Testing

Unit, Integration, Acceptance, BDD & TDD

Testing
A Means To An End

“Our highest priority is to satisfy the customer through early and continuous delivery of valuable software”

http://agilemanifesto.org/principles.html

Testing Today

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

Testing Today

- Before
  - developers finish code, some ad-hoc testing

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

-
  - testing tools & processes highly automated;
  - QA/testing group improves *testability* & *tools*

Testing
A Means To An End

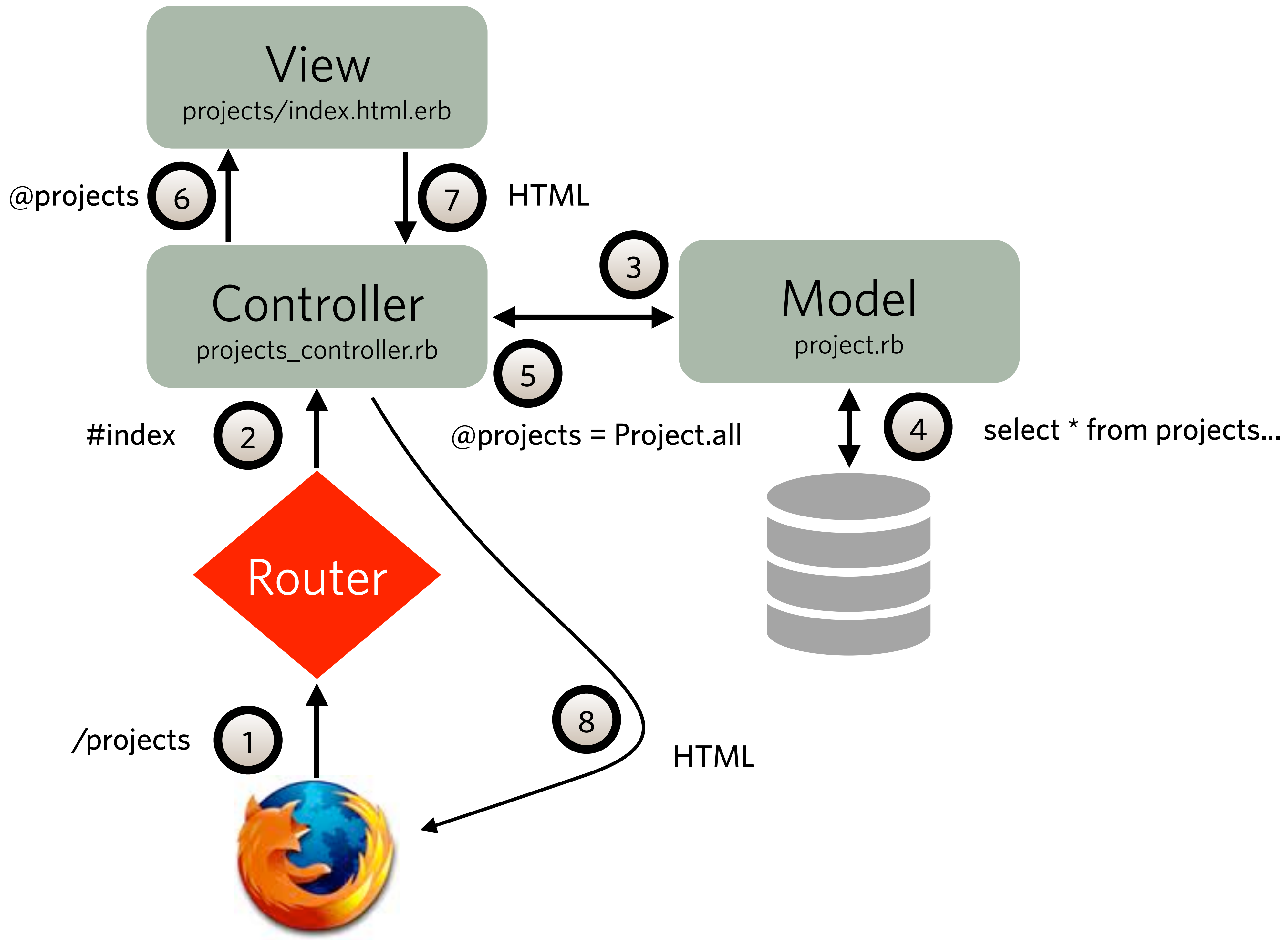
- BDD testing frameworks are DSLs (built on top of Unit Testing Frameworks) to "get the words rights"
- Most examples still use Units (class & methods) to teach BDD. Therefore developers still start at the inside.
- Rails showed early own that Web Application Testing CAN be automated
- Integration testing still hard to define for most developers
- Acceptance testing is NOT integration testing (unless you mean integrating with your users)

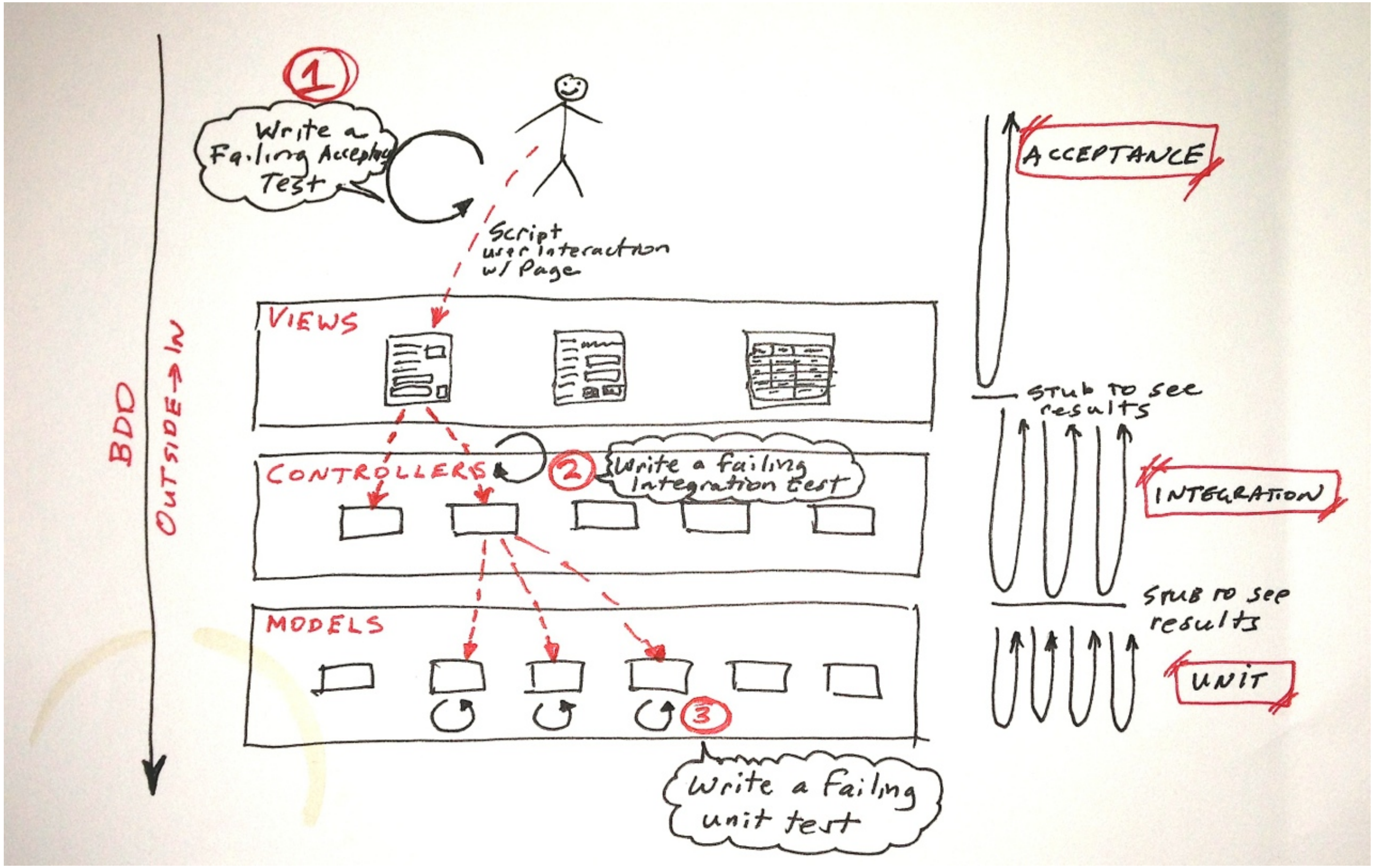
http://c2.com/cgi/wiki?TenYearsOfTestDrivenDevelopment

Request Handling

The Request-Response Pipeline

1. User requests /projects
2. Rails router forwards the request to projects_controller#index action
3. The index action creates the instance variable @projects by using the Project model all method
4. The all method is mapped by ActiveRecord to a select statement for your DB
5. @projects returns back with a collection of all Project objects
6. The index action renders the index.html.erb view
7. An HTML table of Projects is rendered using ERB (embedded Ruby) which has access to the @projects variable
8. The HTML response is returned to the User




1
Write a Failing Acceptance Test
Script user interaction w/ Page
BDD
OUTSIDE → IN
VIEWS
CONTROLLERS
2
Write a failing integration test
MODELS
3
Write a Failing unit test
ACCEPTANCE
Stub to see results
INTEGRATION
Stub to see results
UNIT

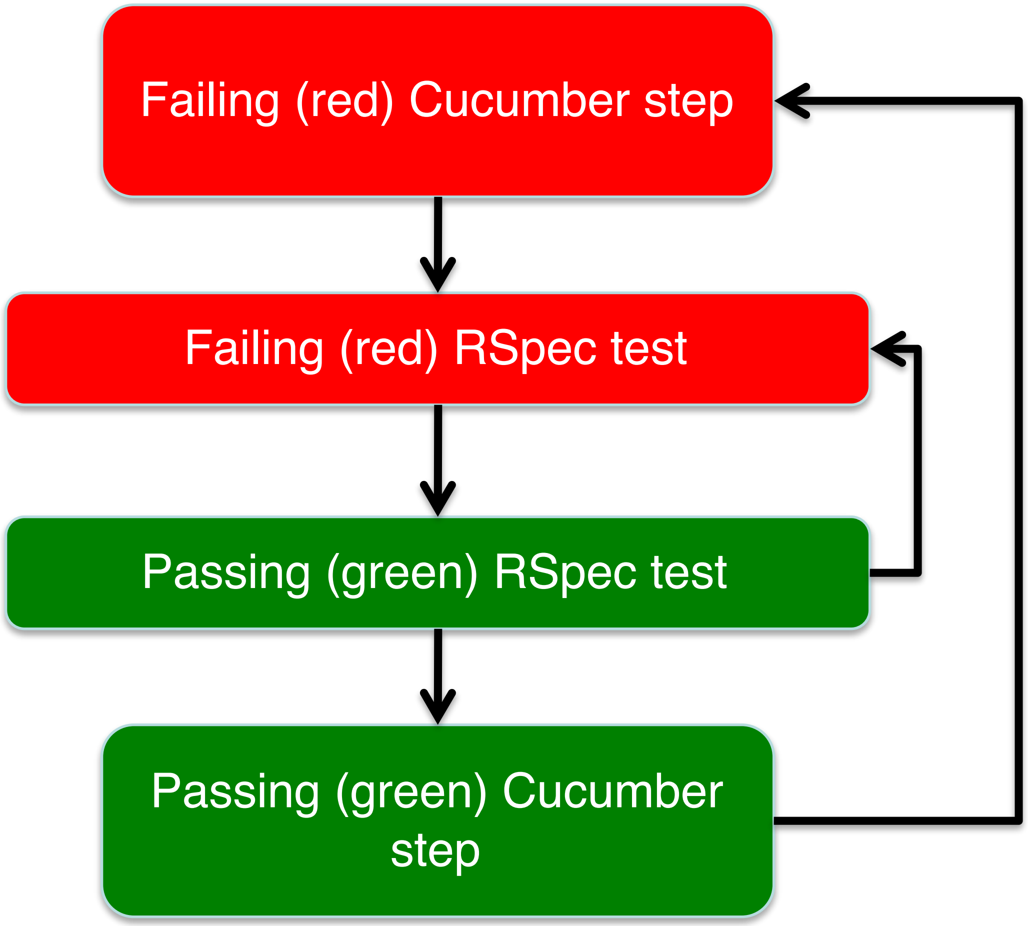
Outside-in Testing, BDD/TDD, Unit, Integration & Acceptance in one picture

BDD+TDD: The Big Picture

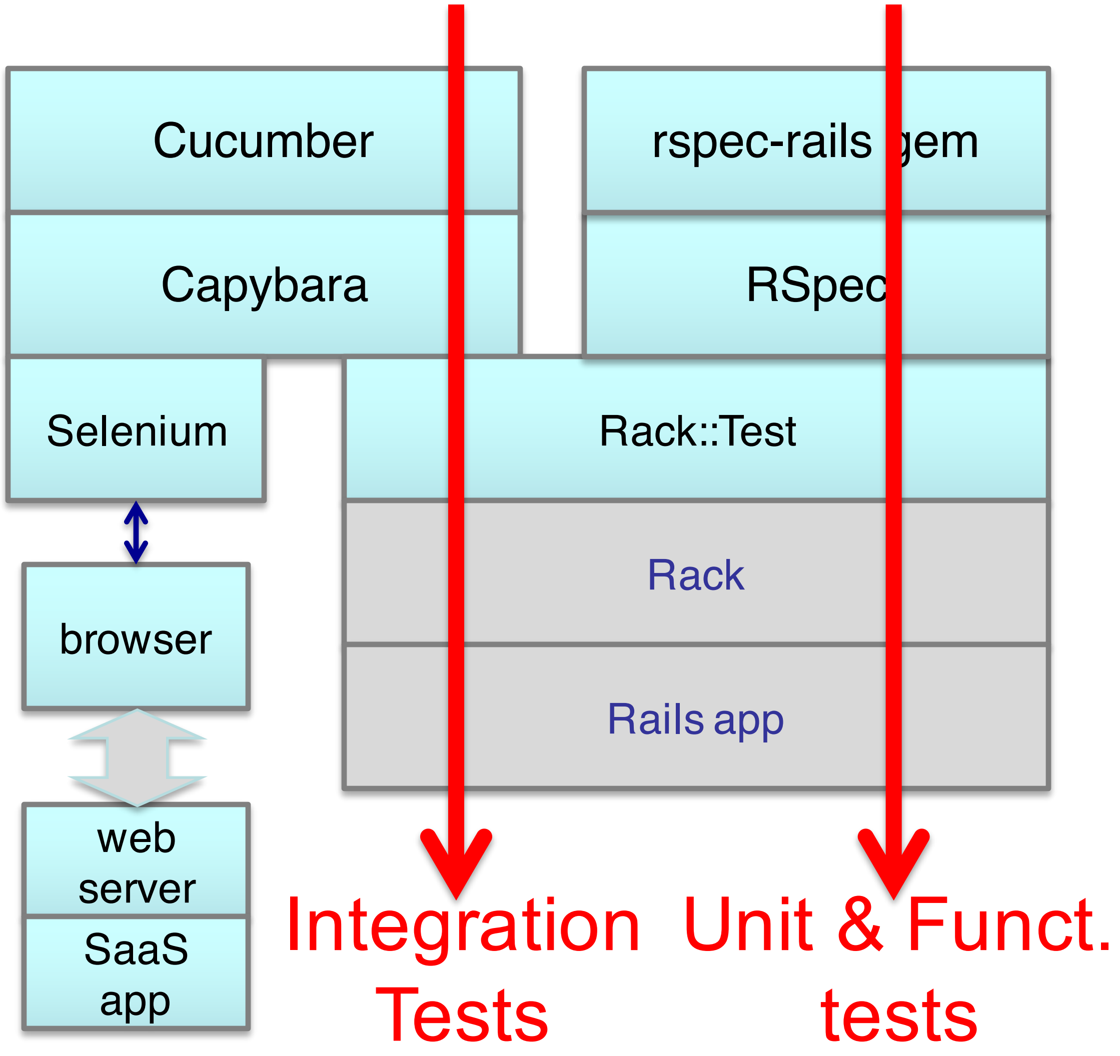
- Behavior-driven design (BDD)
  - develop user stories *(the features you wish you had)* to describe how app will work
  - via Cucumber, user stories become *acceptance tests* and *integration tests*
- Test-driven development (TDD)
  - *step definitions* for new story,may require new code to be written
  - 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*

Cucumber & RSpec

- Cucumber describes *behavior* via features & scenarios (*behavior driven* design)
- RSpec tests individual modules that contribute to those behaviors (*test driven* development)


Failing (red) Cucumber step
Failing (red) RSpec test
Passing (green) RSpec test
Passing (green) Cucumber step

Testing stacks revisited



BDD

Behavior-Driven Development

BDD
Behavior Driven Development

- BDD focuses TDD to deliver the maximum value possible to stakeholders
- BDD is a refinement in the language and tooling used for TDD
- As the name implies with BDD we focus on behavior specifications
- Typically BDD works from the outside in, that is starting with the parts of the software whose behavior is directly perceive by the user
- We say BDD refines TDD in that there is an implicit decoupling of the tests and the implementation (i.e.. don't tests implementation specifics, test perceived behavior)

BDD
Behavior Driven Development

- BDD focuses on “specifications” that describe the behavior of the system
- In the process of fleshing out a story the specifications start from the outside and might move towards the inside based on need
- In the context of a Web Application this Outside-In approach typically means that we are starting with specifications related to the User Interface
- If we are talking about a software component then we mean the API for said component

BDD
Behavior Driven Development

- **BDD** helps us figure out **what to test**, where to start and **what to ignore** (or what to make a target of opportunity)
  - **What** to test? ➜ Use Cases or User Stories, test what something **does (behavior)** rather than what something **is (structure)**
  - **Where** to start? ➜ From the outer most layer
  - **What** to ignore? ➜ Anything else… Until proven that you can’t

BDD
Behavior Driven Development

- BDD focuses on **getting the words right**, the resulting specifications become an **executable/self-verifying** form of **documentation**

- BDD specifications follow a format that makes them easy to be driven by your system's User Stories

*As a [**role**], I want [**goal**] so that [**benefit**]*

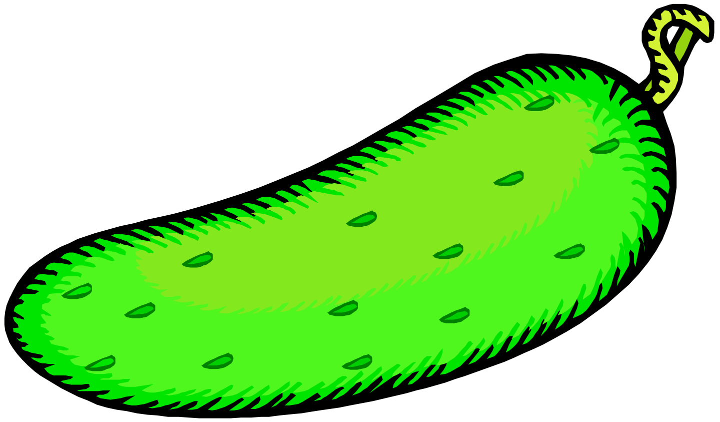
User Story

*Given [**role and its state**]*
*When [**an event/action occurs**]*
*Then [**the benefit**]*

Specification

Introducing Cucumber & Capybara

(Engineering Software as a Service § 7.5)



David Patterson

User stories => Acceptance Tests?

- Wouldn't it be great to automatically map 3x5 card user stories into tests for user to decide if accept the app?
- How would you match the English text to test code?
- How could you run the tests without a human in the loop to perform the actions?

Cucumber: 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 not code, so clear to customer and can be used to reach agreement
  - Also not 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

Cucumber User Story, Feature, and Steps

- User story: typically maps to one feature
- Feature: ≥1 scenarios that show different ways a feature is used
  - Keywords `Feature` and `Scenario` identify respective components
  - both *happy path* & *sad path* scenarios

  `features/*.feature`

- Scenario: typically 3 - 8 steps
- Step definitions: Ruby code to test steps

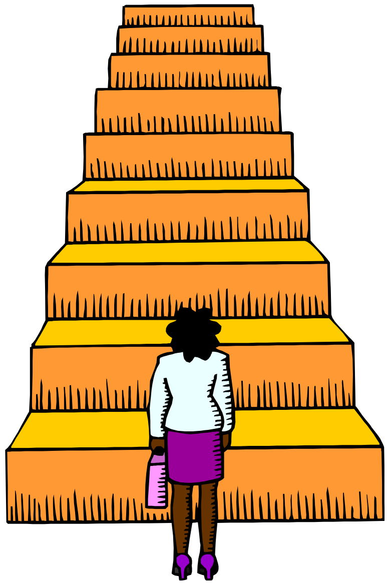
  `features/step_definitions/*_steps.rb`

5 Step Keywords

1. `Given` steps represent state of world before event: preconditions
2. `When` steps represent event
   – e.g., simulate user pushing a button
3. `Then` steps represent expected postconditions; check if true

4. / 5. `And` & `But` extend previous step

These are all aliases for same method



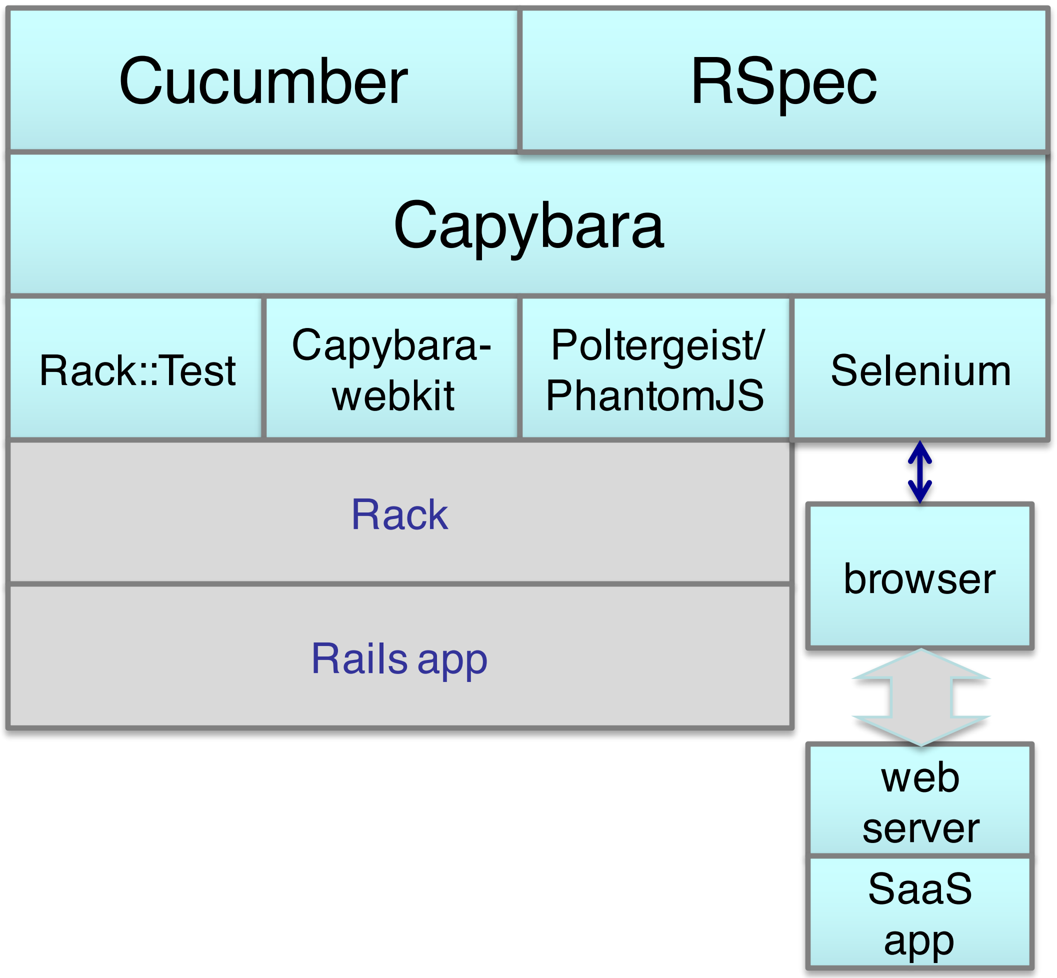
Steps => Step Definitions via Regular Expressions

- ***Regexes match English phrases in steps of scenarios to step definitions!***
- `Given /^(?:|I )am on (.+)$/`
- `“I am on the Rotten Potatoes home page”`
- Step definitions (Ruby code) likely use captured string
  - `“the Rotten Potatoes home page”`

Fake User to try Scenarios?

- Tool that pretends to be user to follow scenarios of user story
- Capybara simulates browser
  - Can interact with app to receive pages
  - Parse the HTML
  - Submit forms as a user would

Cucumber testing stacks

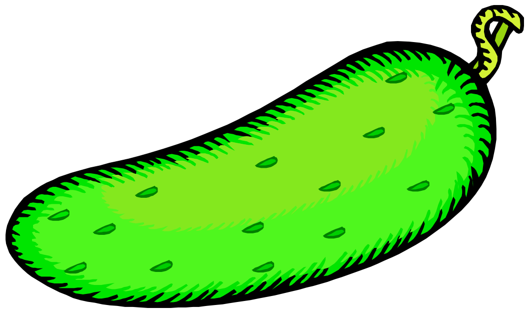


- With Selenium, can script completely external interactions
- SauceLabs.com will run your Selenium tests and send you videos of results

From Red to Green

- `cucumber` *filename* to run one feature, `rake cucumber` runs all
- Green for passing steps
- 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)



Cucumber 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/Cucumber test behavior; TDD/RSpec in folllowing chapter is how write methods to make all scenarios pass

TDD with RSpec

Mini-Tutorial

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 simpifies one task at expense of generality
  - examples so far: migrations, regexes, SQL
- RSpec tests are called *specs* or *examples*

- Run the tests in one file: **`rspec`** *filename*
  - Red failing, Green passing, Yellow *pending*
- *Much better: running* **`guard/autotest`**

Test-Driven Development
Drive your Development with Tests

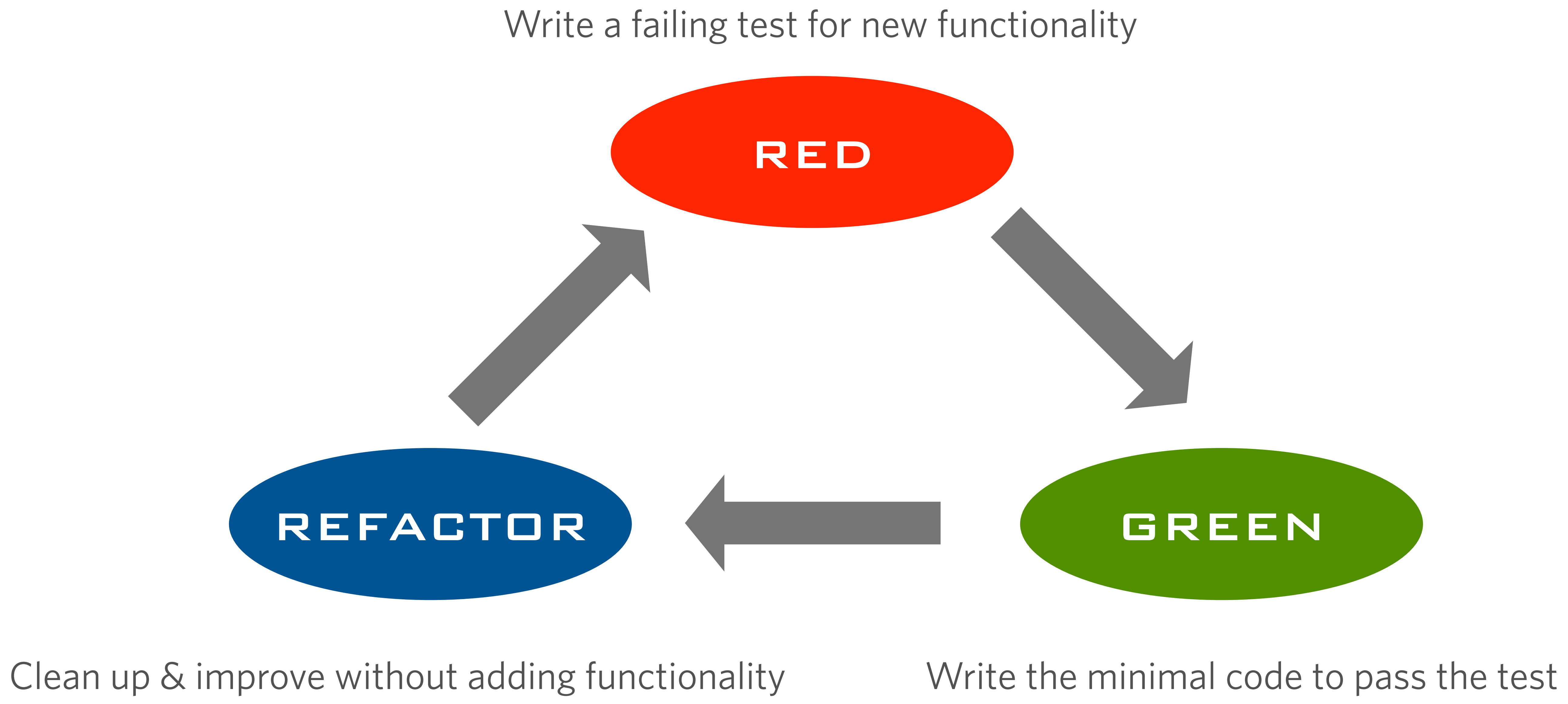
- TDD is **not** *really* **about testing**
- TDD is a **design technique**
- TDD leads to **cleaner code** with **separation** of **concerns**
- Cleaner code is more reliable and easier to maintain (Duh)

Test-Driven Development
Drive your Development with Tests

- TDD creates a **tight loop of development** that **cognitively engages us**
- TDD gives us **lightweight rigor** by making development, **goal-oriented** with a **clear goal setting**, goal reaching and improvement stages
- The stages of TDD are commonly known as the **Red-Green-Refactor** loop

Test-Driven Development
Drive your Development with Tests

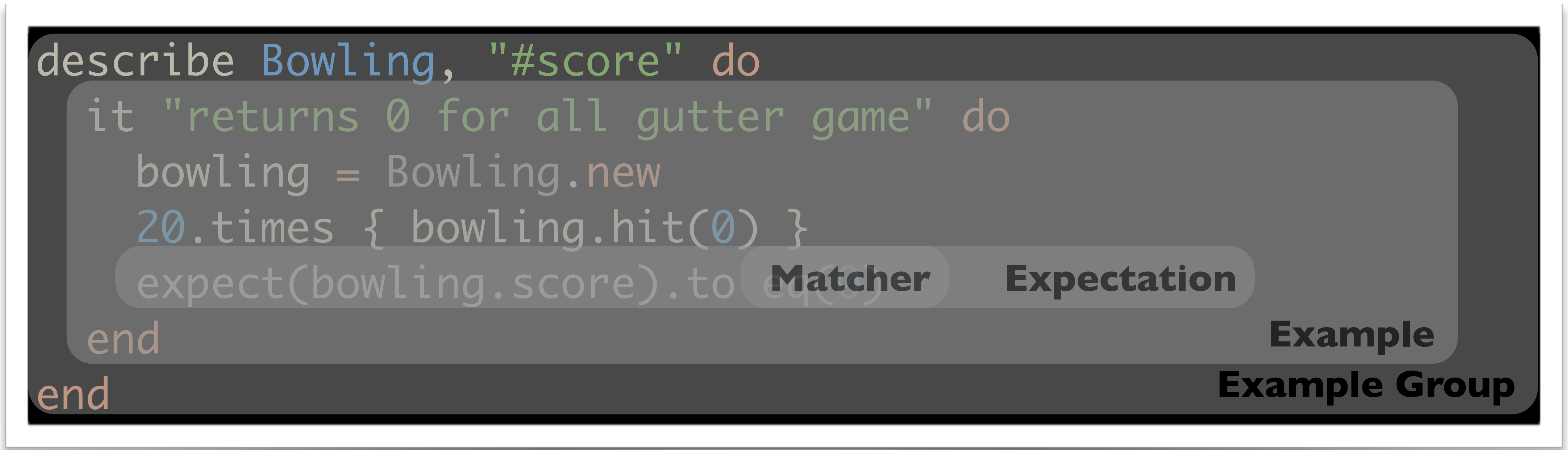
- The **Red-Green-Refactor** Loop:


Write a failing test for new functionality
RED
GREEN
REFACTOR
Clean up & improve without adding functionality
Write the minimal code to pass the test

RSpec

Ruby's BDD Framework

- RSpec uses the method **describe** to create and Example Group
- Example groups can be nested using the **describe** or **context** methods



```ruby
describe Bowling, "#score" do
  it "returns 0 for all gutter game" do
    bowling = Bowling.new
    20.times { bowling.hit(0) }
    expect(bowling.score).to eq(0)
  end
end
```

RSpec Basics by Example

```
x = Math.sqrt(9)
expect(x).to eq 3
expect(sqrt(9)).to
be_within(.5).of(3)

expect(x.odd?).to be_true
expect(x).to be_odd
expect(hash['key']).to be_truthy

m = Movie.new(:rating => 'R')
expect(m).to be_a_kind_of Movie
```

relishapp.com/rspec/rspec-expectations/docs/built-in-matchers

```
require 'ruby_intro.rb'

describe "BookInStock" do
  it "should be defined" do
    expect { BookInStock }.not_to raise_error
  end

  describe 'getters and setters' do
    before(:each)  { @book = BookInStock.new('isbn1', 33.8) }
    it 'sets ISBN' do
      expect(@book.isbn).to eq('isbn1')
    end
    it 'sets price' do
      expect(@book.price).to eq(33.8)
    end
    it 'can change ISBN' do
      @book.isbn = 'isbn2'
      expect(@book.isbn).to eq('isbn2')
    end
    it 'can change price' do
      @book.price = 300.0
      expect(@book.price).to eq(300.0)
    end
  end
```

More RSpec Basics by Example

```
expect { m.save! }.
 to raise_error(ActiveRecord::RecordInvalid)
m =  (create a valid movie)
expect(m).to be_valid
expect { m.save! }.
  to change { Movie.count }.by(1)
```

```
expect {  lambda  }.to(assertion)
expect(expression).to(assertion)
```

RSpec
Matchers

- RSpec comes built in with a nice collection of matchers, including:

```
be_true  # passes if actual is truthy (not nil or false)
be_false # passes if actual is falsy (nil or false)
be_nil   # passes if actual is nil
be       # passes if actual is truthy (not nil or false)

expect { ... }.to raise_error
expect { ... }.to raise_error(ErrorClass)
expect { ... }.to raise_error("message")
expect { ... }.to raise_error(ErrorClass, "message")

expect { ... }.to throw_symbol
expect { ... }.to throw_symbol(:symbol)
expect { ... }.to throw_symbol(:symbol, 'value')

be_xxx         # passes if actual.xxx?
have_xxx(:arg) # passes if actual.has_xxx?(:arg)
```

RSpec
Matchers

- and ...

```
be_empty

be(expected) # passes if actual.equal?(expected)
eq(expected) # passes if actual == expected

== expected     # passes if actual == expected
eql(expected)   # passes if actual.eql?(expected)
equal(expected) # passes if actual.equal?(expected)

be >  expected
be >= expected
be <= expected
be <  expected
=~ /expression/
match(/expression/)
be_within(delta).of(expected)

be_instance_of(expected)
be_kind_of(expected)
```

https://www.relishapp.com/rspec/rspec-expectations/v/2-13/docs/built-in-matchers

So what's in rspec-rails?

- Additional *methods* mixed into RSpec to test Rails-specific things
  - e.g. `get, post, put`, … for controllers
  - `response` object for controllers
- *Matchers* to test Rails apps' behaviors

```
expect(response).to
    render_template("movies/index")
```

- Support for creating various *doubles* needed to test non-toy methods

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 expect to 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?

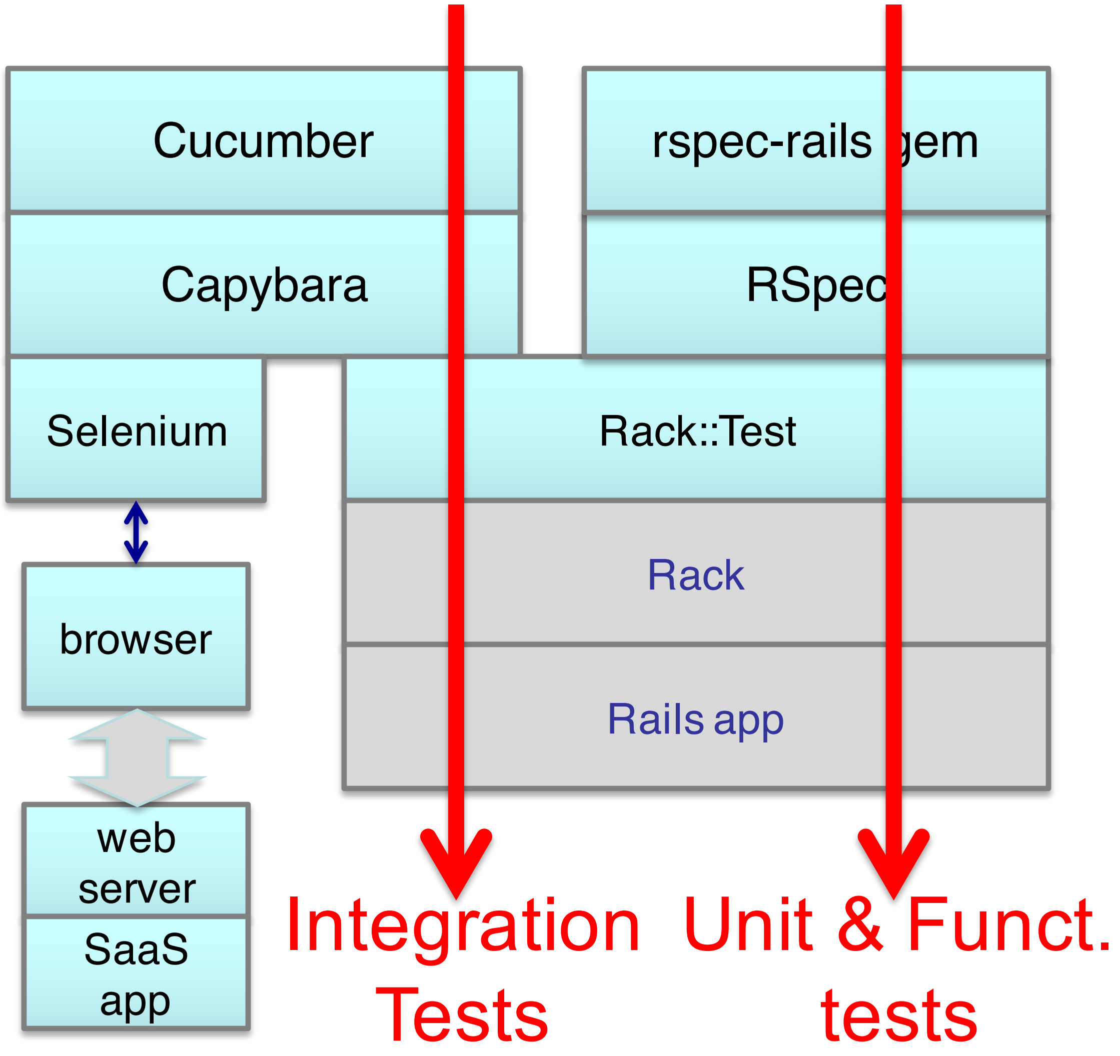
1. call a method that will search TMDb for specified movie
2. if match found: select (new) “Search Results” view to display match
3. If no match found: redirect to home page with message

http://pastebin.com/LcejGjYs

Mocks and Stubs

(Engineering Software as a Service § 8.4)

Testing stacks revisited



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—live demo
2. if match found: it should make search results available to template

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
- D'oh! our current code *doesn't set any instance variables:*

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

- TCWWWH: list of matches in `@movies`

```
it 'makes search results available to template' do
  Movie.stub(:find_in_tmdb).and_return(@fake_results)
  post :search_tmdb, {:search_terms => 'hardware'}
  expect(assigns(:movies)).to eq(@fake_results)
end
```

Two new seam concepts

- `stub`
  - similar to `to_receive`, but not expectation
  - `and_return` optionally controls return value
- `mock`: “stunt double” object, often used for behavior verification (did method get called)
  - stub individual methods on it:

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

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

RSpec Cookery #1

- Each spec should test *just one behavior*
- Use seams as needed to isolate that behavior
- Determine what type of expectation will 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

Unit vs. Functional tests in SaaS apps

- Unit tests: behavior *within a method/class*
  - collaborator classes are *mocked*
  - collaborator methods may be *stubbed out* (in this class or collaborator classes)
  - both are sometimes generically called *doubles*
- Functional test: behavior *across* methods/classes
  - e.g. controller flow from GET/POST all the way to template rendering, which rspec-rails stubs
  - (so not a true full-stack test)

Fixtures and Factories

(Engineering Software as a Service § 8.5)

Armando Fox

When you need the real thing

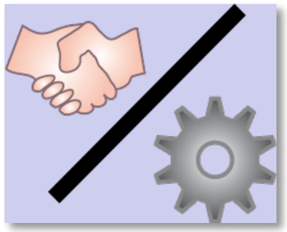
```
fake_movie = mock('Movie')
fake_movie.stub(:title).and_return('Casablanca')
fake_movie.stub(:rating).and_return('PG')
fake_movie.name_with_rating.should == 'Casablanca
   (PG)'
```

Where to get a real object:

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

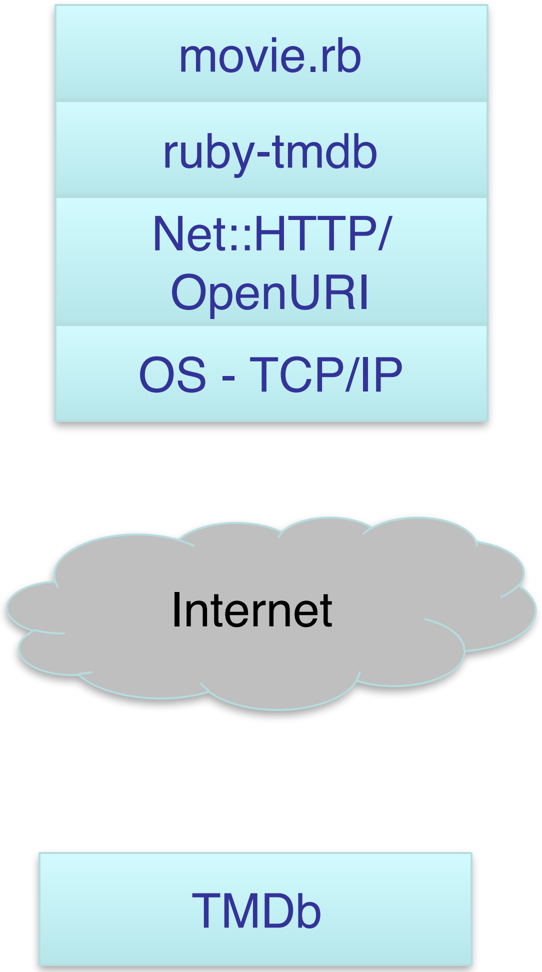
Fixtures

- 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

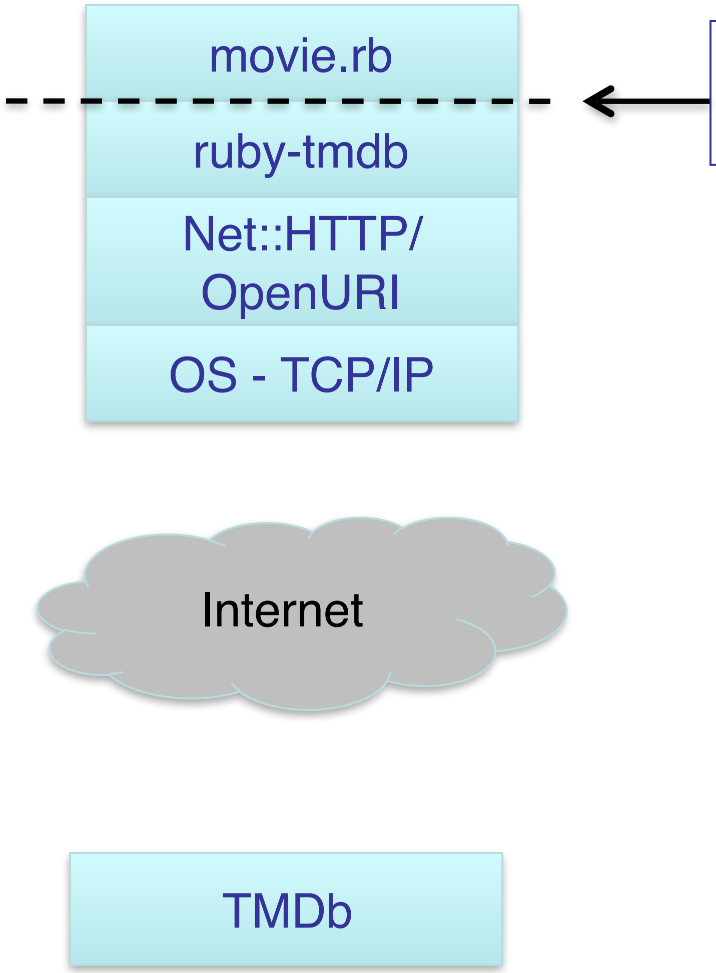


- Pros/uses
  - truly static data, e.g. configuration info that never changes
  - easy to see all test data in one place
- Cons/reasons not to use
  - may introduce dependency on fixture data

Where to stub in Service Oriented Architecture?

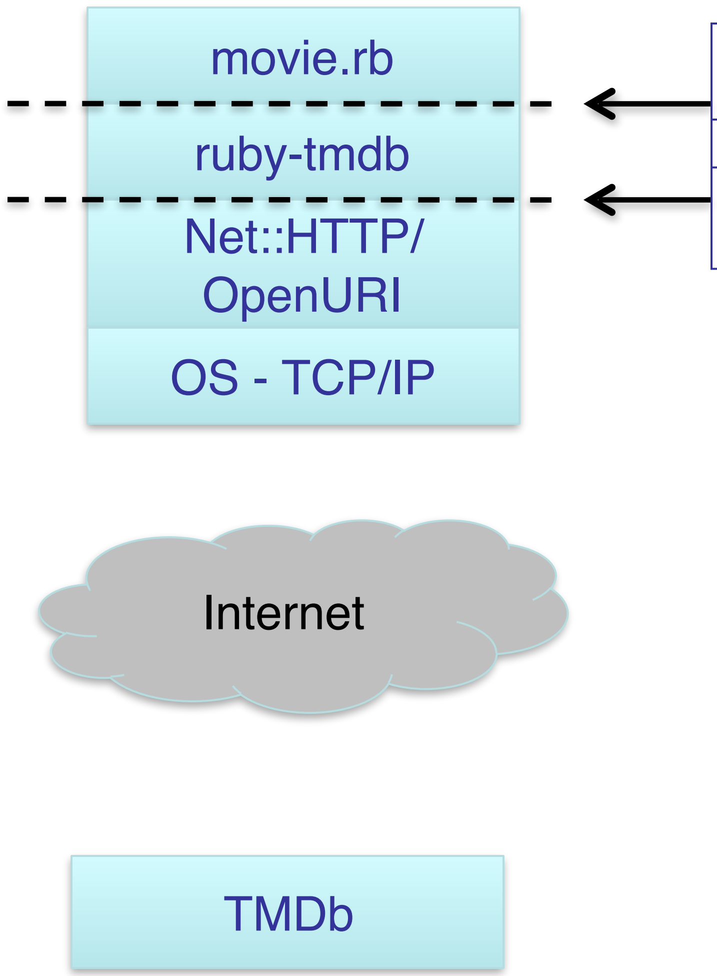


Where to stub in Service Oriented Architecture?



```
Movie.stub(:find_in_tmdb).
  and_return(...)
```

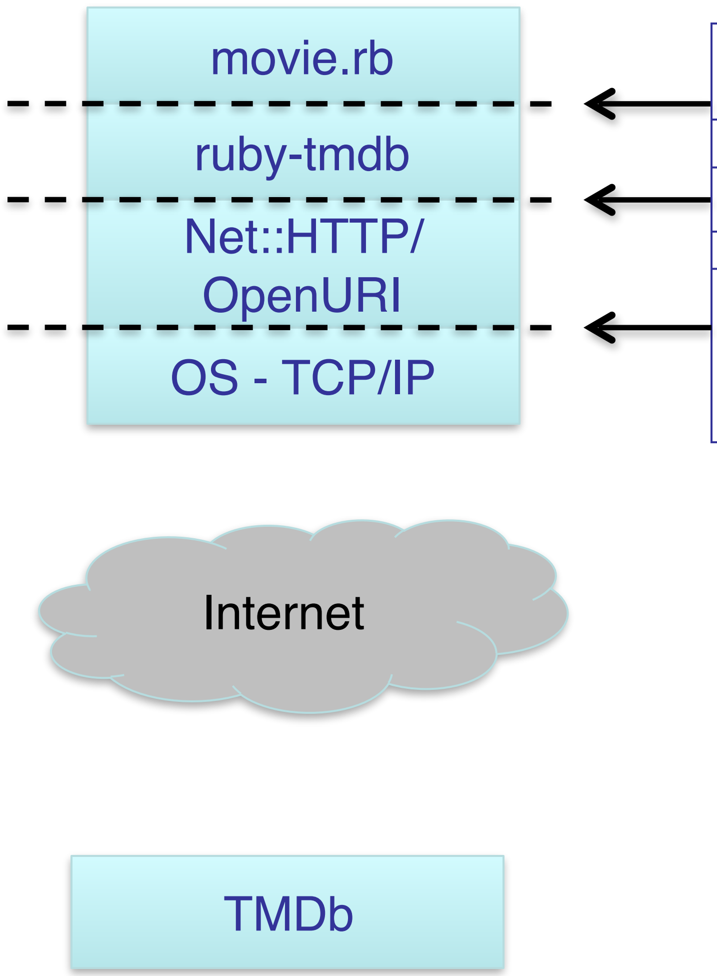
Where to stub in Service Oriented Architecture?



```
Movie.stub(:find_in_tmdb).
  and_return(...)
```

```
TmdbMovie.stub(:find).with(...).
  and_return(...)
```

Where to stub in Service Oriented Architecture?

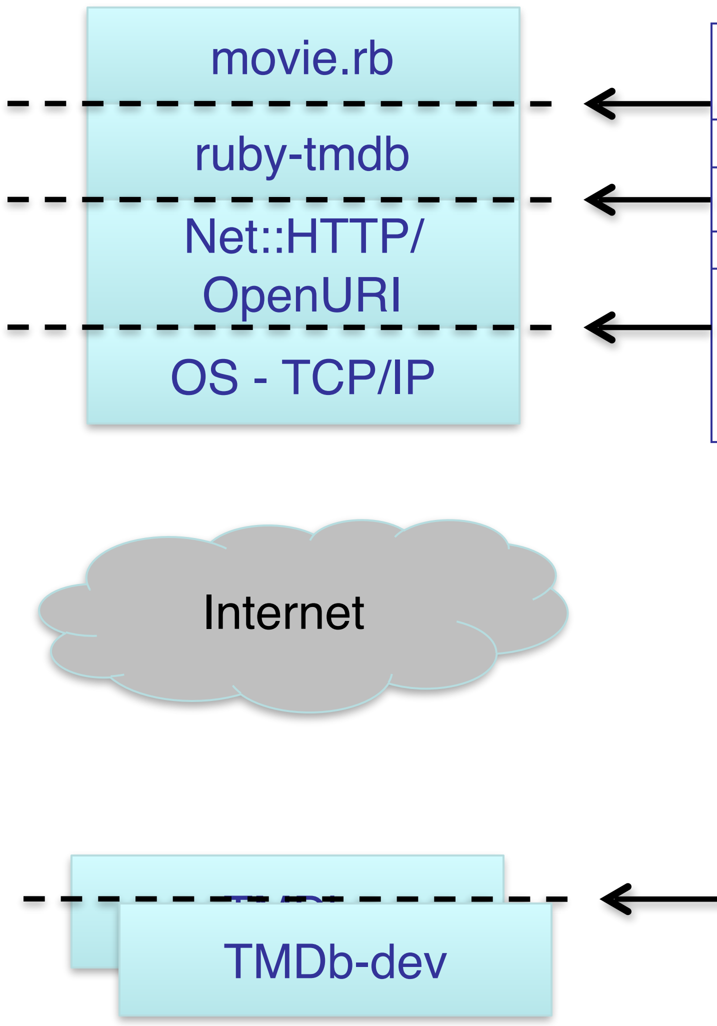


```
Movie.stub(:find_in_tmdb).
  and_return(...)
```

```
TmdbMovie.stub(:find).with(...).
  and_return(...)
```

```
Net::HTTP.stub(:get).
  with(...).
  and_return(...)
```

Where to stub in Service Oriented Architecture?



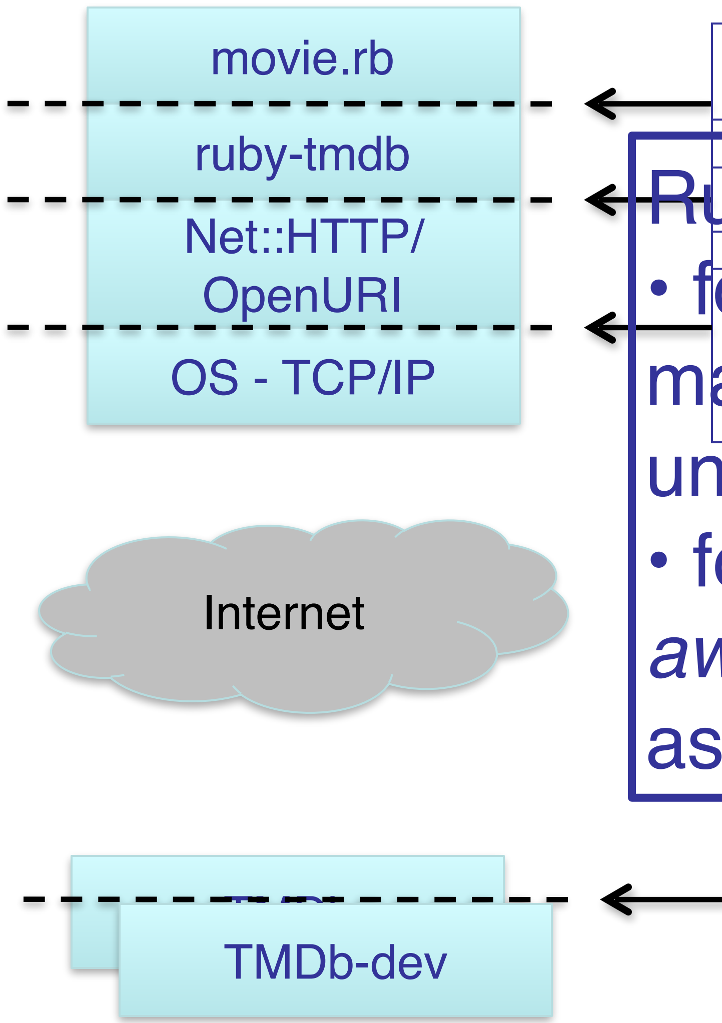
```
Movie.stub(:find_in_tmdb).
  and_return(...)
```

```
TmdbMovie.stub(:find).with(...).
  and_return(...)
```

```
Net::HTTP.stub(:get).
  with(...).
  and_return(...)
```

Parse contrived request
Return canned value(s)
(using `FakeWeb` *gem, for example)*

Where to stub in Service Oriented Architecture?



```
Movie.stub(:find_in_tmdb).
  and_return(...)
```

```
TmdbMovie.stub(:find).with(...).
  and_return(...)
```

```
Net::HTTP.stub(:get).
  with(...).
  and_return(...)
```

Rule of thumb:

• for unit testing, stub nearby, for maximum *isolation* of class under test (Fast, Independent)

• for integration testing, stub *far away*, to test as many interfaces as possible

Parse contrived request
Return canned value(s)
(using `FakeWeb` *gem, for example)*

How much testing is enough?

- Bad: “Until time to ship”
- A bit better: (Lines of test) / (Lines of code)
  - 1.2–1.5 not unreasonable
  - often *much higher* for production systems
- Better question: “How thorough is my testing?”
  - Formal methods
  - Coverage measurement
  - We focus on the latter, though the former is gaining steady traction

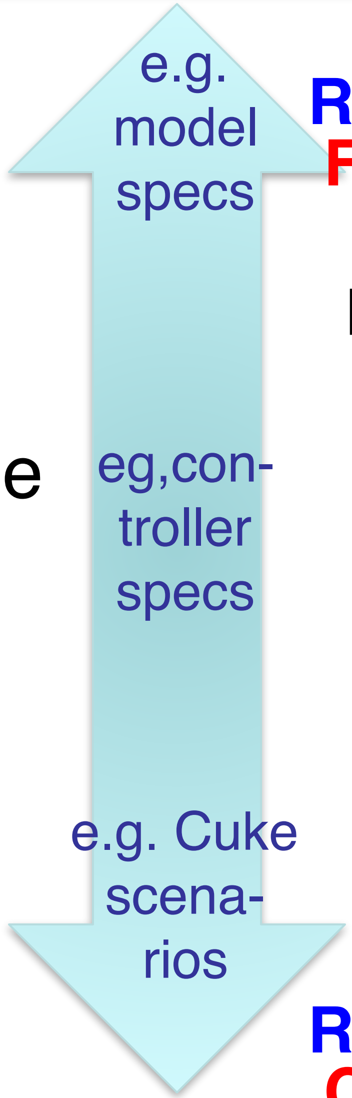
Measuring Coverage—Basics

```
class MyClass
 def foo(x,y,z)
  if x
   if (y && z) then bar(0) end
  else
   bar(1)
  end
 end
 def bar(x) ; @w = x ; end
end
```

- S0: every method called
- S1: every method *from every call site*
- C0: every statement
  - Ruby SimpleCov gem
- C1: every branch in both directions
- C1+decision coverage: every *subexpression* in conditional
- C2: every path (difficult, and disagreement on how valuable)

What kinds of tests?

- Unit (one method/ class)
- Functional or module (a few methods/ classes)
- Integration/system



Runs fast **High coverage**
Fine resolution
Many mocks;
Doesn't test interfaces

Few mocks;
tests interfaces
Runs slow **Low coverage**
Coarse resolution

TDD vs. Conventional debugging

Conventional	TDD
Write 10s of lines, run, hit bug: break out debugger	Write a few lines, with test first; know immediately if broken
Insert printf's to print variables while running repeatedly	Test short pieces of code using expectations
Stop in debugger, tweak/set variables to control code path	Use mocks and stubs to control code path
Dammit, I thought for sure I fixed it, now I have to do this all again	Re-run test automatically

- Lesson 1: TDD uses same skills & techniques as conventional debugging—but more productive (FIRST)

- Lesson 2: writing tests *before* code takes *more time* up-front, but often *less time* overall