SCENARIOS AND BEGINNING TDD

TMDB WITH 2 SCENARIOS

http://pastebin/icQGrYCV

```
Feature: User can add movie by
    searching for it in The Movie
    Database (TMDb)
  As a movie fan
  So that I can add new movies without
    manual tedium
  I want to add movies by looking up
    their details in TMDb

Background: Start from the Search
    form on the home page
  Given I am on the RottenPotatoes
    home page
  Then I should see "Search TMDb for
    a movie"

Scenario: Try to add nonexistent
    movie (sad path)
  When I fill in "Search Terms" with
    "Movie That Does Not Exist"
  And I press "Search TMDb"
  Then I should be on the
    RottenPotatoes home page
  And I should see "'Movie That Does
    Not Exist' was not found in TMDb."
Scenario: Try to add existing movie
  When I fill in "Search Terms" with
    "Inception"
  And I press "Search TMDb"
  Then I should be on the
    RottenPotatoes home page
  And I should see "Inception"
  And I should see "PG-13"
```

HAPPY PATH OF TMDB

- Find an existing movie, should return to Rotten Potatoes home page
- But some steps same on sad path and happy path
  - How make it DRY?
  - `Background` means steps performed before *each* scenario

TMDB WITH 2 SCENARIOS

http://pastebin/icQGrYCV

```
Feature: User can add movie by
    searching for it in The Movie
    Database (TMDb)
  As a movie fan
  So that I can add new movies without
    manual tedium
  I want to add movies by looking up
    their details in TMDb

Background: Start from the Search
    form on the home page
  Given I am on the RottenPotatoes
    home page
  Then I should see "Search TMDb for
    a movie"

Scenario: Try to add nonexistent
    movie (sad path)
  When I fill in "Search Terms" with
    "Movie That Does Not Exist"
  And I press "Search TMDb"
  Then I should be on the
    RottenPotatoes home page
  And I should see "'Movie That Does
    Not Exist' was not found in TMDb."
Scenario: Try to add existing movie
  When I fill in "Search Terms" with
    "Inception"
  And I press "Search TMDb"
  Then I should be on the
    RottenPotatoes home page
  And I should see "Inception"
  And I should see "PG-13"
```

SUMMARY

- New feature => UI for feature, write new step definitions, even write new methods before Cucumber can color steps green
- Usually do happy paths first
- Background lets us DRY out scenarios of same feature
- BDD tests behavior; TDD/BDD used together in next chapter to write methods to make all scenarios pass

EXPLICIT VS. IMPLICIT AND IMPERATIVE VS. DECLARATIVE

EXPLICIT VS. IMPLICIT SCENARIOS

- Explicit requirements usually part of acceptance tests => likely explicit user stories and scenarios
- Implicit requirements are logical consequence of explicit requirements, typically integration testing
  - Movies listed in chronological order or alphabetical order?

IMPERATIVE VS. DECLARATIVE SCENARIOS

- Imperative: specifying logical sequence that gets to desired result
  - Initial user stories usually have lots of steps
  - Complicated `When` statements and `And` steps
- Declarative: try to make a Domain Language from steps, and write scenarios declaratively
- Easier to write declaratively as create more steps and more Rails experience

EXAMPLE IMPERATIVE SCENARIO

- Given I am on the RottenPotatoes home page
- When I follow "Add new movie"
- Then I should be on the Create New Movie page
- When I fill in "Title" with "Zorro"
- And I select "PG" from "Rating"
- And I press "Save Changes"
- Then I should be on the RottenPotatoes home page
- When I follow "Add new movie"
- Then I should be on the Create New Movie page
- When I fill in "Title" with "Apocalypse Now"
- And I select "R" from "Rating"
- And I press "Save Changes"
- Then I should be on the RottenPotatoes home page
- And I should see "Apocalypse Now" before "Zorro"

Only 1 step specifying behavior;
Rest are really implementation.
But BDD should be about design

EXAMPLE DECLARATIVE SCENARIO

- Given I have added "Zorro" with rating "PG-13”
- And I have added "Apocalypse Now" with rating "R”
- And I am on the RottenPotatoes home page sorted by title
- Then I should see "Apocalypse Now" before "Zorro" on the Rotten Potatoes home page

Declarative Scenario Needs New Step Definitions

DECLARATIVE SCENARIO -> NEW STEP DEFINITIONS

```
1.Given /I have added "(.*)" with rating "(.*)"/ do |title, rating|
2.  steps %Q{
3.    Given I am on the Create New Movie page
4.    When  I fill in "Title" with "#{title}"
5.    And   I select "#{rating}" from "Rating"
6.    And   I press "Save Changes"
7.  }
8.end
9.
10.Then /I should see "(.*)" before "(.*)" on (.*)/ do |string1, string2,
  path|
11.  step "I am on #{path}"
12.  regexp = /#{string1}.*#{string2}/m #  /m means match across newlines
13.  page.body.should =~ regexp
14.end
```

- As app evolves, **reuse steps** from first few imperative scenarios -> more concise, descriptive declarative scenarios
- Declarative scenarios focus attention on feature being described and tested vs. steps needed to set up test

PITFALLS

- Customers who confuse mock-ups with completed features
  - May be difficult for nontechnical customers to distinguish a polished digital mock-up from a working feature
- Solution: LoFi UI on paper clearly *proposed* vs. implemented

PITFALLS

- Sketches without storyboards
  - Sketches are static
  - Interactions with SaaS app = sequence of actions over time
- "Animating" the Lo-Fi sketches helps prevent misunderstandings before turning stories are into tests and code
  - "OK, you clicked on that button, here's what you see; is that what you expected?"

PITFALLS

- Adding cool features that do not make the product more successful
  - Customers reject what programmers liked
  - User stories help prioritize, reduce wasted effort

PITFALLS

- Trying to predict what code you need before need it
  - BDD: write tests *before* you write code you need, then write code needed to pass the tests
  - No need to predict, wasting development

PITFALLS

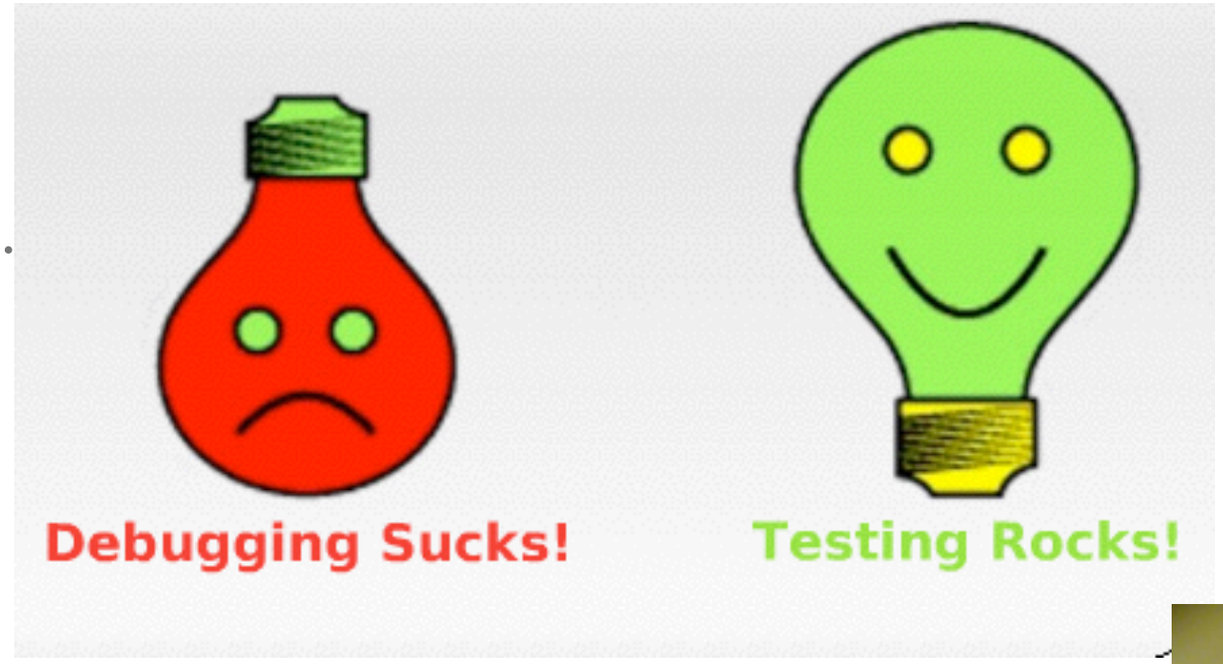
- Careless use of negative expectations
  - Beware of overusing “Then I should not see….”
  - Can’t tell if output is what want, only that it is not what you want
  - Many, many outputs are incorrect
  - Include positives to check results
    “Then I should see …”

PROS AND CONS OF BDD

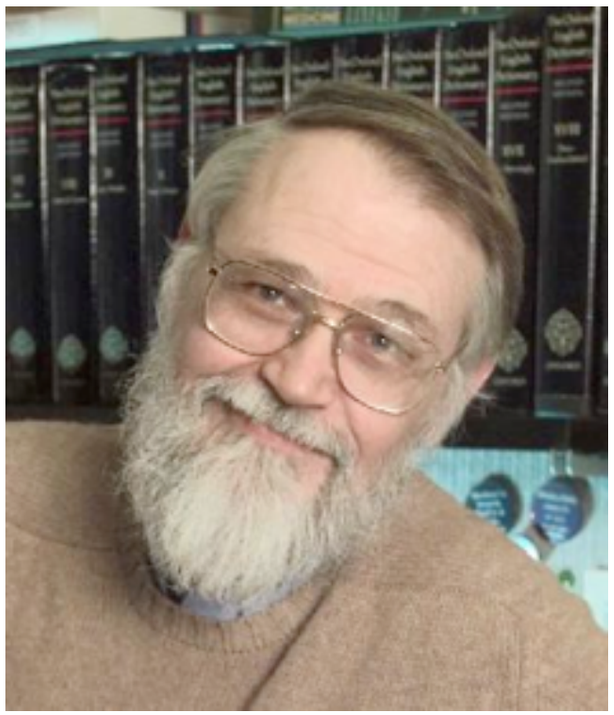
- Pro: BDD/user stories - common language for all stakeholders, including nontechnical
  - 3x5 cards
  - LoFi UI sketches and storyboards
- Pro: Write tests before coding
  - Validation by testing vs. debugging
- Con: Difficult to have continuous contact with customer?
- Con: Leads to bad software architecture?
  - Will cover patterns, refactoring 2nd half of course

BEGINNING TDD WITH UNIT TESTING

RED- RED- GREEN-GREEN-REFACTOR



Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.



Testing can never demonstrate the _____ of errors in software, only their _____

Survey Finds 58% of Software Bugs Result from Test Infrastructure and Process, Not Design Defects

Developers Prefer Taxes to Dealing with Software Testing

Sunnyvale, Calif. — June 2, 2010 Electric Cloud®, the leading provider of software production management (SPM) solutions, today released the results of a survey conducted in partnership with Osterman Research showing that the majority of software bugs are attributed to poor testing procedures or infrastructure limitations rather than design problems. Additionally, the software test process is generally considered an unpleasant process, with software development professionals rating the use of their companies' test systems more painful than preparing taxes.

Fifty-eight percent of respondents pointed to problems in the testing process or infrastructure as the cause of their last major bug found in delivered or deployed software, not design defects.

Specifically, the survey found:

- Completely automated software testing environments are still rare, with just 12 percent of software development organizations using fully automated test systems. Almost 10 percent reported that all testing was done manually.

Testing Today

- Before
  - developers finish code, some ad-hoc testing
  - "toss over the wall to Quality Assurance [QA]"
  - QA people manually poke at software
- Today/Agile
  - testing is part of *every* Agile iteration
  - developers responsible for testing own code
  - testing tools & processes highly automated;
  - QA/testing group improves *testability* & *tools*

Testing Today

➤Before

➤

➤testing tools & processes highly automated;

➤QA/testing group improves *testability* & *tools*

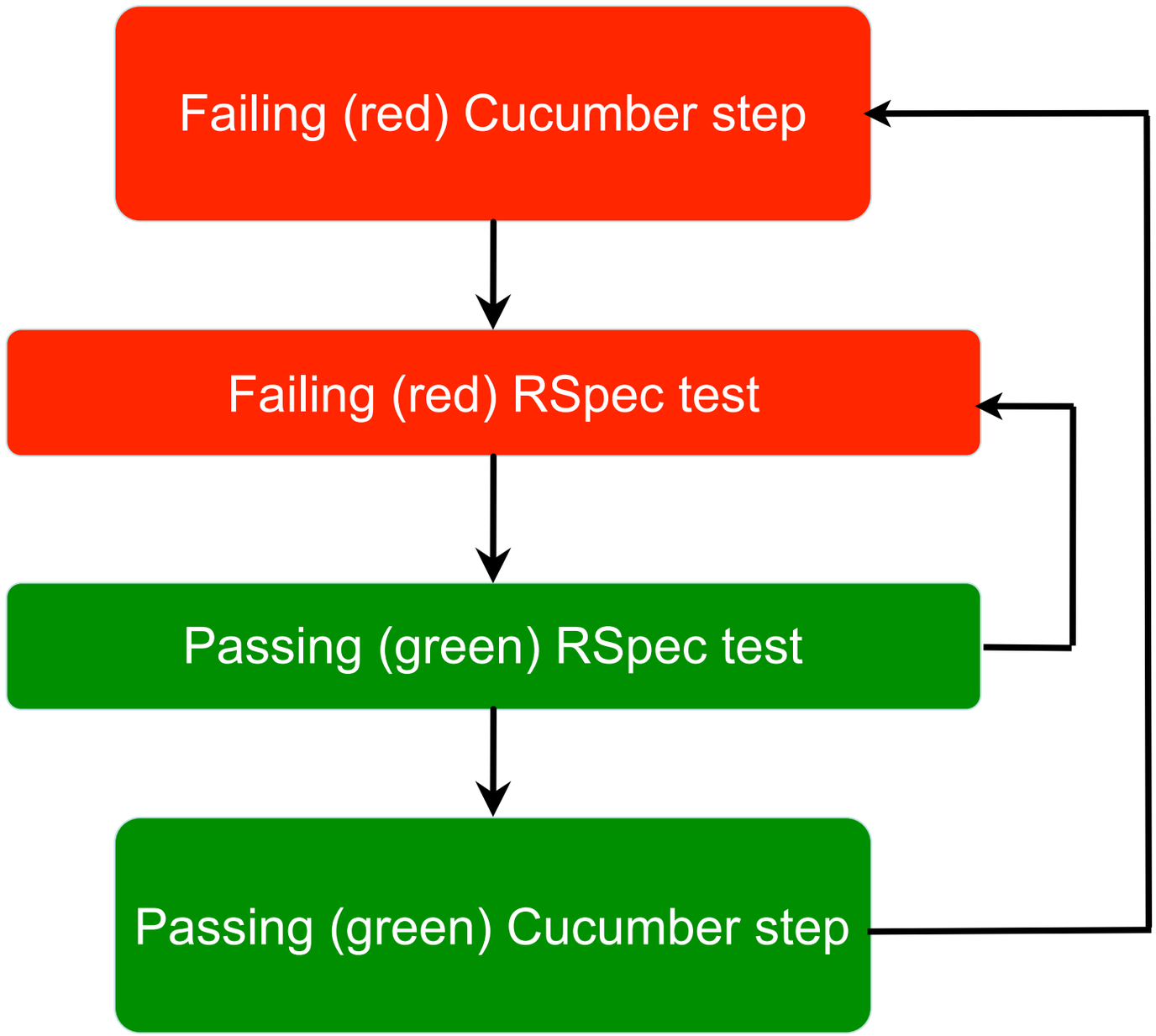
*Software Quality is the result of a good **process**, rather than the responsibility of one specific group*

BDD+TDD: The Big Picture

- Behavior-driven design (BDD) and Integration Testing
  - develop user stories to describe features
  - via Cucumber, user stories become *acceptance tests* and *integration tests*
- Test-driven development (TDD)- now including unit testing
  - *step definitions* for new story, may require new code to be written
  - TDD says: write unit & functional tests for that code *first, **before*** the code itself
  - that is: write tests for *the code you wish you had*
  - via Cucumber and Rspec

Cucumber & RSpec

- Cucumber describes *behavior* via features & scenarios (*acceptance level/integration level*)
- RSpec tests individual modules that contribute to those behaviors (*unit level*)



*

FIRST, TDD, and Getting Started With RSpec

Unit tests should be FIRST

- **F**ast
- **I**ndependent
- **R**epeatable
- **S**elf-checking
- **T**imely

Unit tests should be FIRST

➤Fast: run (subset of) tests quickly (since you'll be running them *all the time*)

➤Independent: no tests depend on others, so can run *any subset* in *any order*

➤**R**epeatable: run N times, get same result (to help isolate bugs and enable automation)

➤**S**elf-checking: test can *automatically* detect if passed (*no human checking* of output)

➤**T**imely: written about the same time as code under test (with TDD, written *first!*)