Chris Van Bael, POLTEQ

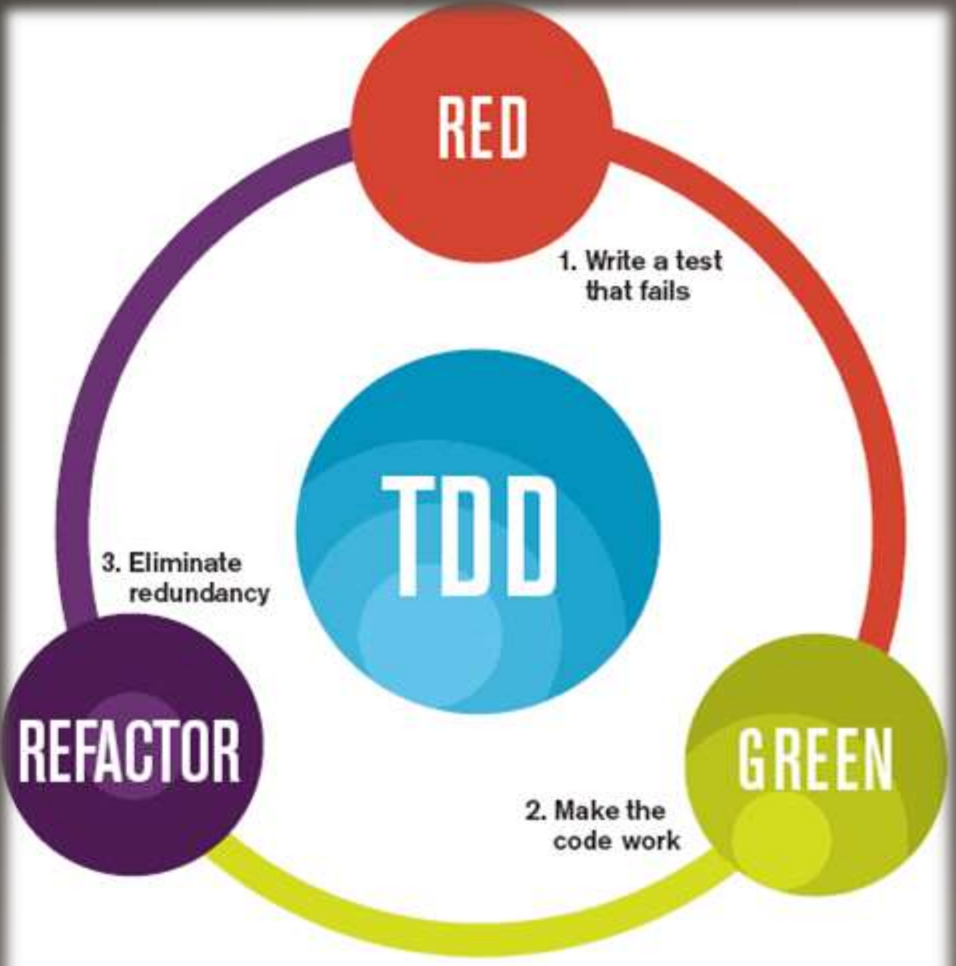
Gherkin Tips & Tricks

Overview

- Introduction to Behavior Driven Development
- Feature file tips
- Background tips
- Scenario tips
- Step tips
- Tag tips
- Step definition tips
- Generic tips

Introduction to BDD

Extends Test Driven Development



The mantra of Test-Driven Development (TDD) is "red, green, refactor."

TDD: not so easy

What to test, what not?

How much in one test?

Why does this test fail?

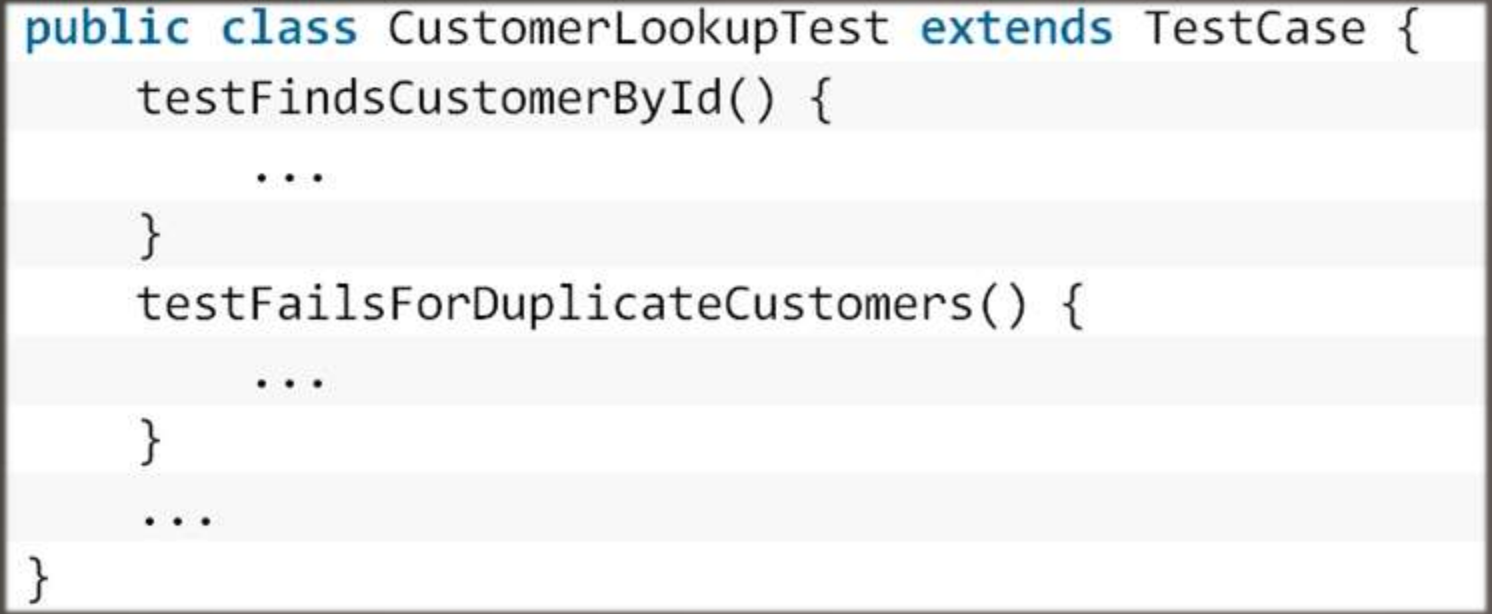
What to call the tests?

Enter Dan North

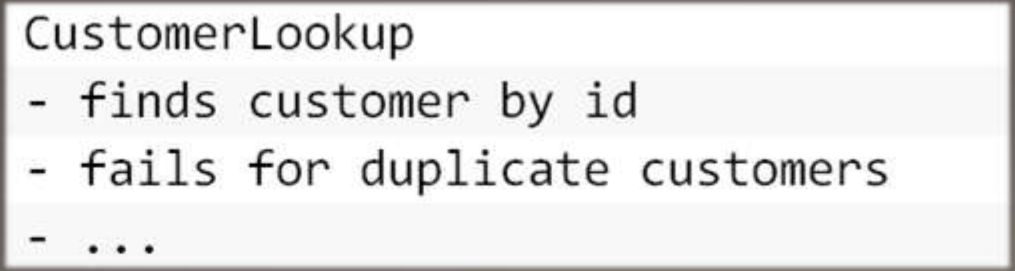
- Using agiledox, he made the mindshift from "test" to "behavior"

- In 3 easy steps

1: test method names should be sentences



```
public class CustomerLookupTest extends TestCase {
    testFindsCustomerById() {
        ...
    }
    testFailsForDuplicateCustomers() {
        ...
    }
    ...
}
```



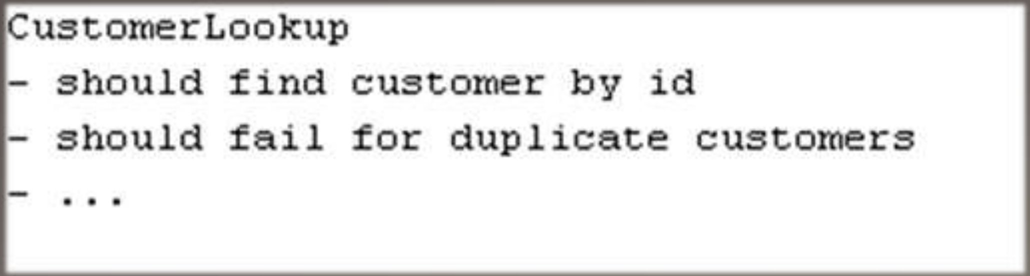
```
CustomerLookup
- finds customer by id
- fails for duplicate customers
- ...
```

2: focus test methods with simple sentences

- Start test methods with “should”
  ⇨ the class should do something!

- If you struggle with the name, perhaps the behavior may belong somewhere else?

- “Should” challenges the premise of the test:
  “should it? really?”



```
CustomerLookup
- should find customer by id
- should fail for duplicate customers
- ...
```

3: “Behavior” is more useful than “test”

- If the test methods don’t comprehensibly describe the behavior of the system, they are lulling you into a false sense of security.

- Replace “test” with “behavior”

- What is a test?

  It’s a sentence describing the next behavior you are interested in.

BDD: much easier!

What to test, what not?

Covered all behavior?

How much in one test?

Simple sentence

Why does this test fail?

Read the method name

What to call the tests?

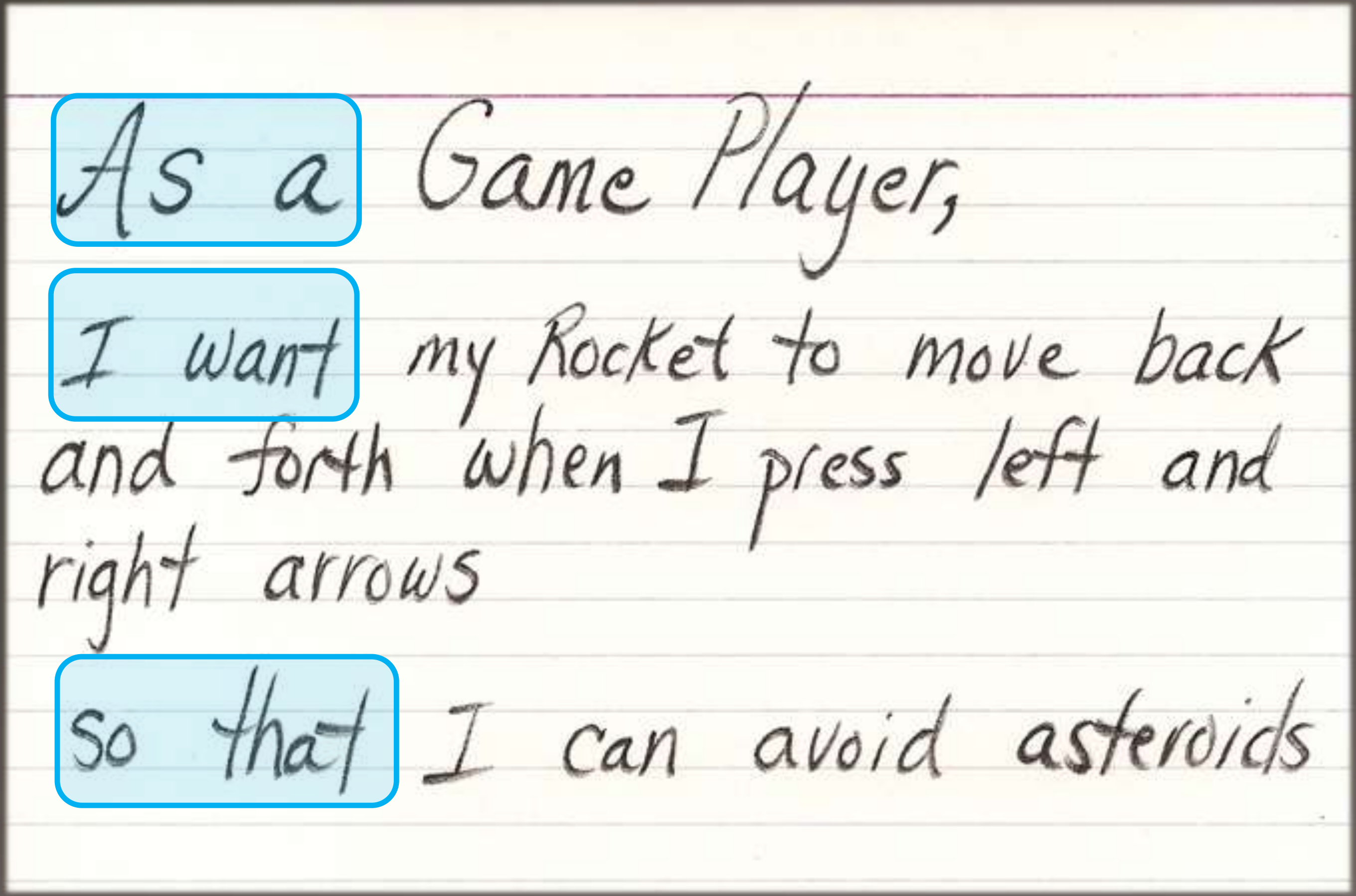
CustomerBehavior - should fail for ...

Priorities?

- Think about the business value
- When writing code:
  What's the next most important thing the system *doesn't* do yet?

⇨BDD provides a ubiquitous language for analysis
- eliminate ambiguity & miscommunication
- useful for analysts, developers, testers, business

Fits with user stories



As a Game Player,
I want my Rocket to move back and forth when I press left and right arrows
so that I can avoid asteroids

But how to define the acceptance criteria?

How to capture acceptance criteria?

Example story of ATM

+Title: Customer withdraws cash+

As a customer,

I want to withdraw cash from an ATM,

so that I don’t have to wait in line at the bank.

When is this story complete?

- sufficient credit
- overdrawn
- overdrawn within credit limit
- sufficient credit, but over daily limit
- …

Example scenarios of ATM

+Scenario 1: Account is in credit+

Given the account is in credit

And the card is valid

And the dispenser contains cash

When the customer requests cash

Then ensure the account is debited

And ensure cash is dispensed

And ensure the card is returned

+Scenario 2: Account is overdrawn past the overdraft limit+

Given the account is overdrawn

And the card is valid

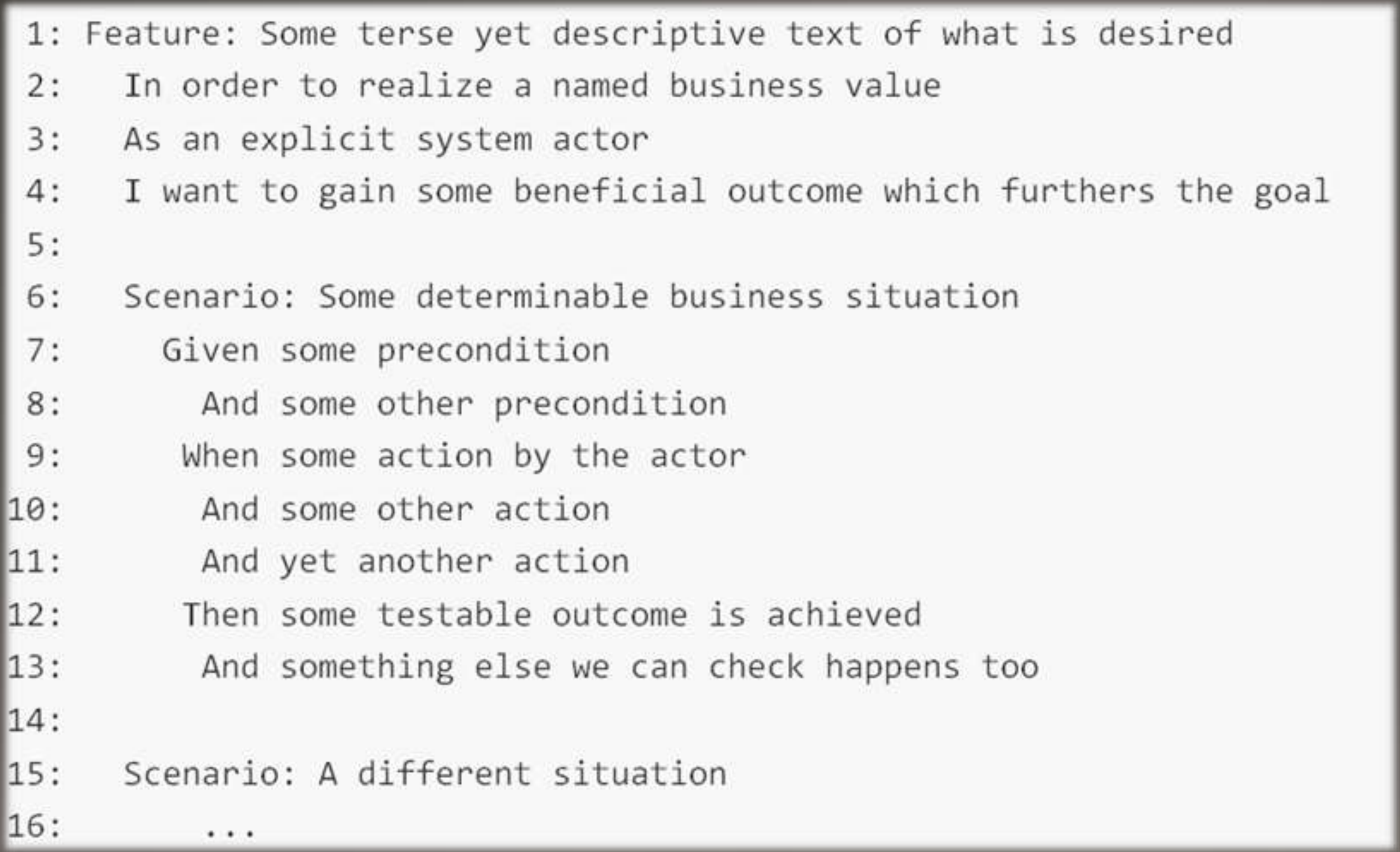
When the customer requests cash

Then ensure a rejection message is displayed

And ensure cash is not dispensed

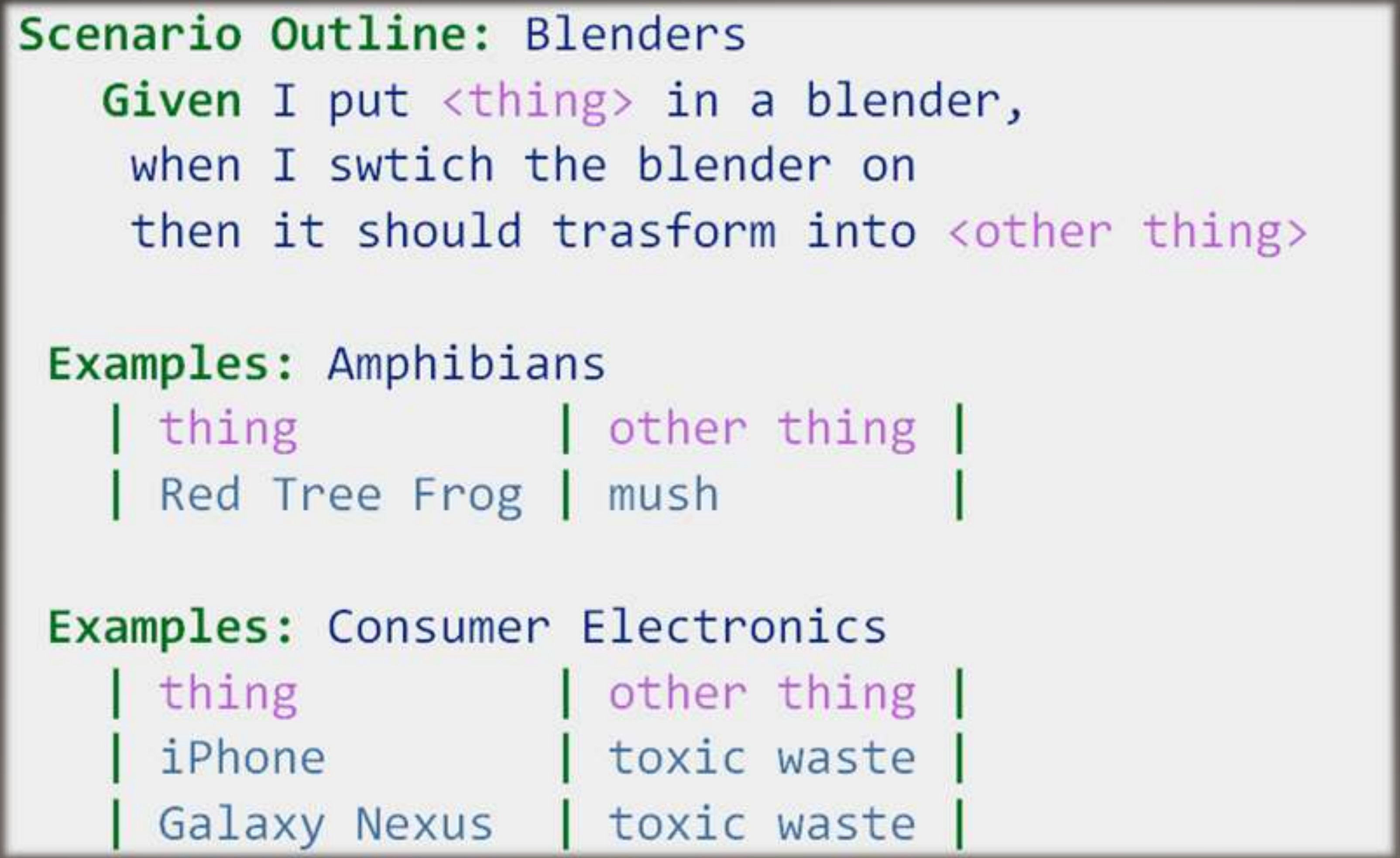
And ensure the card is returned

On features and scenarios …



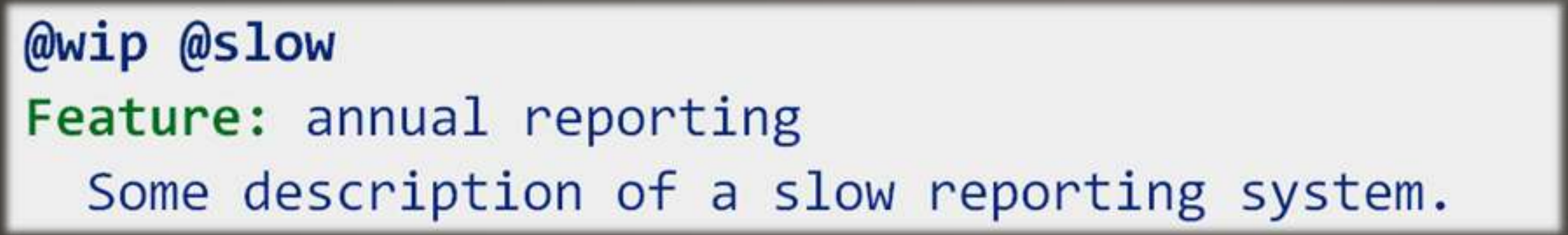
```
Feature: Some terse yet descriptive text of what is desired
  In order to realize a named business value
  As an explicit system actor
  I want to gain some beneficial outcome which furthers the goal

  Scenario: Some determinable business situation
    Given some precondition
      And some other precondition
     When some action by the actor
      And some other action
      And yet another action
     Then some testable outcome is achieved
      And something else we can check happens too

  Scenario: A different situation
      ...
```

On examples …



```
Scenario Outline: Blenders
   Given I put <thing> in a blender,
    when I swtich the blender on
    then it should trasform into <other thing>

 Examples: Amphibians
   | thing          | other thing |
   | Red Tree Frog  | mush        |

 Examples: Consumer Electronics
   | thing          | other thing |
   | iPhone         | toxic waste |
   | Galaxy Nexus   | toxic waste |
```

On tags …



```
@wip @slow
Feature: annual reporting
  Some description of a slow reporting system.
```

- Tag selection on the command-line:
  - --tags @wip,@slow

    Will select all cases tagged *either* with “wip” or “slow”
  - --tags @wip --tags @slow

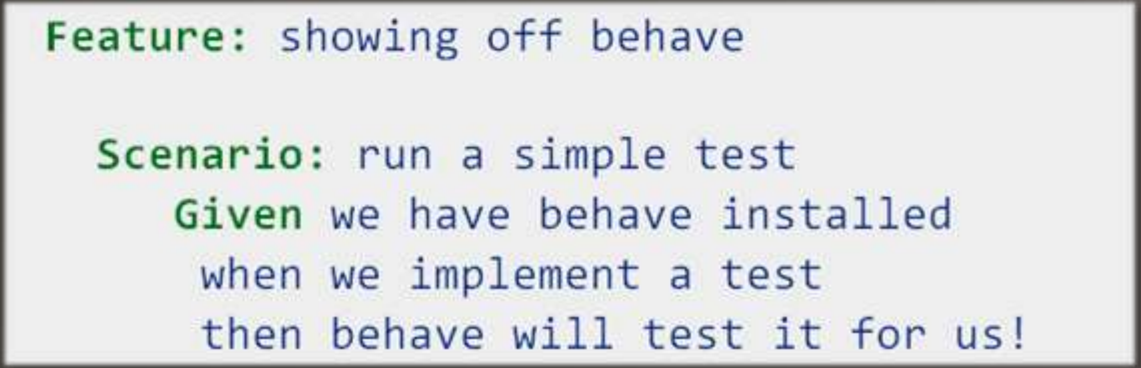
    Will select all cases tagged *both* “wip” and “slow”
  - --tags ~@slow

    Will select all cases *except* the slow ones

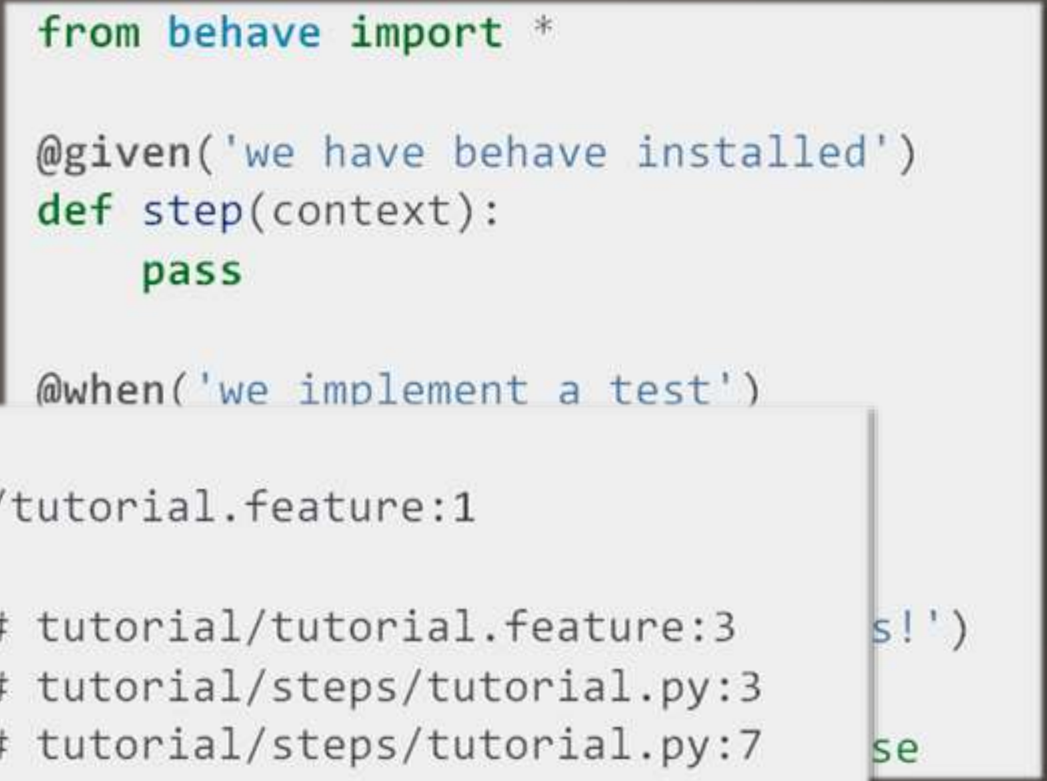
Gherkin is a standard (sort of …)

- There is no ISO nor IEEE standard
- But it is used in several tools:
    - Cucumber (Ruby)
    - Behave, Lettuce (Python)
    - Jbehave (Java)
    - Nbehave (.NET)
    - Javascript (Vows-BDD)
    - …
- The grammar exists in over 40 spoken languages: Arabic, German, French, Dutch, …
  But also: Welsh, Pirate, LOLCAT, Scouse, …

On automation …



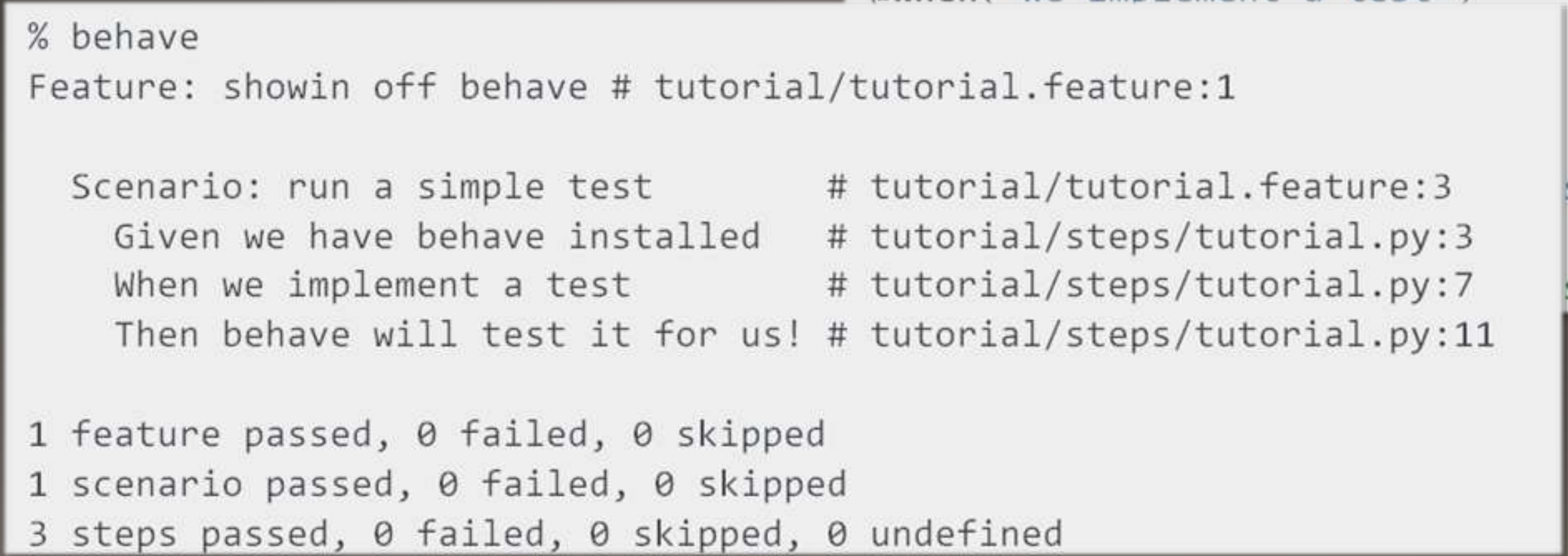
```
Feature: showing off behave

  Scenario: run a simple test
     Given we have behave installed
      when we implement a test
      then behave will test it for us!
```



```
from behave import *

@given('we have behave installed')
def step(context):
    pass

@when('we implement a test')
```

```
s!')
```

```
se
```



```
% behave
Feature: showin off behave # tutorial/tutorial.feature:1

  Scenario: run a simple test        # tutorial/tutorial.feature:3
    Given we have behave installed   # tutorial/steps/tutorial.py:3
    When we implement a test         # tutorial/steps/tutorial.py:7
    Then behave will test it for us! # tutorial/steps/tutorial.py:11

1 feature passed, 0 failed, 0 skipped
1 scenario passed, 0 failed, 0 skipped
3 steps passed, 0 failed, 0 skipped, 0 undefined
```

Not limited to web applications!

- You can use all sort of ‘helpers’:
  - Selenium Webdriver for web testing
  - Aruba (Ruby), Pexpect (Python) for command line testing
  - Swinger for GUI testing of JAVA/Swing applications

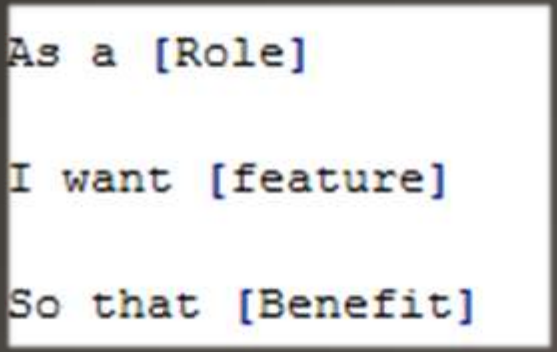
- Use the power of Ruby to test
  - XML interfaces (i.e. webservices)
  - Command line and mainframe applications

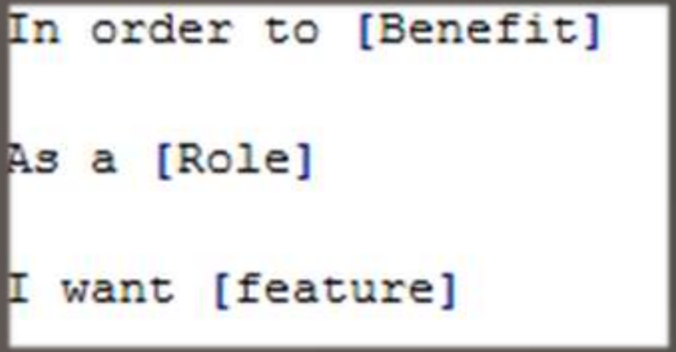
Feature file tips

Avoid long descriptions

- Features should have a short and sensible title and description
- This improves readability
- 1 sentence describing scope and content

Choose single format


```
As a [Role]

I want [feature]

So that [Benefit]
```


```
In order to [Benefit]

As a [Role]

I want [feature]
```

- Pick one format and stick to it
- Always include benefit: makes it easier to decide the business value

Features

…not big portions of the application !

- One feature per file
- Reflect the feature in the file name
- In larger teams preference towards smaller feature files

Domain language

- Involve the customers
- Use their domain language
- Involve them in writing the user stories
- Or at least have them review the user stories
- Keep the language consistent

Organization

- Organize your Features and Scenarios with the same discipline like you would organize code
- For example: speed
    - Fast: < 1/10 s
    - Slow: <1 s
    - Glacial: longer
- Put them in separate subdirectories
- Or tag them

Background tips

Use backgrounds

- It reduces repetitions in the feature file
- But keep them short (max 4 lines):
  - user has to keep background in mind while reading/writing the scenarios
- And don't include technical stuff ➔ feature file is about the user
  - Start/stop webserver, clear tables, … can be implemented in the step definitions
- Don't use a background if you have only one scenario
- Don't mix backgrounds with @before hooks

Scenario tips

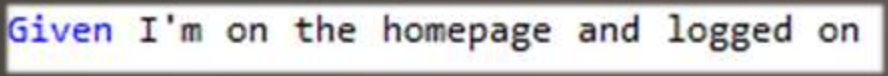
Scenarios and steps

- Scenario vs Scenario Outline
    - 1 example: scenario
    - More examples: scenario outline + table
- Keep scenarios short; hide implementation details
- Given ➔ When ➔ Then is the correct order
- Declarative steps vs imperative steps

Step tips

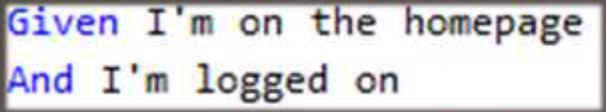
Step tips

- AND/OR are keywords, don’t use them within a step



```
Given I'm on the homepage and logged on
```

Should be



```
Given I'm on the homepage
And I'm logged on
```

- Cover happy and non-happy paths

  Testing is more than only proving it works

Refactor

- Your library of steps will increase in time
  - ➔ Try to generalize your steps to increase reuse

- Your understanding of the domain will increase
  - ➔ Update your language and the steps

Tag tips

Use tags

- Tags allow you to organize your features and scenarios
- You can have multiple tags per feature or scenario

    ➔ Never tag the background

- Feature tags are also valid for all child scenarios

    ➔ Don’t tag scenario with same tag as feature

- Think hard on the benefit of tagging a feature

    ➔ Using the User Story number might be useful

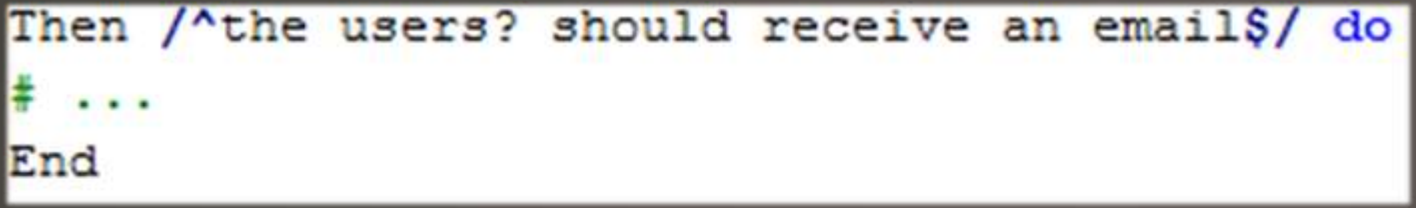
Possible tag categories

- Frequency of Execution: @checkin, @hourly, @daily, @nightly
- Dependencies: @local, @database, @fixtures, @proxy
- Progress: @wip, @todo, @implemented, @blocked
  - ➔ Keep these up to date ! (if not, don’t use them)
- Level: @functional, @acceptance, @smoke, @sanity
- Environment: @integration, @test, @stage, @live

Step definition tips

Use flexible pluralization

- Add a ? after the pluralized word

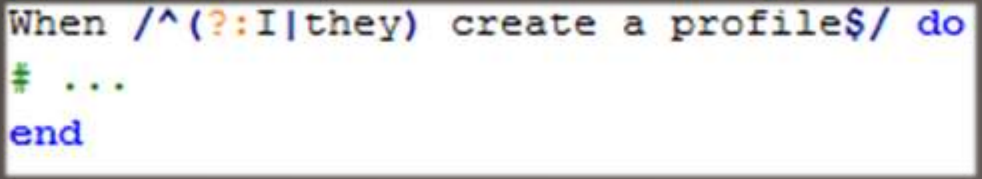


```
Then /^the users? should receive an email$/ do
# ...
End
```

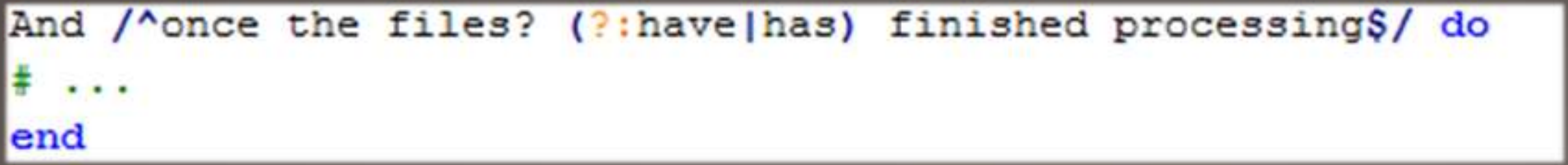
- ? Specifies that you are looking for zero or more of the proceeding characters
- This way it matches both user and users

Use non-capturing groups

- Instead of (some text), use (?:some text)
- Result is not captured and not passed as an argument to your step definition
- Useful in combination with alternation



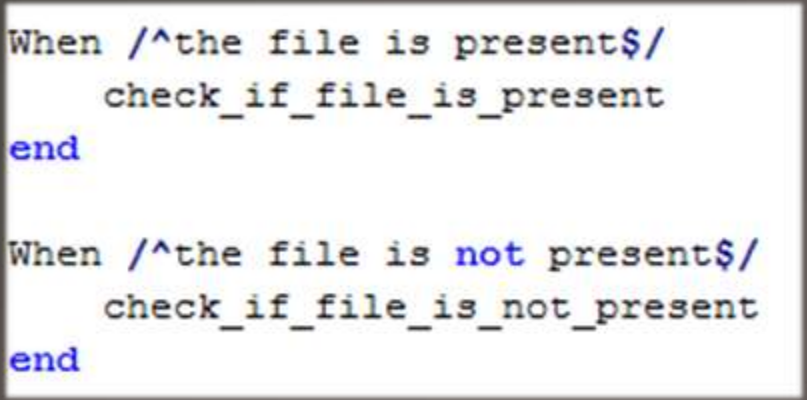
```
When /^(?:I|they) create a profile$/ do
# ...
end
```



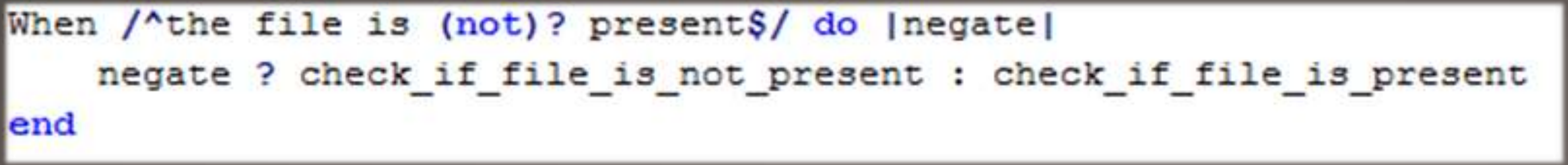
```
And /^once the files? (?:have|has) finished processing$/ do
# ...
end
```

Consolidate step definitions

- You can test both positive and negative assertions



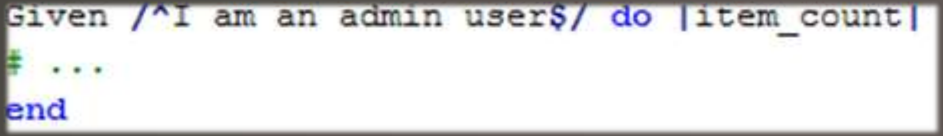
```
When /^the file is present$/
    check_if_file_is_present
end

When /^the file is not present$/
    check_if_file_is_not_present
end
```



```
When /^the file is (not)? present$/ do |negate|
    negate ? check_if_file_is_not_present : check_if_file_is_present
end
```

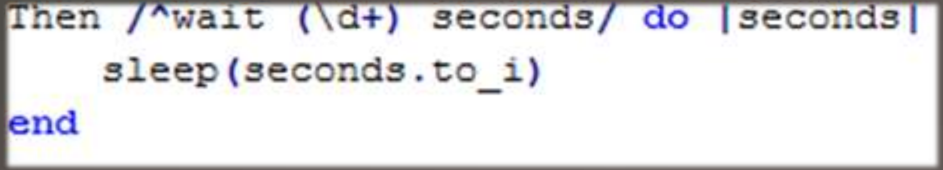
Use unanchored regular expressions

- Normally you anchor start with ^ and end with $



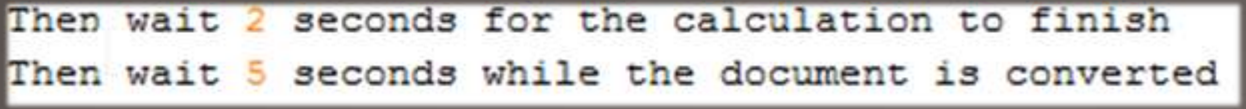
```
Given /^I am an admin user$/ do |item_count|
# ...
end
```

- Sometimes it might be useful to omit one



```
Then /^wait (\d+) seconds/ do |seconds|
    sleep(seconds.to_i)
end
```

- To increase readability and write flexible expressive steps



```
Then wait 2 seconds for the calculation to finish
Then wait 5 seconds while the document is converted
```

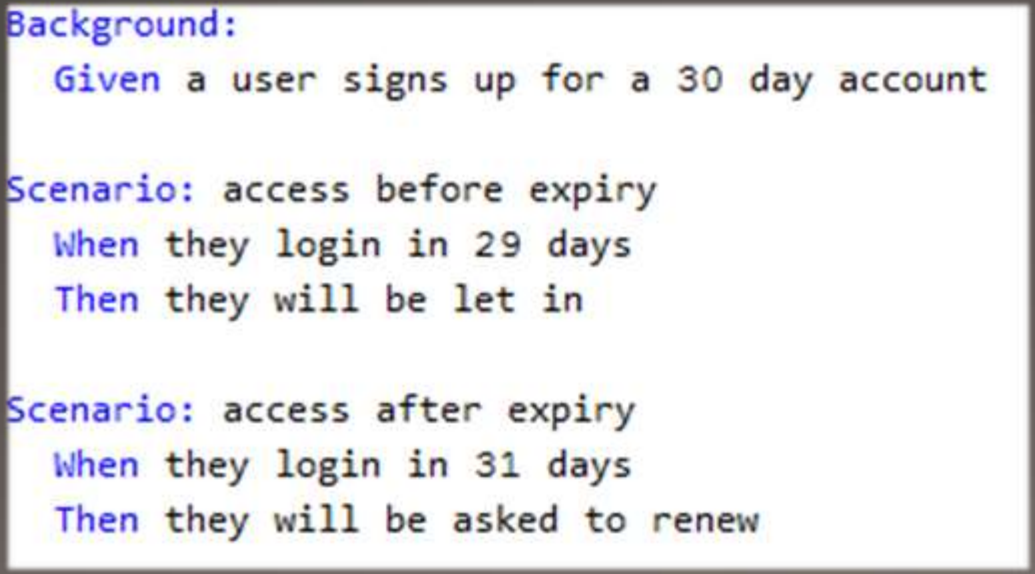
- Don’t misuse this or over do it!

Be DRY

- Don’t Repeat Yourself
- Refactor
- Reuse step definitions
  - Within a project across features
  - Perhaps even across projects

Parse date/time in a natural way

- Use a library for parsing dates and times
- Ruby: Chronic, Python: parsedatetime or pyparsing



```
Background:
  Given a user signs up for a 30 day account

Scenario: access before expiry
  When they login in 29 days
  Then they will be let in

Scenario: access after expiry
  When they login in 31 days
  Then they will be asked to renew
```

Generic tips

Discipline

- Treat your code as production code
- Refactor when necessary
- Run your tests as often as possible
- Don’t be too smart: somebody needs to understand it next year

Checklists

- Chris.vanbael@Polteq.com
- www.linkedin.com/in/chrisvanbael
- http://bit.ly/1OmmLFr

Questions ?