviously we have to create a Currency::Money class to make this work.
  Currency::Money.new(digits, currency_symbol)
end
```

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.