More Controller Specs and Refactoring, continued

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!)

Test techniques we know

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

Optional!

d = mock(‘impostor')
book = double("book")

obj.should *match-condition*

Rails-specific extensions to RSpec:
assigns(:instance_var)
response()
render_template()

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

When you need the real thing

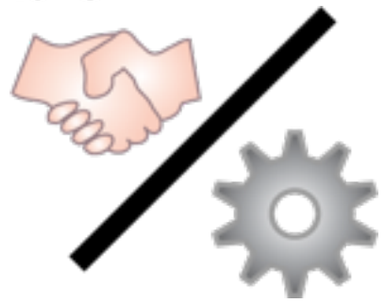
Where to get a real object: http://pastebin.com/N3s1A193

```
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/**

```
# in spec/factories/movie.rb
FactoryGirl.define do
  factory :movie do
    title 'A Fake Title' # default values
    rating 'PG'
    release_date { 10.years.ago }
  end
end
# in your specs
it 'should include rating and year' do
  movie = FactoryGirl.build(:movie, :title => 'Milk')
  # etc.
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

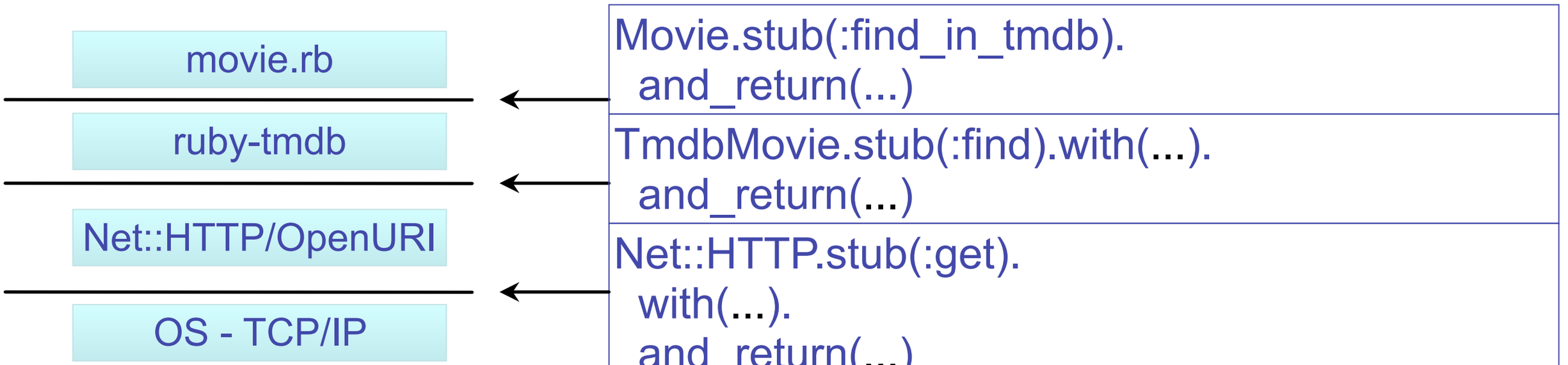
```
1. require 'spec_helper'
2.
3. describe Movie do
4.   describe 'searching Tmdb by keyword' do
5.     context 'with valid key' do
6.       it 'should call Tmdb with title keywords' do
7.         TmdbMovie.should_receive(:find).
8.           with(hash_including :title => 'Inception')
9.         Movie.find_in_tmdb('Inception')
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) (see double!)
d = mock('impostor') (see double!)
obj.should raise_error(SomeClass::SomeError)
describe, context
```

Rails-specific extensions to RSpec:

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

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