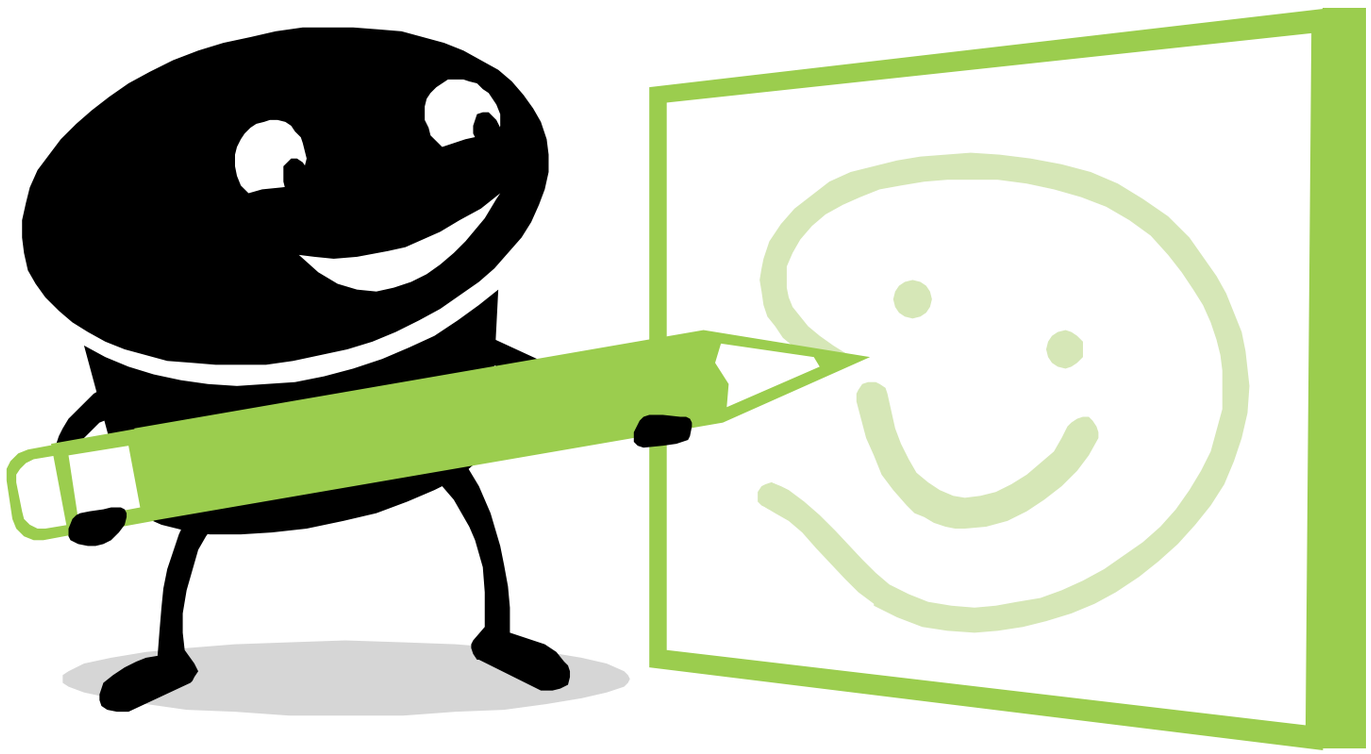
-Talking to the Customer-
Learning and Recording
What They Want

Then turn ideas into tests!

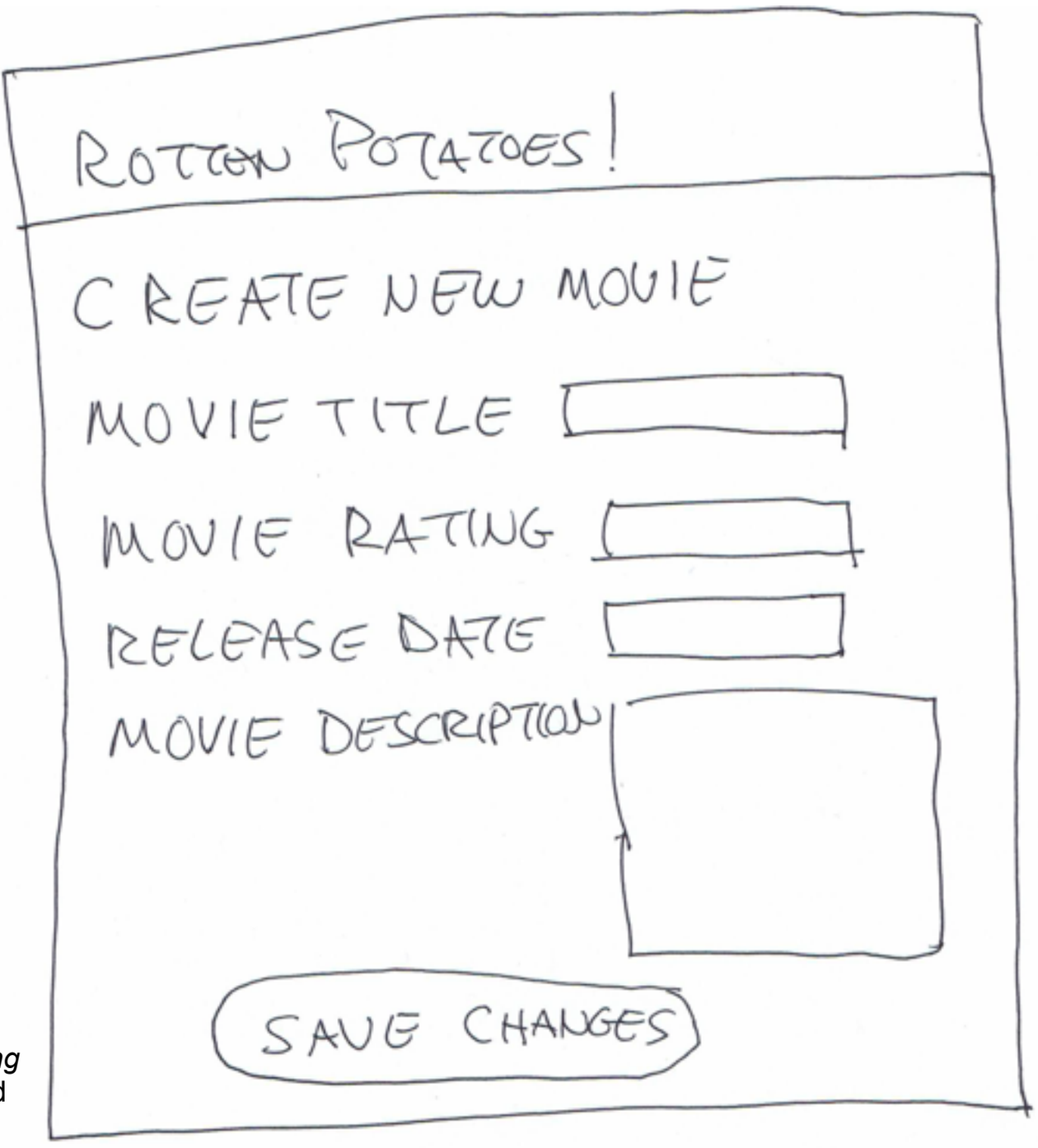
Announcements

- HW2- DUE LAST NIGHT
- Iter 0-3 due Sunday
  - Lo-Fi UI Diagrams and Storyboards
  - Team Evaluation: Due Wed
  - Customer Evaluation: Due Wed
  - Quick demo of projects: Tues
- OPTIONAL HOMEWORKS WILL BE POSTED
- FIRST TEST: THURSDAY 3/2
  - Review guide will be posted

LO-FI UI SKETCHES AND STORYBOARDS



LO-FI UI EXAMPLE

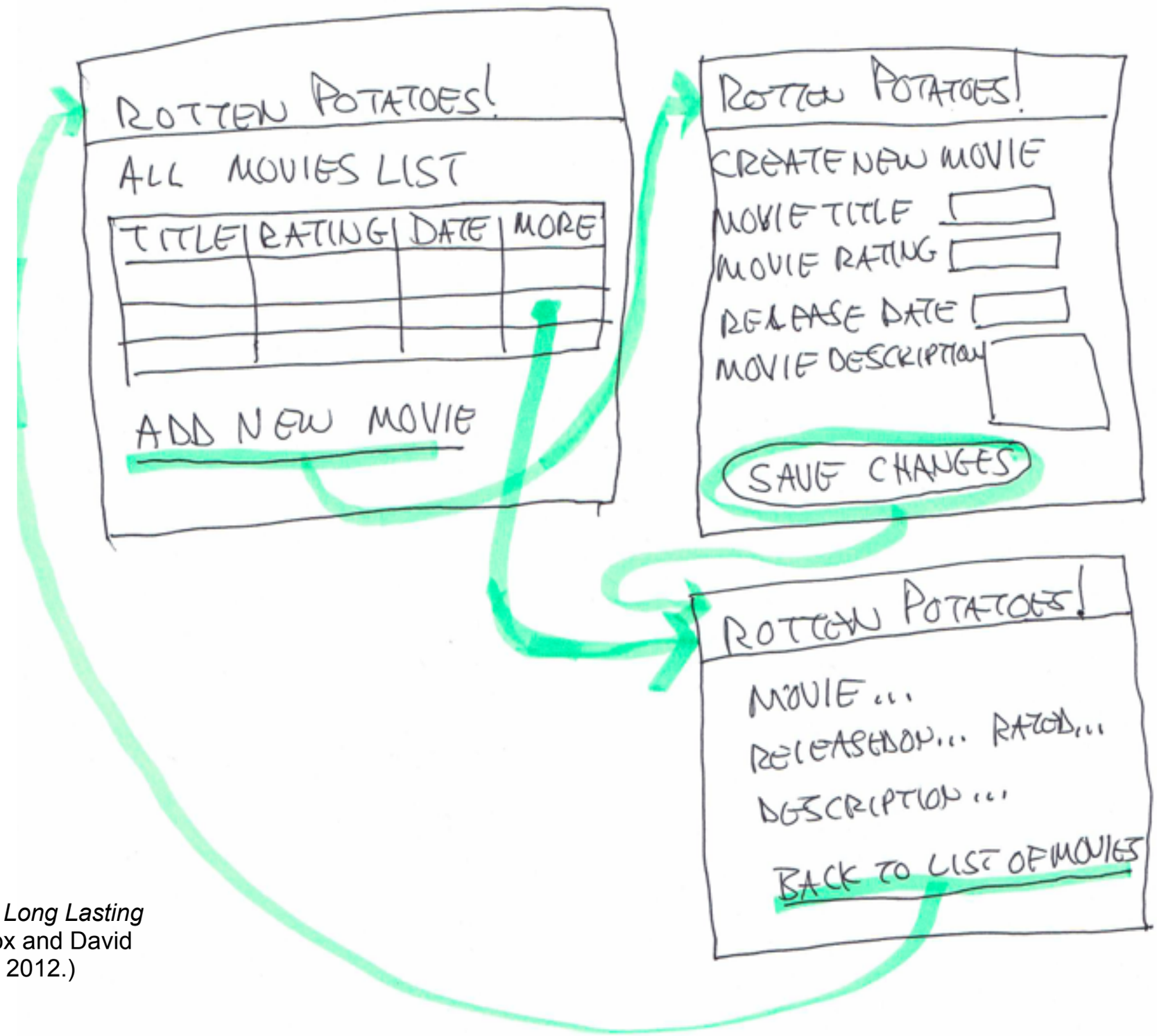


(Figure 4.3, *Engineering Long Lasting Software* by Armando Fox and David Patterson, Alpha edition, 2012.)

STORYBOARDS

- Need to show how UI changes based on user actions
- HCI => “storyboards”
- Like scenes in a movie
- But not linear

EXAMPLE STORYBOARD



(Figure 4.4, *Engineering Long Lasting Software* by Armando Fox and David Patterson, Alpha edition, 2012.)

LO-FI TO HTML

- Tedious to do sketches and storyboards,
  but easier than producing HTML!
  - Also less intimidating to nontechnical stakeholders =>
    More likely to suggest
    changes to UI if not code behind it
  - More likely to be happy with ultimate UI
- Next steps: More on CSS (Cascading Style Sheets) and Haml
  - Make it pretty *after* it works

BEFORE YOU CODE: TEST

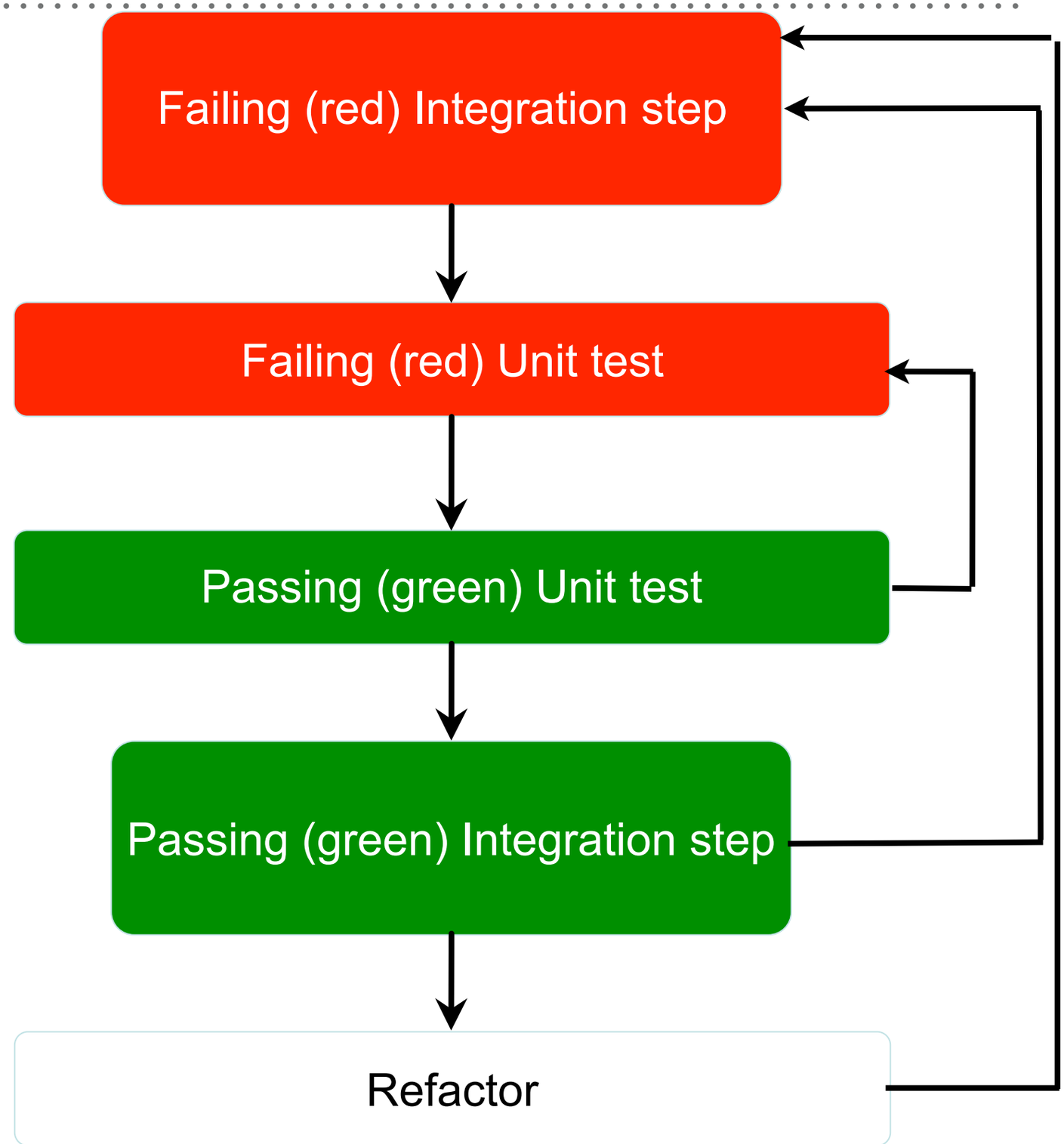
- Test Driven Development
  - Write tests
  - THEN write code
- Advantages?
- Disadvantages?

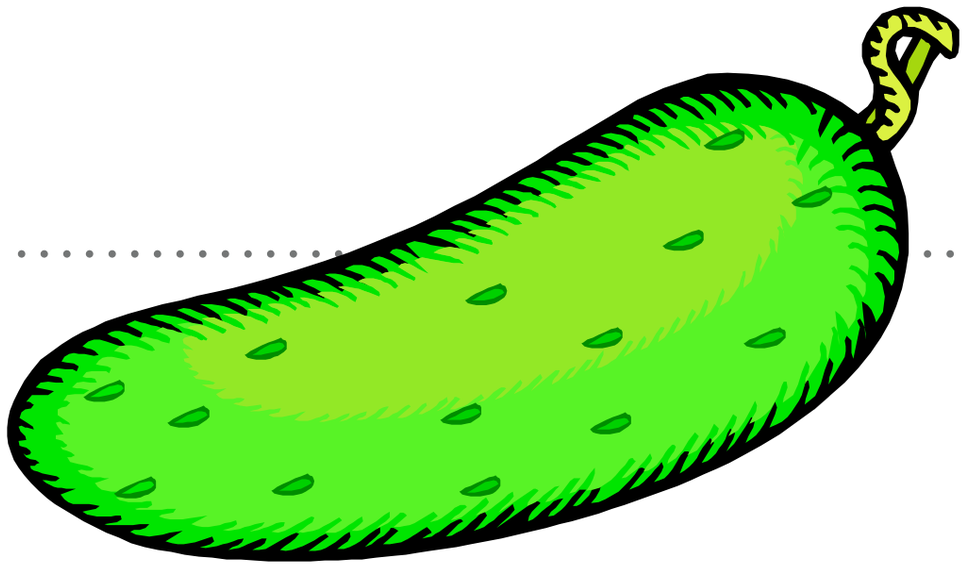
- Levels of Testing
  - Acceptance
  - System
  - Integration
  - Function
  - Unit

Cucumber & RSpec

➤Integration testing describes *behavior* via features & scenarios (supports *behavior driven* design and *test driven development*)

➤Unit testing tests individual modules that contribute to those behaviors (used with *test driven* development)





INTRODUCING CUCUMBER

CUCUMBER (TOOL): BIG IDEA

- Tests from customer-friendly user stories
  - Acceptance: ensure satisfied customer
  - Integration: ensure interfaces between modules consistent assumptions, communicate correctly.
- Cucumber meets halfway between customer and developer
  - User stories don't look like code, so clear to customer and can be used to reach agreement
  - Also aren't completely freeform, so can connect to real tests

EXAMPLE USER STORY

```
Feature: User can manually add movie
```

1 Feature

```
Scenario: Add a movie
```

≥1 Scenarios / Feature

```
  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 "Men In Black"

  And I select "PG-13" from "Rating"

  And I press "Save Changes"

  Then I should be on the RottenPotatoes home page

  And I should see "Men In Black"
```

3 to 8 Steps / Scenario

INTRO TO BDD AND TDD

CUCUMBER USER STORY, FEATURE, AND STEPS

- User story: refers to a single feature
- Feature: 1 or more scenarios that show different ways a feature is used
  - Keywords `Feature` and `Scenario` identify the respective components
- Scenario: 3 to 8 steps that describe scenario
- Step definitions: Ruby code that tests steps
  - Usually many steps per step definition

5 STEP KEYWORDS

1. Given steps represent the state of the world before an event: preconditions
2. When steps represent the event (e.g., push a button)
3. Then steps represent the expected outcomes; check if its true
4. / 5. And and But extend the previous step

STEPS, STEP DEFINITIONS, AND REGULAR EXPRESSIONS

- User stories kept in one set of files: steps
- Separate set of files has Ruby code that tests steps: step definitions
- Step definitions are like method definitions, steps of scenarios are like method calls

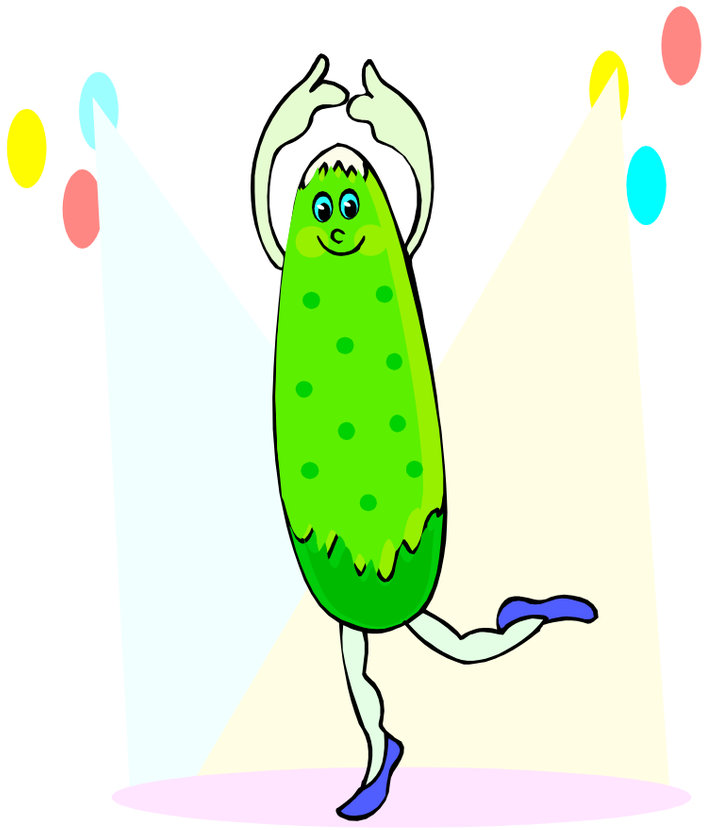
- How match steps with step definitions?
- ***Regexes to match the English phrases in steps of scenarios to step definitions!***
  - `Given /^(?:|{}I )am on (.+)\$/`
  - `“I am on the Rotten Potatoes home page”`

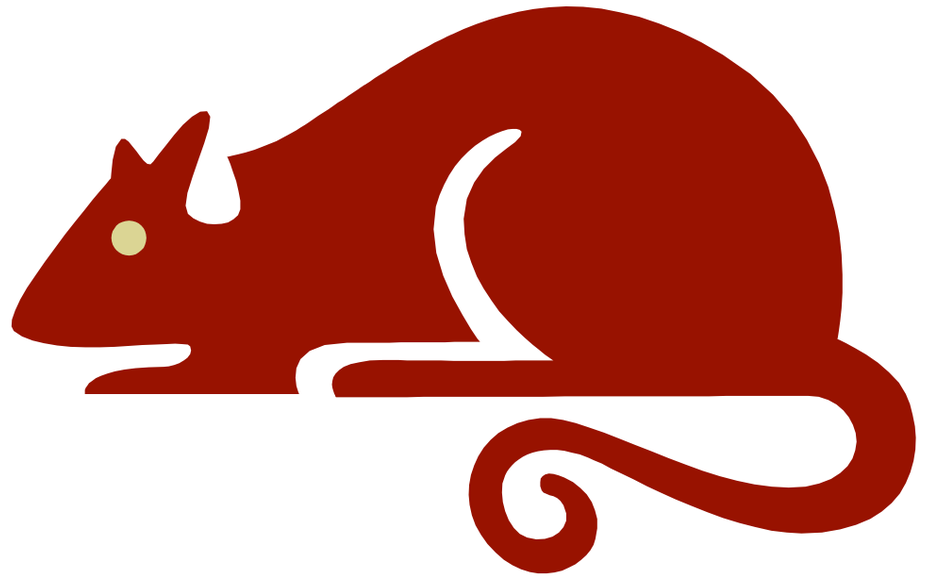
RED-YELLOW-GREEN ANALYSIS

- Cucumber colors steps
- Green for passing
- Yellow for not yet implemented
- Red for failing
  (then following steps are Blue)
- Goal: Make all steps green for pass

  (Hence green vegetable for name of tool)

RUNNING CUCUMBER AND INTRODUCING CAPYBARA





CAPYBARA (SELENIUM ALSO RECOMMENDED)

- Need tool to act like user that pretends to be user follow scenarios of user story
- Capybara simulates browser
  - Can interact with app to receive pages
  - Parse the HTML
  - Submit forms as a user would
- Cannot handle JavaScript
  - Other tool (Webdriver) can handle JS, but it runs a lot slower, won't need yet

REVIEW

- Agile – prototypes, iterate with customer
- BDD – Design of app before implementation
- User Story – all stakeholders write what features want on 3x5 cards
- Cucumber – magically turns 3x5 card user stories into acceptance tests for app
- Capybara - pretends to be a human interacting with a web browser

VIDEO TO WATCH BEFORE YOU START IMPLEMENTING

- Add feature to cover existing functionality
  - Kinda backwards- Just done for pedagogic reasons
- (Watch screencast:

  http://vimeo.com/34754747)

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

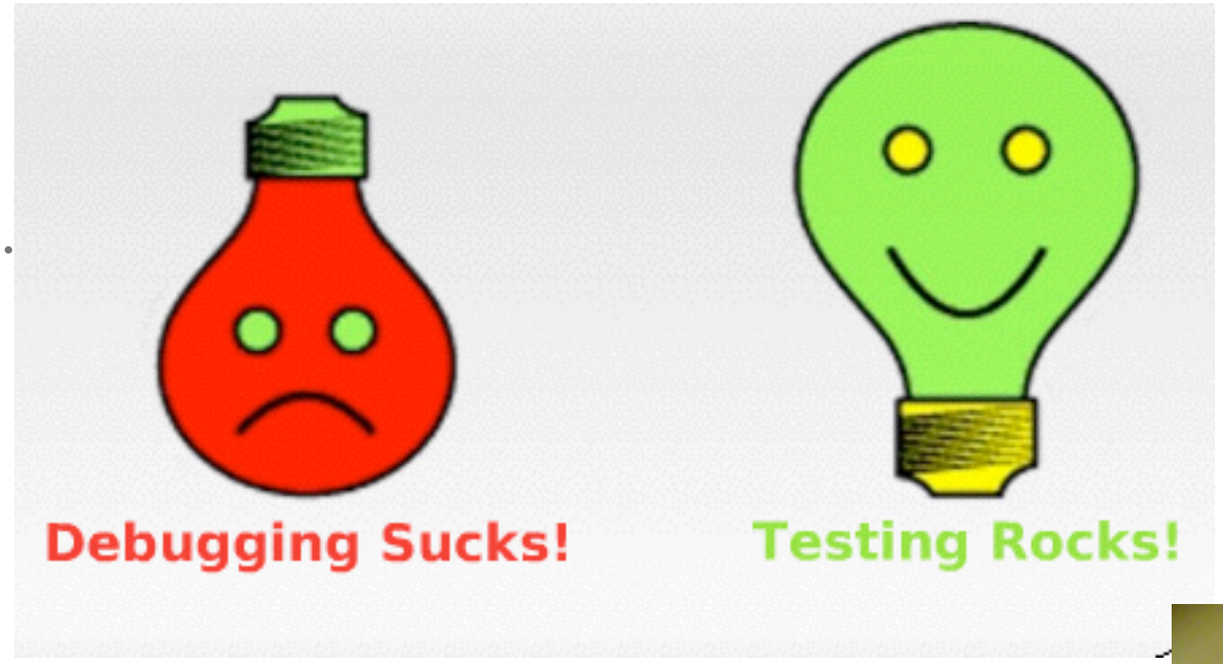
- Careless use of negative expectations
  - Beware of overusing “Then I should not see….”
  - Can’t tell if output is what want, only that it is not what you want
  - Many, many outputs are incorrect
  - Include positives to check results
    “Then I should see …”

PROS AND CONS OF BDD

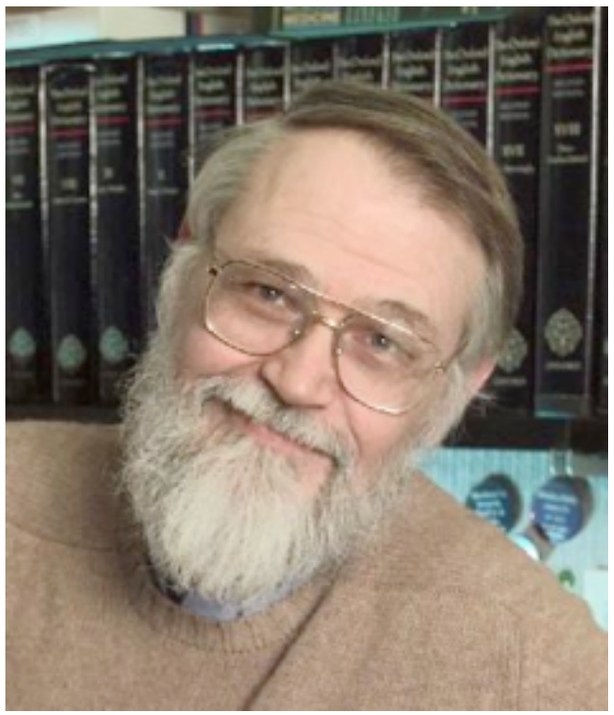
- Pro: BDD/user stories - common language for all stakeholders, including nontechnical
  - 3x5 cards
  - LoFi UI sketches and storyboards
- Pro: Write tests before coding
  - Validation by testing vs. debugging
- Con: Difficult to have continuous contact with customer?
- Con: Leads to bad software architecture?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

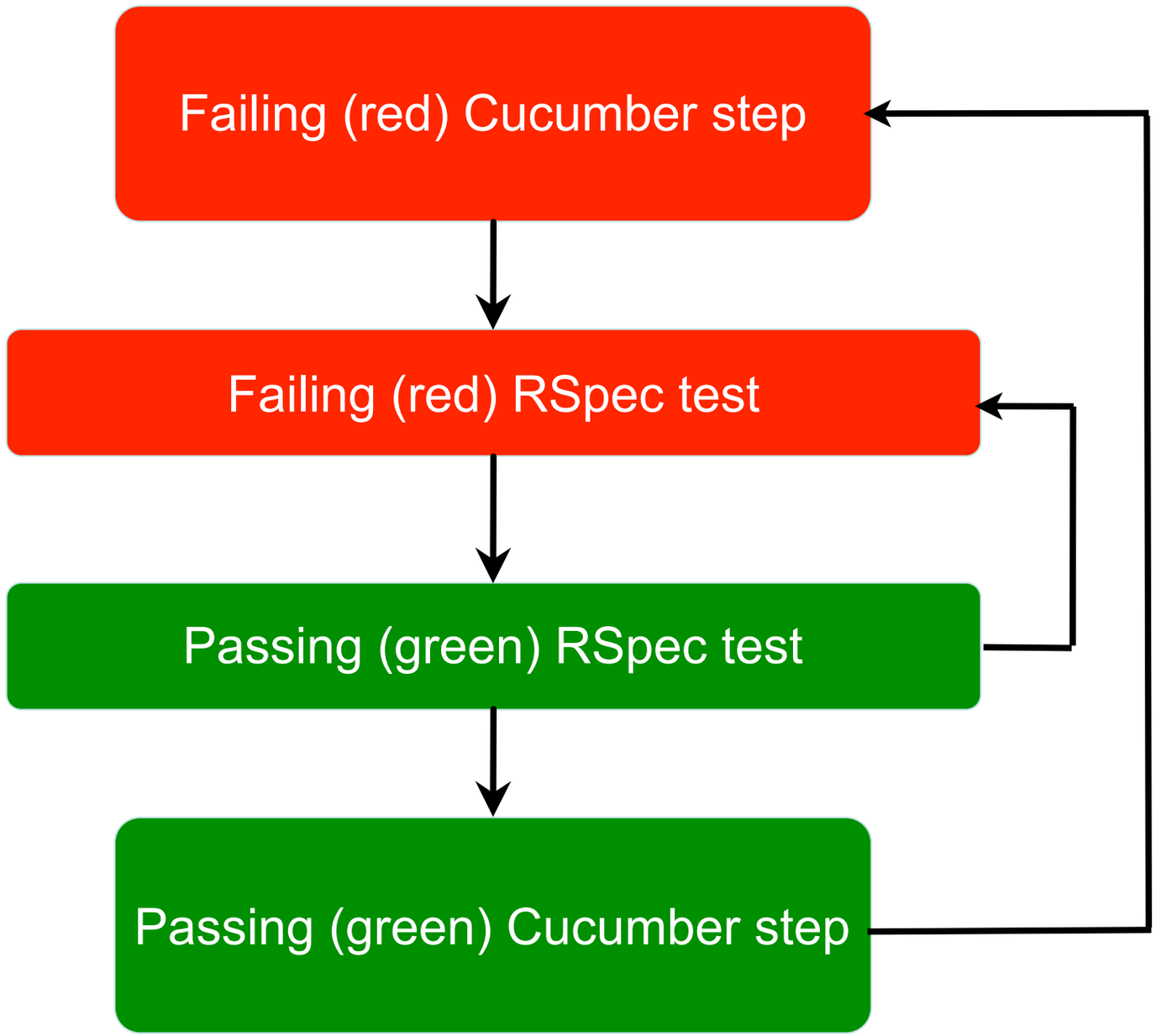
*Software Quality is the result of a good **process**, rather than the responsibility of one specific group*

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!*)

RSpec, a Domain-Specific Language for testing

- DSL: small programming language that simplifies one task at expense of generality
  - examples so far: migrations, regexes, SQL
- RSpec tests are called *specs*, and inhabit spec directory

app/models/*.rb	spec/models/*_spec.rb
app/controllers/ *_controller.rb	spec/controllers/ *_controller_spec.rb
app/views/*/*.html.haml	(use Cucumber!)

Example: calling TMDb

➤New RottenPotatoes feature: add movie using info from TMDb (vs. typing in)

➤How should user story steps behave?

When I fill in "Search Terms" with "Inception"

And I press "Search TMDb"

Then I should be on the RottenPotatoes homepage

...

Recall Rails Cookery #2:
adding new feature ==
new route+new controller method+(new view?)

The Code You Wish You Had

What should the *controller method* do that receives the search form?

1. it should call a method that will search TMDb for specified movie
2. if match found: it should select (new) “Search Results” view to display match
3. If no match found: it should redirect to RP home page with message

http://pastebin.com/kJxjwSF6

EXAMPLE (SEE PASTEBIN LINK ON PRIOR SLIDE)

```
require 'spec_helper'

describe MoviesController do
  describe 'searching TMDb' do
    it 'should call the model method that performs TMDb
search'
    it 'should select the Search Results template for
rendering'
    it 'should make the TMDb search results available to that
template'
  end
end
```

The TDD Cycle:

Red—Green—Refactor

Example: calling TMDb

➤New RottenPotatoes feature: add movie using info from TMDb (vs. typing in)

➤How should user story steps behave?

When I fill in "Search Terms" with "Inception"

And I press "Search TMDb"

Then I should be on the RottenPotatoes homepage

...

Recall Rails Cookery #2:
adding new feature ==
new route+new controller method+(new view?)

The Code You Wish You Had

What should the *controller method* do that receives the search form?

1. it should call a method that will search TMDb for specified movie
2. if match found: it should select (new) “Search Results” view to display match
3. If no match found: it should redirect to RP home page with message

http://pastebin.com/kJxjwSF6

EXAMPLE (SEE PASTEBIN)

```
require 'spec_helper'

describe MoviesController do
  describe 'searching TMDb' do
    it 'should call the model method that performs TMDb
search'
    it 'should select the Search Results template for
rendering'
    it 'should make the TMDb search results available to that
template'
  end
end
```

Test-First development

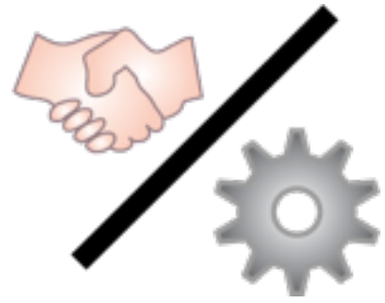
- Think about one thing the code *should* do
- Capture that thought in a test, which fails
- Write the simplest possible code that lets the test pass
- Refactor: DRY out commonality w/other tests
- Continue with next thing code should do

Red – Green – Refactor

Aim for "always have working code"

TDD for the Controller action: Setup

➤Add a route to **`config/routes.rb`**



```
# Route that posts 'Search TMDb' form

post '/movies/search_tmdb'
```

➤Convention over configuration will map this to MoviesController#search_tmdb

➤Create an empty view:

```
touch app/views/movies/search_tmdb.html.haml
```

➤Replace fake “hardwired” method in **`movies_controller.rb`** with empty method:

```
def search_tmdb

end
```

What model method?

➤Calling TMDb is responsibility of the model... but no model method exists to do this yet!

➤No problem...we'll use a seam to test the *code we wish we had ("CWWWH")*, Movie.find_in_tmdb

➤Game plan:

- ➤Simulate POSTing search form to controller action.
- ➤Check that controller action *tries to call* Movie.find_in_tmdb with data from submitted form.
- ➤The test will fail (red), because the (empty) controller method *doesn't* call find_in_tmdb.
- ➤Fix controller action to make green.

http://pastebin.com/zKnwphQZ

MOVIES_CONTROLLER_SPEC.RB

```
require 'spec_helper'

describe MoviesController do
  describe 'searching TMDb' do
    it 'should call the model method that performs TMDb search' do
      Movie.should_receive(:find_in_tmdb).with('hardware')
      post :search_tmdb, {:search_terms => 'hardware'}
    end
  end
end
```

Seams

➤A place where you can change your app's *behavior* without editing the *code*. (Michael Feathers, *Working Effectively With Legacy Code*)

➤Useful for testing: ***isolate*** behavior of some code from that of other code it depends on.

➤should_receive uses Ruby's open classes to create a seam for isolating controller action from behavior of (possibly buggy or missing) Movie.find_in_tmdb

➤Rspec *resets* all mocks & stubs after *each example* (keep tests Independent)

How to make this spec green?

- Expectation says controller action should call Movie.find_in_tmdb
- So, let's call it!

http://pastebin.com/DxzFURiu

```
def search_tmdb
  Movie.find_in_tmdb(params[:search_terms])
end
```

The spec has *driven* the creation of the controller method to pass the test.

- But shouldn't find_in_tmdb *return* something?

Test techniques we know

```
obj.should_receive(a).with(b)
```

Optional!