Enhancing Rotten Potatoes

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 Scenarios

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

```
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

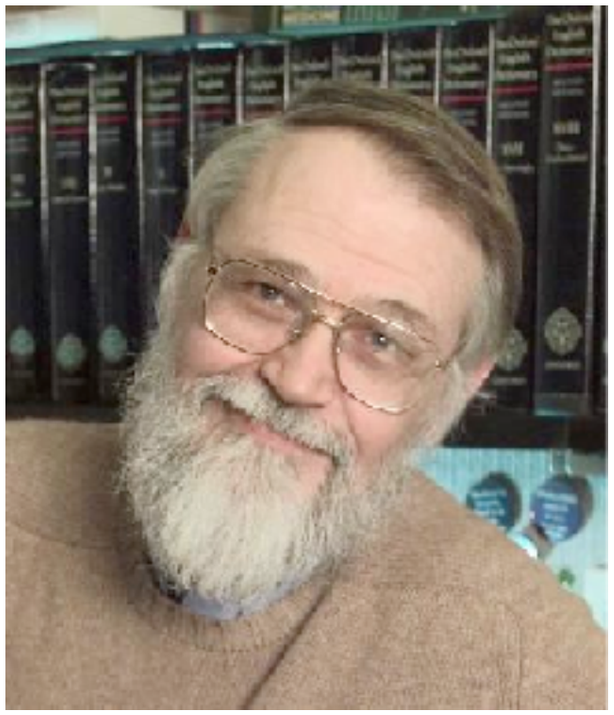
- Careless use of negative expectations
  - –Beware of overusing “Then I should not see….”
  - –Can’t tell if output is what want, only that it is not what you want
  - –Many, many outputs are incorrect
  - –Include positives to check results “Then I should see …”

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

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

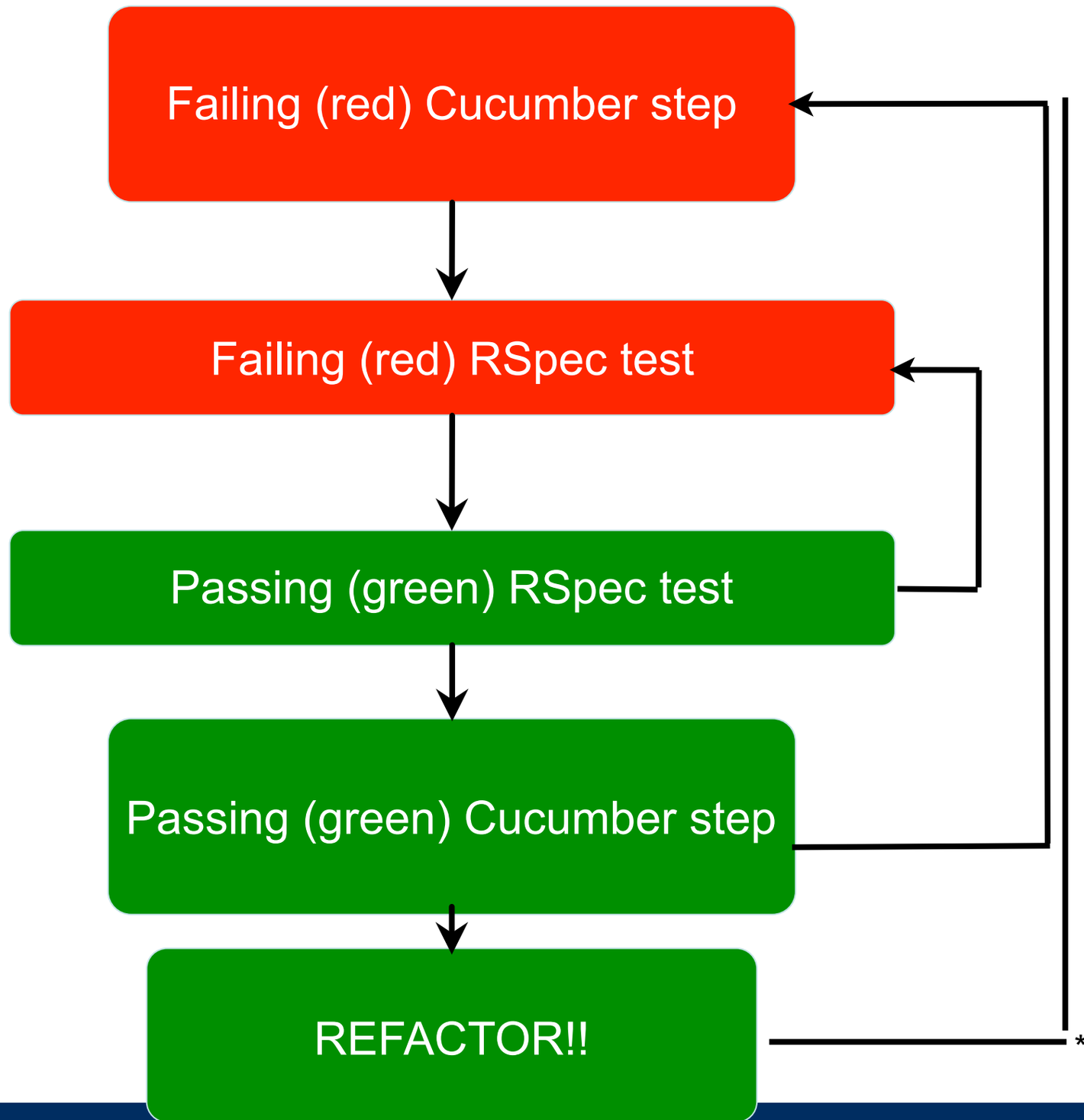
*Software Quality is the result of a good **process**, rather than the responsibility of one specific group*

BDD+TDD: The Big Picture

- Behavior-driven design (BDD)
  - develop user stories to describe features
  - via Cucumber, user stories become *acceptance tests* and *integration tests*
- Test-driven development (TDD)
  - *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 (*behavior driven* design)
- RSpec tests individual modules that contribute to those behaviors (*test driven* development)



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

- **F**ast: run (subset of) tests quickly (since you'll be running them *all the time*)
- **I**ndependent: 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

rails generate rspec:install creates structure

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) movies_controller_spec.rb

```
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) movies_controller_spec.rb

```
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

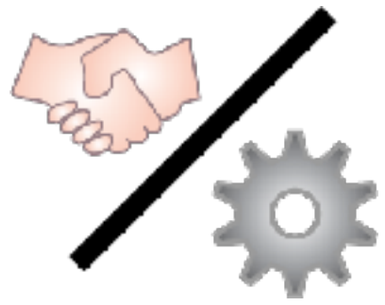
- Think about one thing the code *should* do
- Capture that thought in a test, which fails
- Write the simplest possible code that lets the test pass
- Refactor: DRY out commonality w/other tests
- Continue with next thing code should do

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 — UPDATE: expect(obj).to receive
- 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!

REPLACED by EXPECT!!

expect(obj).to receive(a).with(b)

For example:
foo.should eq(bar)
foo.should_not eq(bar)
-> expect(foo).to eq(bar)
-> expect(foo).not_to eq(bar)

http://rspec.info/blog/2012/06/rspecs-new-expectation-syntax/
https://www.relishapp.com/rspec/rspec-expectations/docs/built-in-matchers

More Controller Specs and Refactoring

Where we are & where we're going: "outside in" development

- Focus: write *expectations* that drive development of controller method
  - Discovered: must *collaborate* w/model method
  - Use outside-in recursively: *stub* model method in this test, write it later
- Key idea: *break dependency* between method under test & its collaborators
- Key concept: *seam*—where you can affect app behavior without editing code

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

“it should select Search Results view to display match”

- Really 2 specs:
  - It should decide to render Search Results
    - more important when different views could be rendered depending on outcome
  - It should make list of matches available to that view
- New *expectation* construct:
  - obj.should *match-condition* (or expect!)
  - Many built-in matchers, or define your own

Should & Should-not

- Matcher applies test to receiver of *should*

count.should == 5 (expect(count).to eq (5))	Syntactic sugar for count.should.==(5)
5.should(be.<(7)) (expect(5).to < (7))	be creates a lambda that tests the predicate expression
5.should be < 7	Syntactic sugar allowed
5.should be_odd? (expect(5).to be_odd?)	Use method_missing to call odd? on 5
result.should include(elt) (expect(result).to include(elt))	calls Enumerable#include?
result.should match(/regex/) (expect(result).to match(/regex/))	
should_not/not to also available (example: expect("a string").not to include(“foo"))	

result.should render_template('search_tmdb')

https://www.relishapp.com/rspec/rspec-expectations/docs/built-in-matchers — GREAT REF!

Checking for rendering

- After post :search_tmdb, response() method returns controller's *response object*
  - render_template matcher can check what view the controller tried to render

- Note that this view has to exist!
  - post :search_tmdb will try to do the whole MVC flow, including rendering the view
  - hence, controller specs can be viewed as *functional testing*

http://pastebin.com/C2x13z8M

An example

```
1.  require 'spec_helper'
2.
3.  describe MoviesController do
4.    describe 'searching TMDb' do
5.      it 'should call the model method that performs TMDb search' do
6.        Movie.should_receive(:find_in_tmdb).with('hardware')
7.        post :search_tmdb, {:search_terms => 'hardware'}
8.      end
9.      it 'should select the Search Results template for rendering' do
10.       Movie.stub(:find_in_tmdb)
11.       post :search_tmdb, {:search_terms => 'hardware'}
12.       response.should render_template('search_tmdb')
13.     end
14.   end
15. end
```

Updated Example (note: double instead of stub)

```
require 'spec_helper'

describe MoviesController do
  describe 'searching TMDb' do
    it 'should call the model method that performs TMDb search' do
      expect(Movie).to receive(:find_in_tmdb).with(‘hardware’)
      post :search_tmdb, {:search_terms => 'hardware'}
    end
    it 'should select the Search Results template for rendering' do
      Movie.double(:find_in_tmdb)
      post :search_tmdb, {:search_terms => 'hardware'}
      expect(response).to render_template('search_tmdb')
    end
  end
end
```

Test techniques we know

obj.should_receive(a).with(b) -> (expect(obj).to receive(a).with(b))

obj.should *match-condition*

- expect(obj).should match-condition

Rails-specific extensions to RSpec:

response()
render_template()

More Controller Specs and Refactoring, continued

It should make search results available to template

- Another rspec-rails addition: assigns()
  - pass symbol that names controller instance variable
  - returns value that controller assigned to variable

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

- Ugh! Our current code *doesn't set any instance variables:*

http://pastebin.com/DxzFURiu

- TCWWWH: list of matches in @movies

http://pastebin.com/4W08wL0X

Two new seam concepts

- stub
  - similar to should_receive, but not expectation
  - and_return optionally controls return value
- mock/double: create dumb "stunt double" object
  - stub individual methods on it:

```
m = double('movie1')  m.stub(:title).and_return('Rambo')
```

  - shortcut:

```
m=mock('movie1',:title=>'Rambo')
```

each seam enables just enough functionality for some *specific* behavior under test

Nice cheat sheet: http://www.relishapp.com/rspec/rspec-mocks/v/2-3/docs/method-stubs

Test Cookery #1

- Each spec should test *just one behavior*
- Use seams as needed to isolate that behavior
- Determine which expectation you'll use to check the behavior
- Write the test and make sure it fails for the right reason
- Add code until test is green
- Look for opportunities to refactor/beautify

Test techniques we know

```
obj.should_receive(a).with(b).and_return(c)
expect(obj).to receive (a).with(b).and_return(c)
obj.stub(a).and_return(b)
```

Optional!

d = mock('impostor') (or double!)

obj.should *match-condition (expect!)*

Rails-specific extensions to RSpec:

```
assigns(:instance_var)
response()
render_template()
```

Test techniques we know

```
obj.should_receive(a).with(b).and_return(c)
obj.stub(a).and_return(b)


d = mock('impostor')
```

Optional!

obj.should *match-condition*

Rails-specific extensions to RSpec:

assigns(:instance_var)
response()
render_template()

When you need the real thing

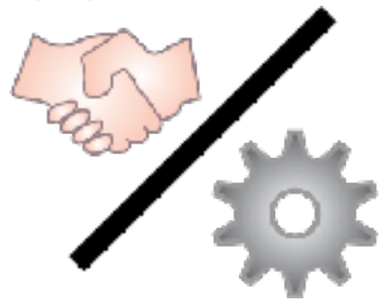
Where to get a real object: http://pastebin.com/N3s1A193

```
1. fake_movie = double('Movie')
2. fake_movie.stub(:title).and_return('Casablanca')
3. fake_movie.stub(:rating).and_return('PG')
4. fake_movie.name_with_rating.should == 'Casablanca (PG)'
```

- Fixture: statically preload some known data into database tables
- Factory: create only what you need per-test

Fixtures

- database wiped & reloaded before *each spec*
  - add fixtures :movies at beginning of describe
  - **spec/fixtures/movies.yml** are Movies and will be added to **movies** table
- Pros/uses
  - truly static data, e.g. configuration info that never changes
  - easy to see all test data in one place
- Cons/reasons not to use
  - Introduces dependency on fixture data



Factories

- Set up “helpers” to quickly create objects with default attributes, as needed per-test
- Example: FactoryGirl gem
  - or just add your own code in **spec/support/**

```
1. # in spec/factories/movie.rb
2. FactoryGirl.define do
3.   factory :movie do
4.     title 'A Fake Title' # default values
5.     rating 'PG'
6.     release_date { 10.years.ago }
7.   end
8. end
9. # in your specs
10.it 'should include rating and year' do
11.  movie = FactoryGirl.build(:movie, :title => 'Milk')
12.  # etc.
13.end
```

http://pastebin.com/bzvKG0VB

Factories

- Set up “helpers” to quickly create objects with default attributes, as needed per-test
- Example: FactoryGirl gem
  - or just add your own code in **spec/support/**

http://pastebin.com/bzvKG0VB

- Pros/uses:
  - Keep tests Independent: unaffected by presence of objects they don’t care about
- Cons/reasons not to use:
  - Complex relationships may be hard to set up (but may indicate too-tight coupling in code!)

Pitfall: *mock trainwreck*

- Goal: test searching for movie by its director or by awards it received

```
a = mock('Award', :type => 'Oscar')
d = mock('Director',
   :name => 'Darren Aronovsky'
m = mock('Movie', :award => a,
   :director => d)
...etc...
m.award.type.should == 'Oscar'
m.director.name.split(/ +/).last.should
  == 'Aronovsky'
```

TDD for the Model
& Stubbing the Internet

Explicit vs. Implicit Requirements

- find_in_tmdb should call TmdbRuby gem with title keywords
  - If we had no gem: It should directly submit a RESTful URI to remote TMDb site
- What if TmdbRuby gem signals error?
  - API key is invalid
  - API key is not provided
- Use *context & describe* to divide up tests according to cases

Example: Describe and Context

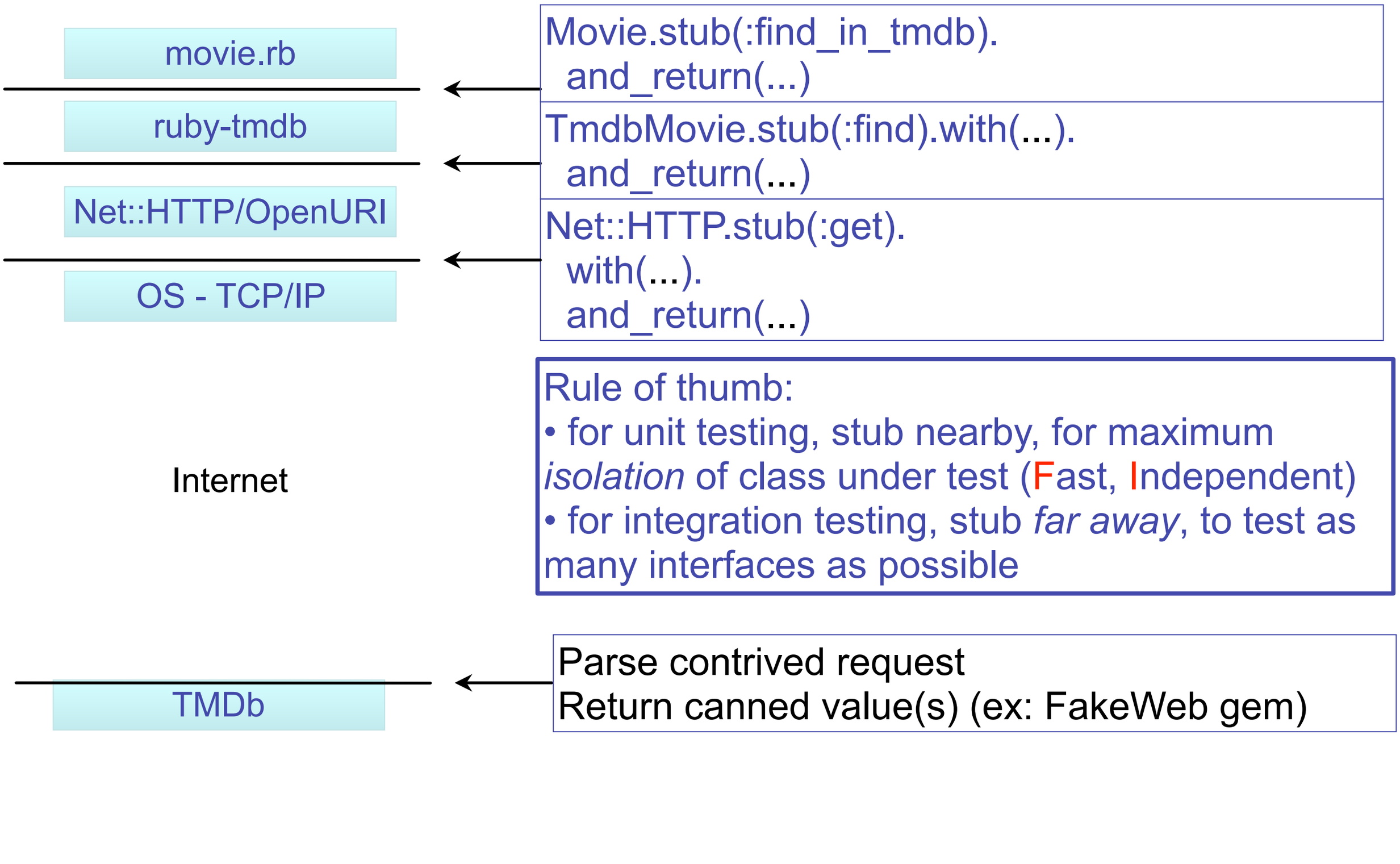
```
1. require 'spec_helper'
2.
3. describe Movie do
4.   describe 'searching Tmdb by keyword' do
5.     context 'with valid key' do
6.       it 'should call Tmdb with title keywords' do
7.        TmdbMovie.should_receive(:find).
8.          with(hash_including :title => 'Inception')
9.        Movie.find_in_tmdb('Inception')
10.      end
11.    end
12.    context 'with invalid key' do
13.      it 'should raise InvalidKeyError if key not given' do
14.        Movie.stub(:api_key).and_return('')
15.        lambda { Movie.find_in_tmdb('Inception') }.
16.          should raise_error(Movie::InvalidKeyError)
17.      end
18.      it 'should raise InvalidKeyError if key is bad' do
19.        TmdbMovie.stub(:find).
20.          and_raise(RuntimeError.new('API returned code 404'))
21.        lambda { Movie.find_in_tmdb('Inception') }.
22.          should raise_error(Movie::InvalidKeyError)
23.      end
24.    end
25.  end
26.end
```

http://pastebin.com/ELQfC8je

Review

- Implicit requirements derived from explicit
  - by reading docs/specs
  - as byproduct of designing classes
- We used 2 different stubbing approaches
  - case 1: we *know* TMDb will *immediately* throw error; want to test that we catch & convert it
  - case 2: need to *prevent* underlying library from contacting TMDb at all
- context & describe group similar tests
  - in book: using before(:each) to setup common preconditions that apply to whole group of tests

Where to stub in Service Oriented Architecture?



Test techniques we know

```
obj.should_receive(a).with(b).and_return(c)
                .with(hash_including 'k'=>'v')
obj.stub(a).and_raise(SomeClass::SomeError)


d = mock('impostor')
```

```
obj.should raise_error(SomeClass::SomeError)
describe, context
```

Rails-specific extensions to RSpec:

```
assigns(:instance_var)
response()
render_template()
```

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?