Hasso Plattner Institut

IT Systems Engineering | Universität Potsdam

Software Engineering 2 (SWT2)

Chapter 3:
BDD and Testing (in Rails)

Agenda

- Why Behavior-driven Design (BDD)?
- Building Blocks of Tests and BDD
- Testing Tests & Hints for Successful Test Design
- Outlook

Agenda

- **Why Behavior-driven Design (BDD)?**
    - **Goals of Automated Testing**
    - The Case for BDD
    - Writing Software that Matters
- Building Blocks of Tests and BDD
- Testing Tests & Hints for Successful Test Design
- Outlook

Goals of Automated Developer Testing

- Feature 1: Website registration

Developer 1 (no TDD/BDD)

- Minute 5: working registration page
- Minute 8: feature is tested (3 times)

Developer 2: with TDD/BDD (almost no browser testing)

- Minute 5: working test
- Minute 10: working implementation
- Minute 10.30: feature is tested (3 times)

Goals of Automated Developer Testing

- Feature 2: Special case for feature 1

Developer 1 (no TDD/BDD)

- Minute 11: implemented
- Minute 14: tested (3 times)
- Minute 17: refactoring ready
- Minute 19: tested feature 1
- Minute 21: tested feature 2
- Minute 22: committed

Developer 2: with TDD/BDD (almost no browser testing)

- Minute 12.30: test ready
- Minute 15.30: implemented
- Minute 16.00: tested (3 times)
- Minute 19: refactoring ready
- Minute 19.10: tested
- Minute 20.10: committed

Goals of Automated Developer Testing

- Finding errors faster
- Better code (correct, robust, maintainable)
- Automated developer testing is frequently faster
- Easier to add new features
- Easier to modify existing features

- BUT
  - □ Tests might have bugs
  - □ Test environment != production environment
  - □ Code changes break tests
  - □ …

➔ we'll cover a bit of this in this lecture

Agenda

- **Why Behavior-driven Design (BDD)?**
  - Goals of Automated Testing
  - **The Case for BDD**
  - Writing Software that Matters
- Building Blocks of Tests and BDD
- Testing Tests & Hints for Successful Test Design
- Outlook

How Traditional Projects Fail

- Delivering late
- Delivering over budget
- Delivering the wrong thing
- Unstable in production
- Costly to maintain

Why Traditional Projects Fail

- Smart people trying to do good work
- Stakeholders are well intended

Process in traditional projects

- Planning → Analysis → Design → Code → Test → Deploy
- Much effort for
  - Documents for formalized hand-offs
  - Templates
  - Review committees
  - …

Why Traditional Projects Fail

- The later we find a defect, the more expensive to fix it
- Does front-loading a software development process make sense?

Reality shows

- Project plans are wonderful
- Adjustments/assumptions are made during analysis, design, code
- Re-planning takes place
- Example: testing phase
  - □ Tester raises a defect
  - □ Programmer claims he followed the specification
  - □ Architect blames business analyst etc.
  - □ → exponential cost

Why Traditional Projects Fail

- People are afraid of making changes
- Unofficial changes are carried out
- Documents get out of sync
- ...

- Again, why do we do that!?
- To minimize the risk of finding a defect to late

A Self-Fulfilling Prophecy

- We conduct the front-loaded process to minimize exponential costs of change
  - □ Project plan
  - □ Requirements spec
  - □ High-level design documents
  - □ Low-level design documents
- This process causes the exponential costs of change!

➔ A self-fulfilling prophecy

- Makes sense for a bridge, ship, or a building
- Software (and Lego) are EASY to change!

A Self-Fulfilling Prophecy

- Many teams work very successful with traditional approaches
- High degree of communication and collaboration is needed

- Behavior-driven development … an Agile methodology

Behavior-driven Development

People ask

- Why do so many software projects fail?
- Why are we consistently bad at delivering software?
- Why do larger teams suffer more?
- What can be done?

- A series of lightweight methods
- Focus: delivering working software to users

The Agile Manifesto

We are uncovering better ways of developing software by doing it and helping others do it.

Through this work we have come to value:

Individuals and interactions over processes and tools
Working software over comprehensive documentation
Customer collaboration over contract negotiation
Responding to change over following a plan

That is, while there is value in the items on the right, we value the items on the left more.

http://agilemanifesto.org/

How Agile Methods Address Project Risks

- No longer late or over budget
  - Tiny iterations
  - Easy to calculate budget
  - High-priority requirements first

- No longer delivering the wrong thing
  - Strong stakeholder communication
  - Short feedback cycles

How Agile Methods Address Project Risks

- No longer unstable in production
  - Delivering each iteration
  - High degree of automation

- No longer costly to maintain
  - Maintenance mode since Sprint 2
  - Maintenance of multiple versions during development

The Cost of Going Agile

- Outcome-based planning / no complete detailed project plan
- Streaming requirements / a new requirements process
- Evolving design / no complete upfront design → flexible
- Changing existing code / need for refactoring

The Cost of Going Agile

- Frequent code integration / continuous integration
- Continual regression testing / add nth feature; test n-1 features
- Frequent production releases / organizational challenges
- Co-located team / keep momentum

Agenda

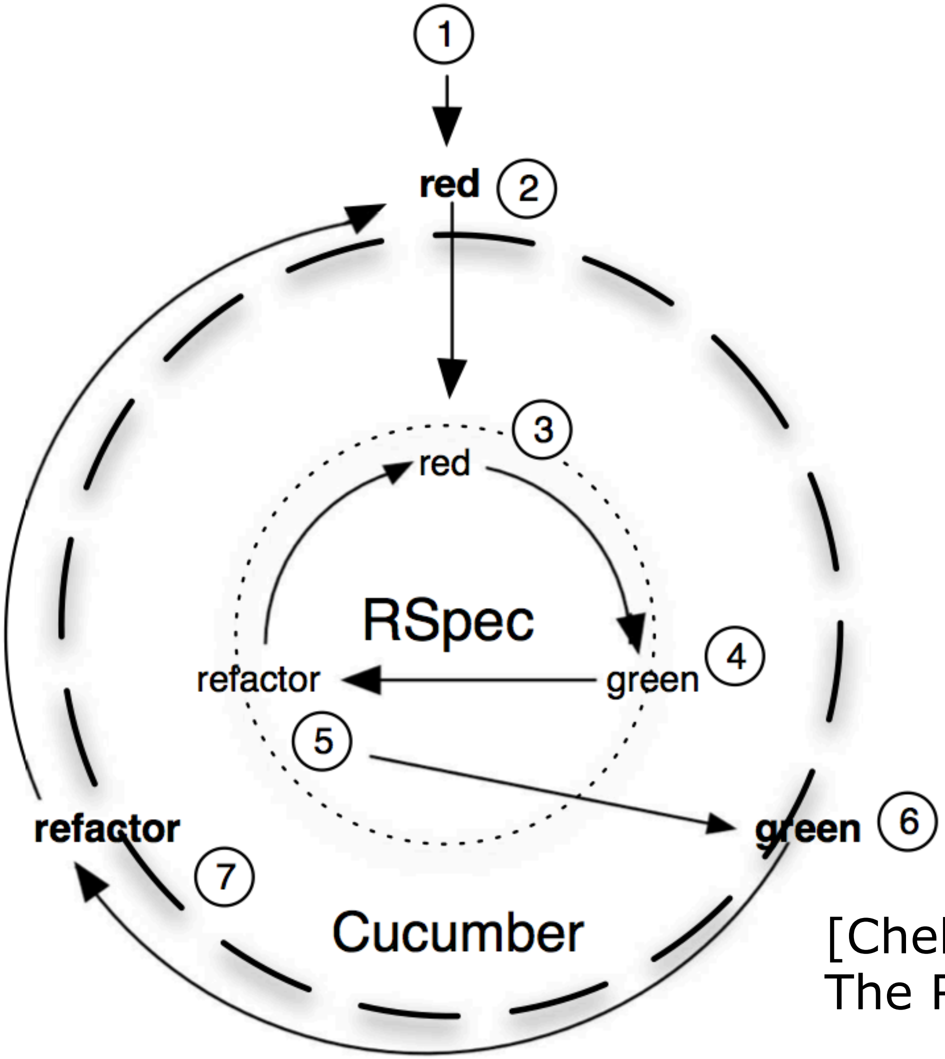
- **Why Behavior-driven Design (BDD)?**
  - Goals of Automated Testing
  - The Case for BDD
  - **Writing Software that Matters**
- Building Blocks of Tests and BDD
- Testing Tests & Hints for Successful Test Design
- Outlook

Writing Software that Matters

- "BDD is about implementing an application by describing its behavior from the perspective of its stakeholders"

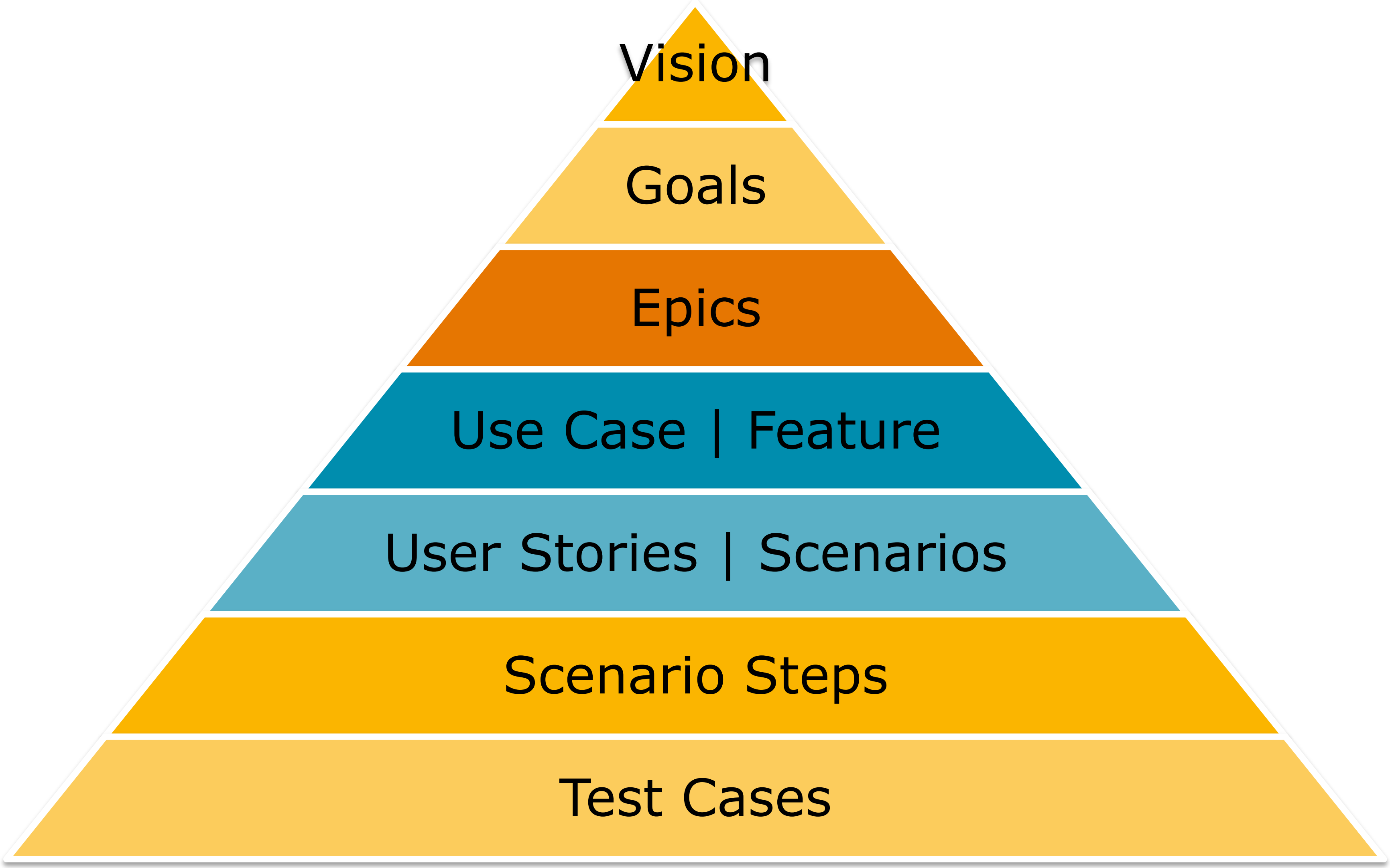
- Principles of BDD
  1. Enough is enough
  2. Deliver stakeholder value
  3. It's all behavior

BDD Cycle



[Chelimsky et al.:
The Rspec Book, 2010]

Maximum BDD Pyramid



Vision

- All Stakeholders, one statement
- Core stakeholders vs. incidental stakeholders
- Example: improve Supply Chain; understand customers better
- Core stakeholders have to define the vision
- Incidental stakeholders help understand
  - what is possible
  - at what cost
  - with what likelihood

Goals

- Goals have to be identified

- Examples
  - Easier ordering process
  - Better access to suppliers’ information

- Goals should be SMART
  - Specific (when is it done?)
  - Measurable (was the objective reached?)
  - Achievable (reduce unrealistic expectations)
  - Relevant (not every exception is relevant)
  - Timeboxed (certain investment in time)

Epics

- Huge themes / feature sets are described as an “epic”
- Too high level to start coding
- Useful for conversations
- Examples
  - Reporting
  - Customer registration

Use Case | Features

- Describe the behavior we will implement in software
- Can be traced back to a stakeholder
- Warning: do not directly start at this level
- Is it a waterfall process?
  - Yes, we think about goals to be achieved
  - No, we just do enough
- Explain the value/context of a feature to stakeholders → not too much detail
- Features deliver value to stakeholders

User Stories

- Stories are demonstrable functionality
- Attributes (INVEST)
  - **I**ndependent
  - **N**egotiable
  - **V**aluable (from a business Point of View)
  - **E**stimable
  - **S**mall enough to be implemented in one iteration
  - **T**estable
- 1 feature → 1..n User Stories
- Stories should be vertical
- A token for a conversation

User Stories

- Story content
  - Title
  - Narrative
    - Description, reason, benefit
    - “As a <stakeholder>, I want <feature> so that <benefit>”
    - “In order to <benefit>, a <stakeholder> wants to <feature>”
  - Acceptance criteria

- Stories in, features out

Scenarios, Scenario Steps, Test Cases

- 1 User Story → 1..n scenarios
- Each scenario describes one aspect of a User Story
- Describe high-level behavior

- 1 scenario → m scenario steps + step implementation
- Given – When – Then

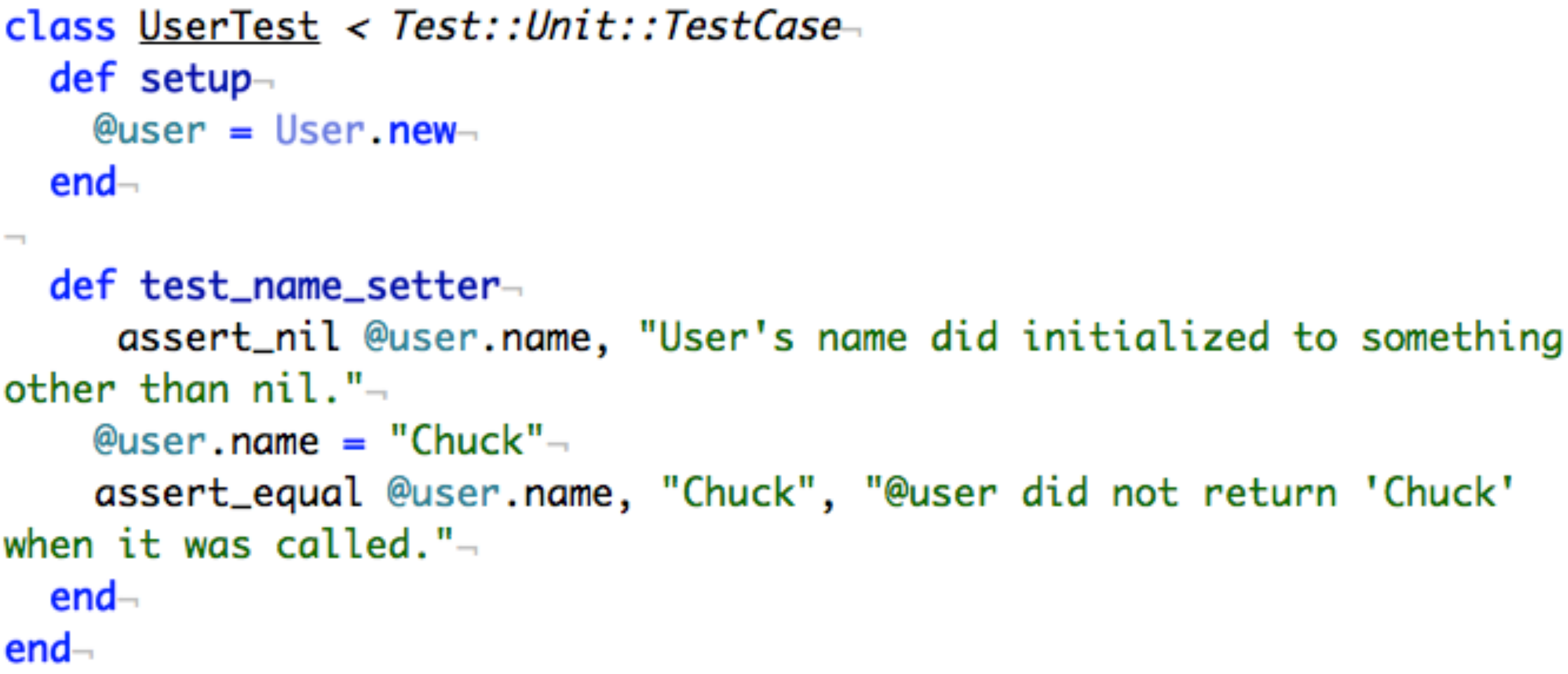
- 1 scenario step → 0..i tests (e.g., in RSpec)
- Describe low-level behavior

Agenda

- Why Behavior-driven Design (BDD)?
- **Building Blocks of Tests and BDD**
  - □ Test Data
  - □ Test Doubles
  - □ Setup and Teardown
  - □ Model Tests
  - □ View Tests
  - □ Controller Tests
  - □ Routing Tests
  - □ Outgoing Mail Tests
  - □ Helper Tests
  - □ Integration and Acceptance Tests
- Testing Tests & Hints for Successful Test Design
- Outlook

Test::Unit vs. RSpec

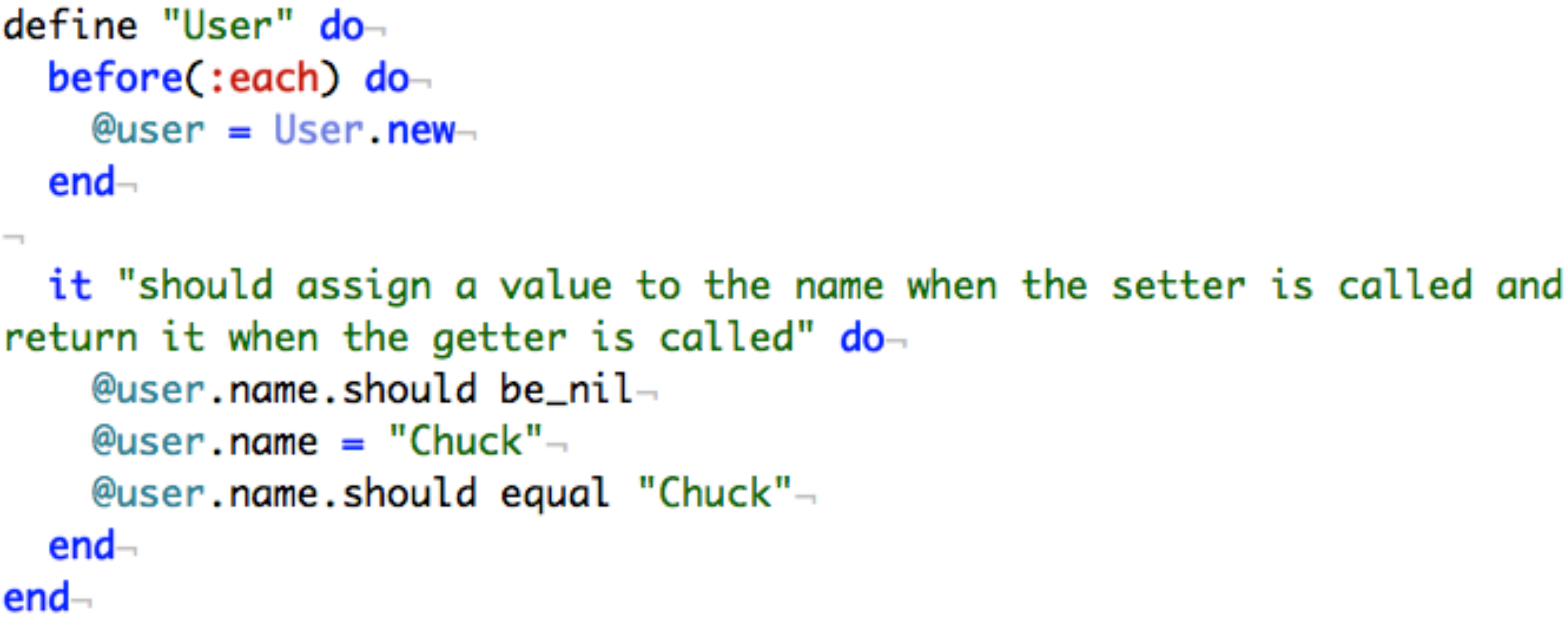
- Test::Unit comes with Ruby



```
class UserTest < Test::Unit::TestCase¬
  def setup¬
    @user = User.new¬
  end¬
¬
  def test_name_setter¬
     assert_nil @user.name, "User's name did initialized to something
other than nil."¬
    @user.name = "Chuck"¬
    assert_equal @user.name, "Chuck", "@user did not return 'Chuck'
when it was called."¬
  end¬
end¬
```

Test::Unit vs. RSpec

- RSpec has syntactical sugar in it



```
define "User" do¬
  before(:each) do¬
    @user = User.new¬
  end¬
¬
  it "should assign a value to the name when the setter is called and
return it when the getter is called" do¬
    @user.name.should be_nil¬
    @user.name = "Chuck"¬
    @user.name.should equal "Chuck"¬
  end¬
end¬
```

- We’ll use RSpec ;)

http://teachmetocode.com/articles/rspec-vs-testunit/

Agenda

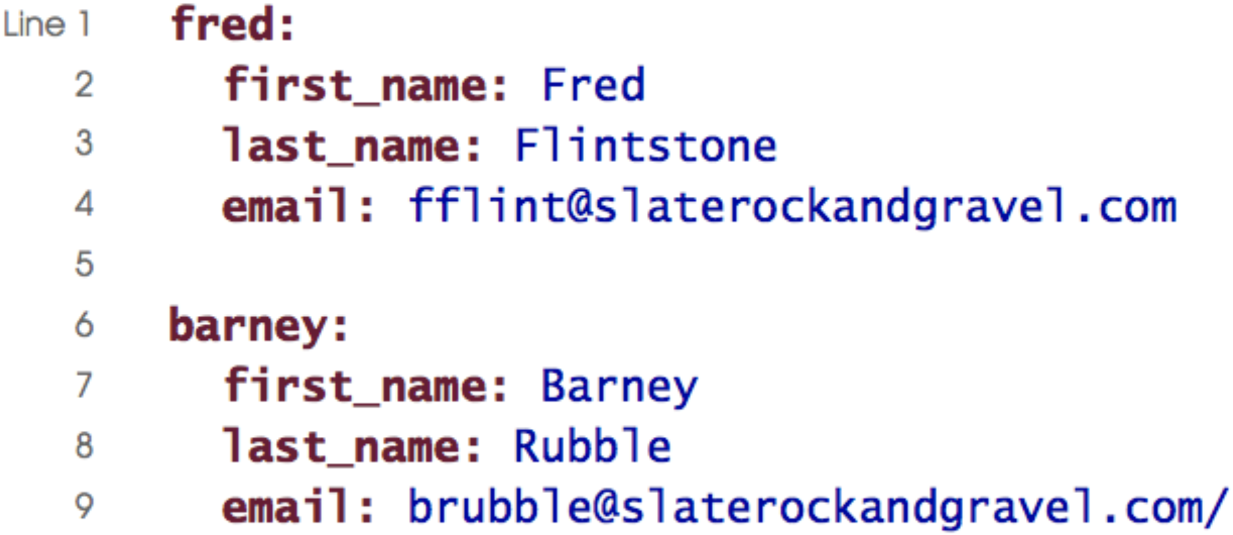
- Why Behavior-driven Design (BDD)?
- Building Blocks of Tests and BDD
  - □ **Test Data**
  - □ Test Doubles
  - □ Setup and Teardown
  - □ Model Tests
  - □ View Tests
  - □ Controller Tests
  - □ Routing Tests
  - □ Outgoing Mail Tests
  - □ Helper Tests
  - □ Integration and Acceptance Tests
- Testing Tests & Hints for Successful Test Design
- Outlook

Test Data Overview

- Fixtures
  - Fixed state at the beginning of a test
  - Assertions can be made against this state
- Factories
  - Blueprint for models
  - Used to generate test data locally in the test

Fixtures

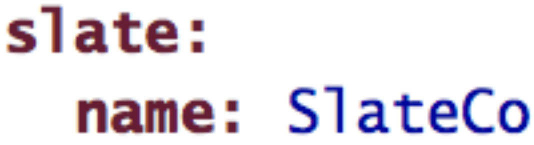
test/fixtures/users.yml



```
fred:
  first_name: Fred
  last_name: Flintstone
  email: fflint@slaterockandgravel.com

barney:
  first_name: Barney
  last_name: Rubble
  email: brubble@slaterockandgravel.com/
```

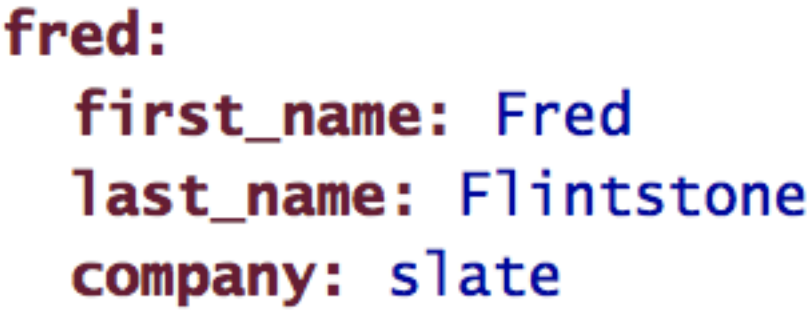
Fixtures, 1:n relationship

test/fixtures/companies.yml



```
slate:
  name: SlateCo
```

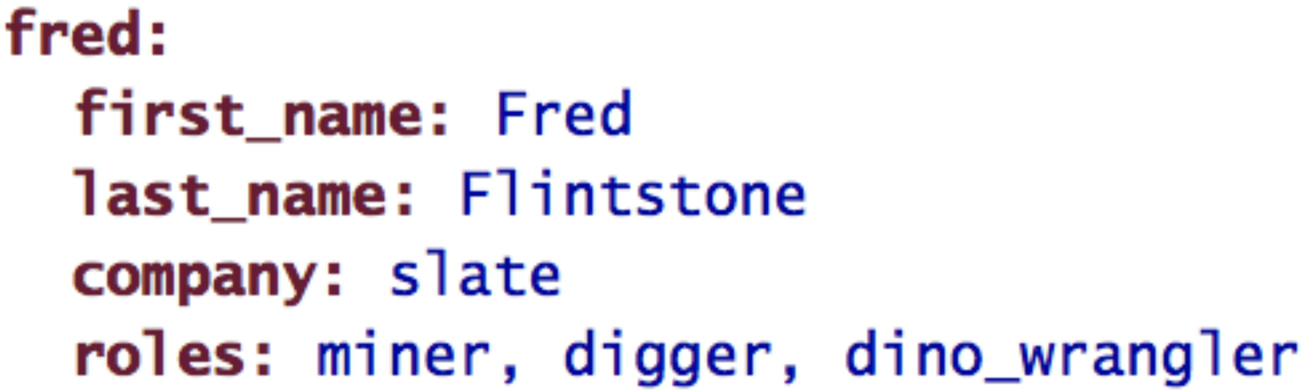
test/fixtures/users.yml



```
fred:
  first_name: Fred
  last_name: Flintstone
  company: slate
```

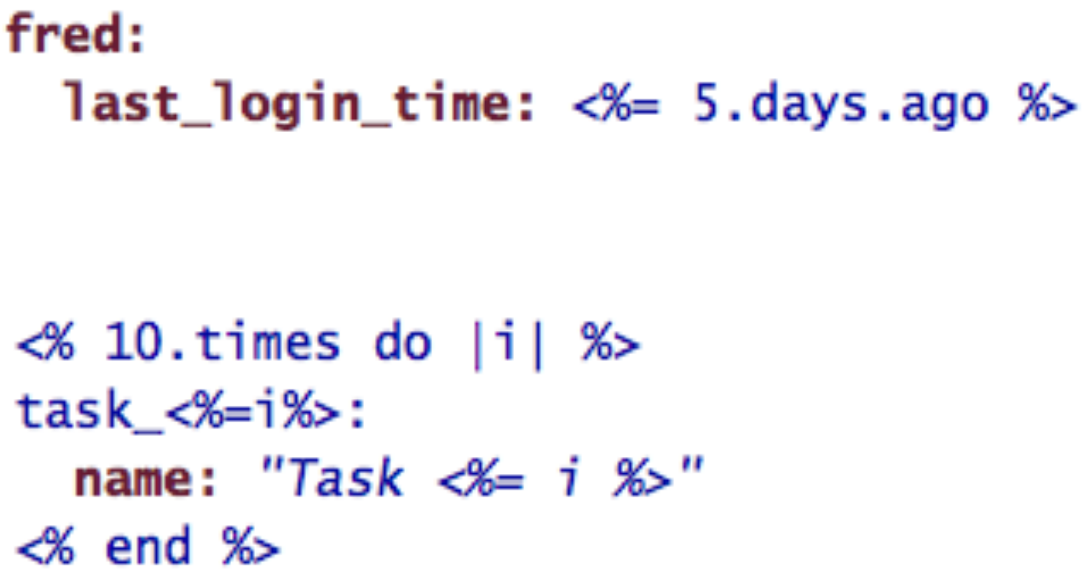
Fixtures, n:m relationship

test/fixtures/users.yml



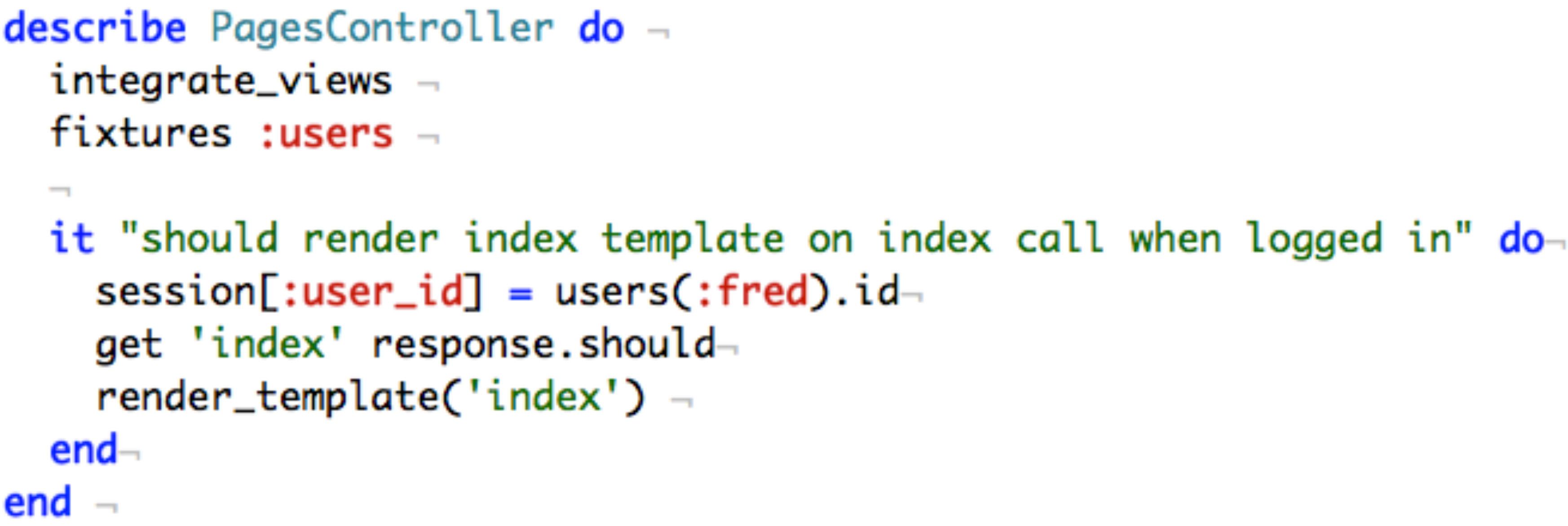
```
fred:
  first_name: Fred
  last_name: Flintstone
  company: slate
  roles: miner, digger, dino_wrangler
```

Fixtures are ERB Files



```
fred:
  last_login_time: <%= 5.days.ago %>


<% 10.times do |i| %>
task_<%=i%>:
  name: "Task <%= i %>"
<% end %>
```

Loading Fixtures



```
describe PagesController do ¬
  integrate_views ¬
  fixtures :users ¬
  ¬
  it "should render index template on index call when logged in" do¬
    session[:user_id] = users(:fred).id¬
    get 'index' response.should¬
    render_template('index') ¬
  end¬
end ¬
```

Why Fixtures are a Pain

- Fixtures are global
  - Only ONE set of data
  - Every test has to deal with ALL test data

- Fixtures are spread out
  - Own directory
  - One file per model → data for one test is spread out over many files
  - Tracing relationships is a pain

Why Fixtures are a Pain

- Fixtures are distant
  - A test fails
  - It is unclear which data is used
  - How are values computed?
  - assert_equal(users(:ernie).age + users(:bert).age), 20)

- Fixtures are brittle
  - Tests rely on this data
  - Tests break when data is changed
  - Data requirements may be incompatible

Fixing Fixtures with Factories

Test data should be

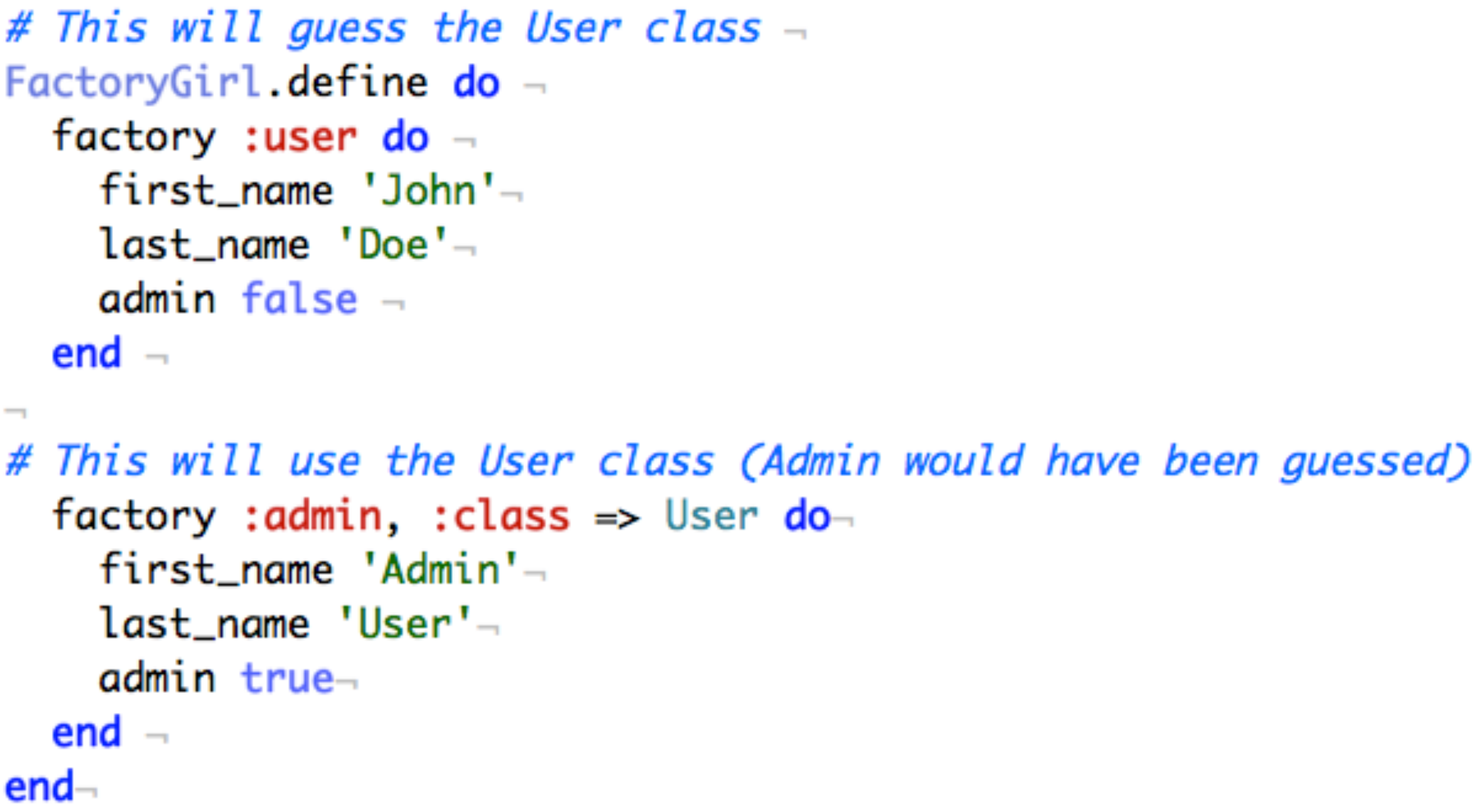
- Local (defined as closely as possible to the test)
- Compact (easy and quick to generate; even complex data sets)
- Robust (independent to other tests)

➔ Data factories

Data Factories

- Blueprint for sample instances
- Rails tool support
  - Factory Girl
  - Machinist
  - Fabrication
  - FictureBuilder
  - ObjectDaddy
  - …
  - https://www.ruby-toolbox.com/categories/rails_fixture_replacement
- Similar structure
  - Syntax for creating the factory blueprint
  - API for creating new objects
- We’ll use factory_girl

Defining Factories



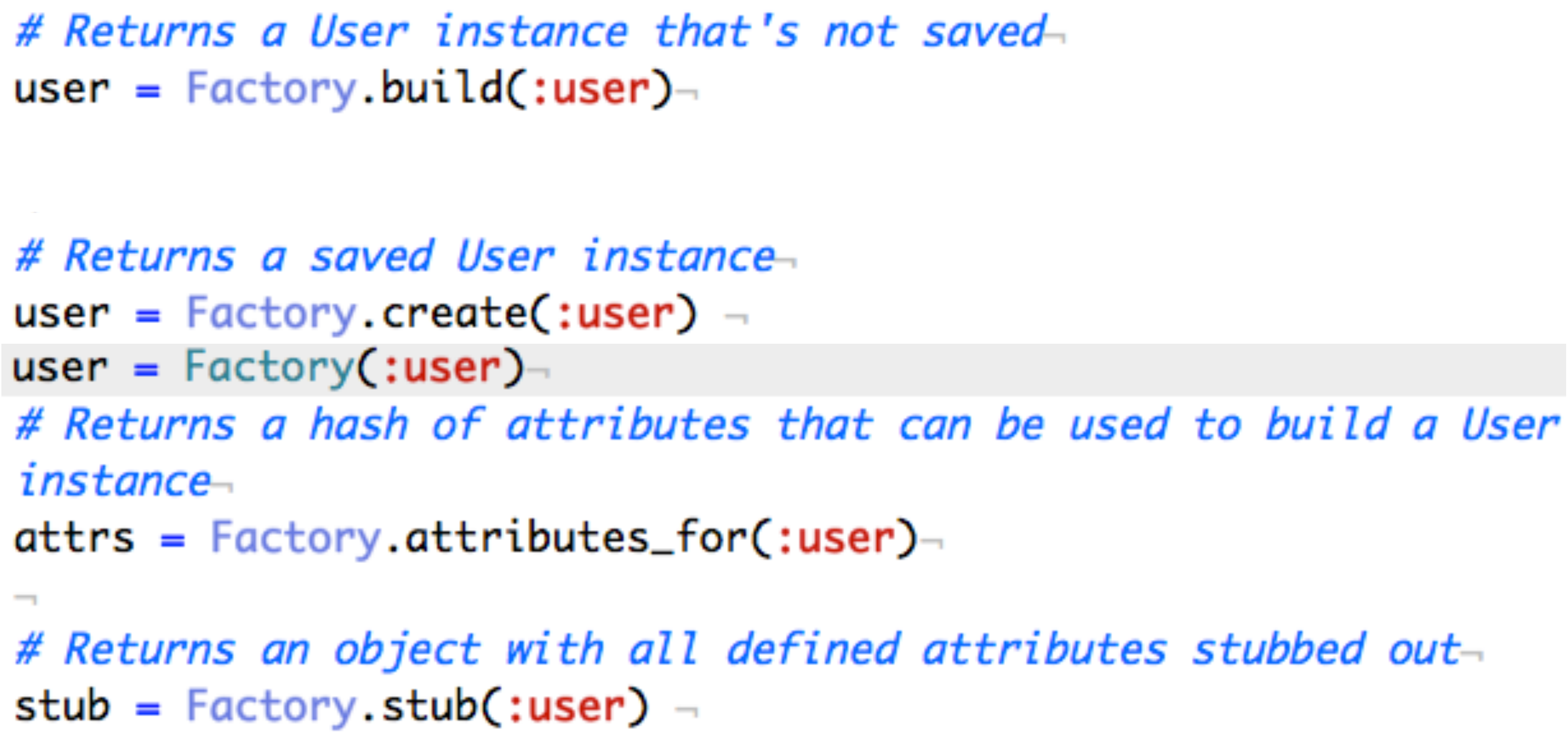
```ruby
# This will guess the User class
FactoryGirl.define do
  factory :user do
    first_name 'John'
    last_name 'Doe'
    admin false
  end

# This will use the User class (Admin would have been guessed)
  factory :admin, :class => User do
    first_name 'Admin'
    last_name 'User'
    admin true
  end
end
```

Defining Factories

- One most simplistic factory <model_name> for each class
- Put in
  - test/factories.rb
  - spec/factories.rb
  - test/factories/*.rb
  - **spec/factories/*.rb ←** with * = <model_name>

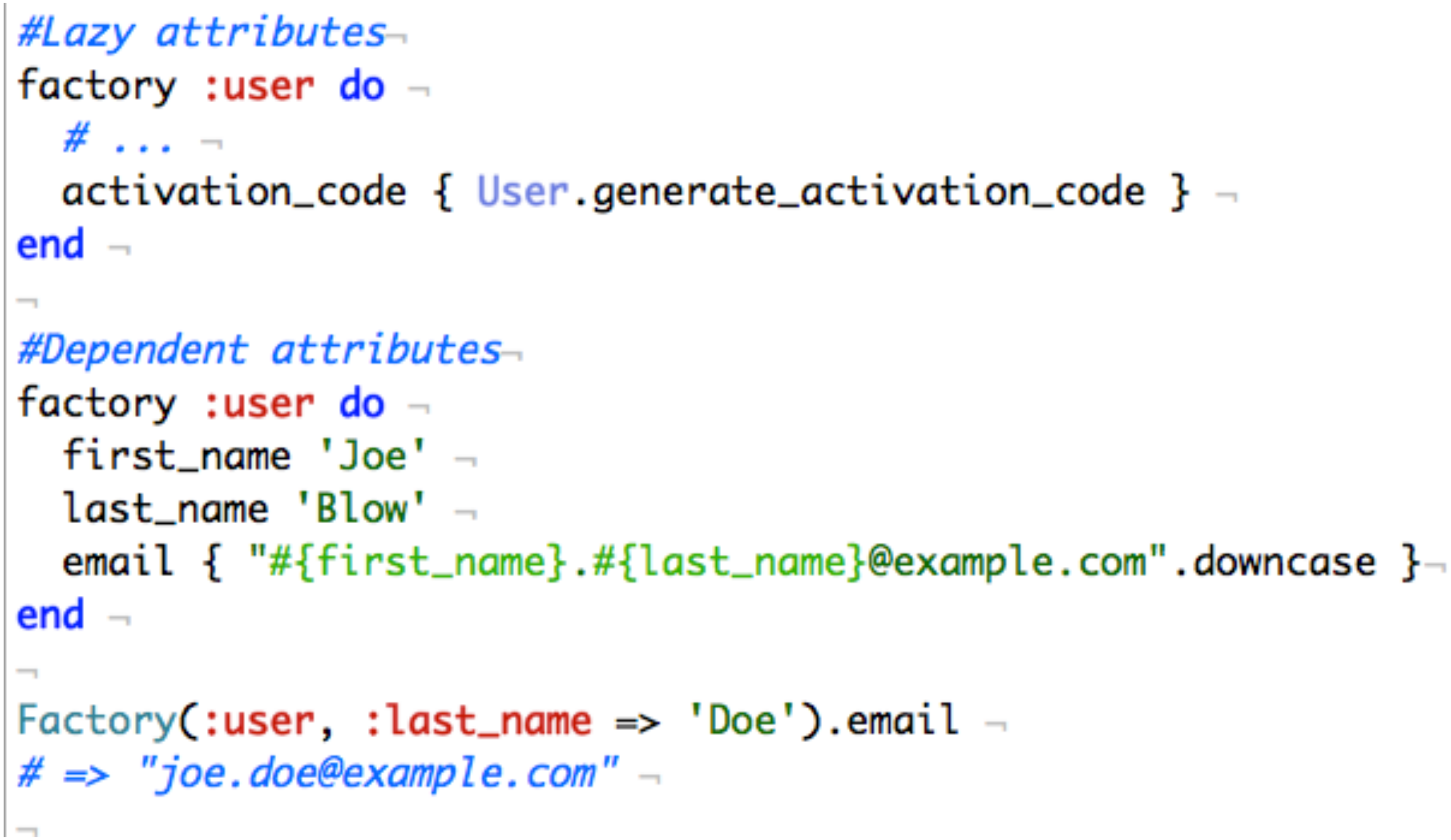
Using Factories

- Build strategies: build, create ← standard, attributes_for, stub



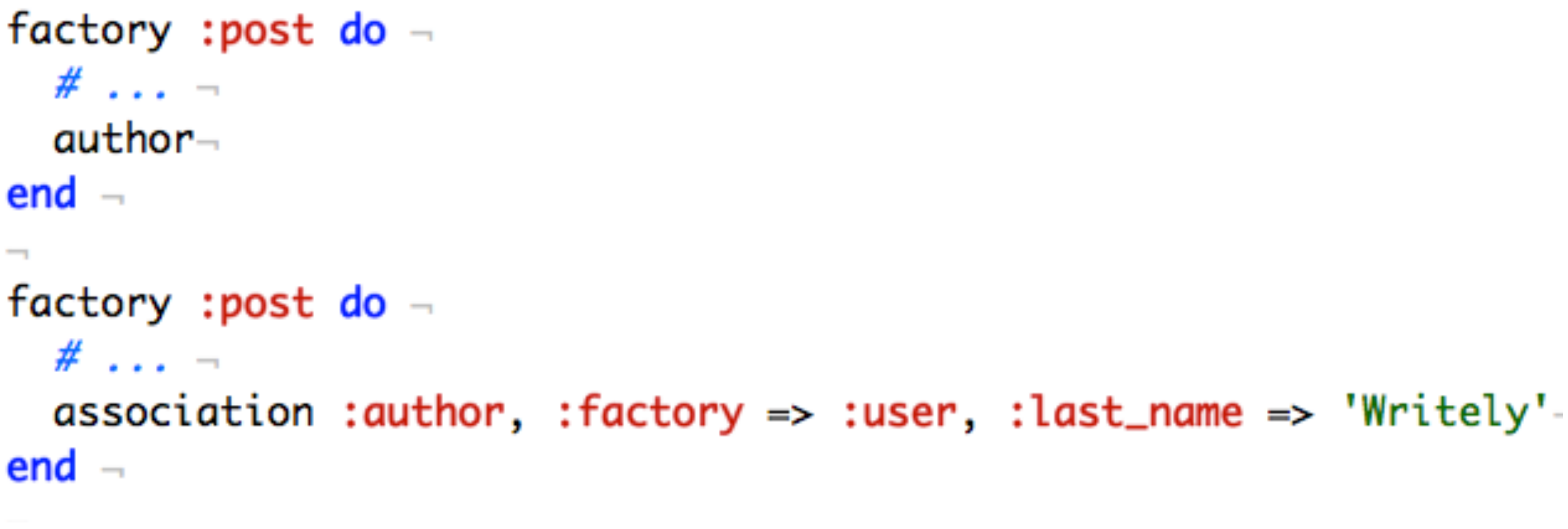
```
# Returns a User instance that's not saved
user = Factory.build(:user)


# Returns a saved User instance
user = Factory.create(:user)
user = Factory(:user)
# Returns a hash of attributes that can be used to build a User
instance
attrs = Factory.attributes_for(:user)

# Returns an object with all defined attributes stubbed out
stub = Factory.stub(:user)
```

Attributes



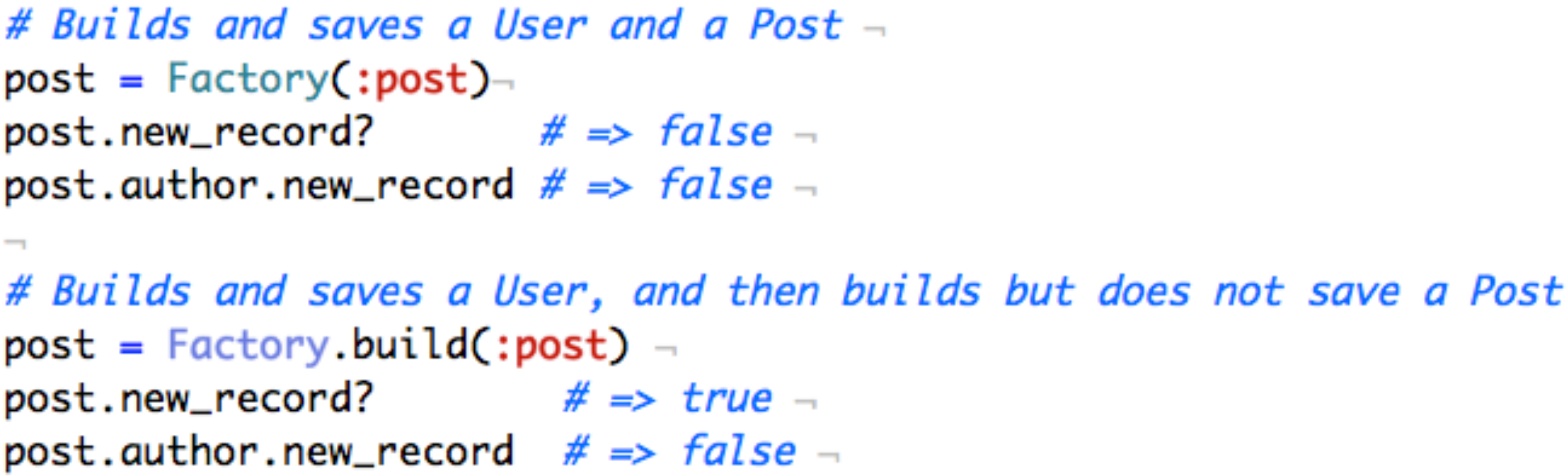
```
#Lazy attributes
factory :user do
  # ...
  activation_code { User.generate_activation_code }
end

#Dependent attributes
factory :user do
  first_name 'Joe'
  last_name 'Blow'
  email { "#{first_name}.#{last_name}@example.com".downcase }
end

Factory(:user, :last_name => 'Doe').email
# => "joe.doe@example.com"
```

Associations



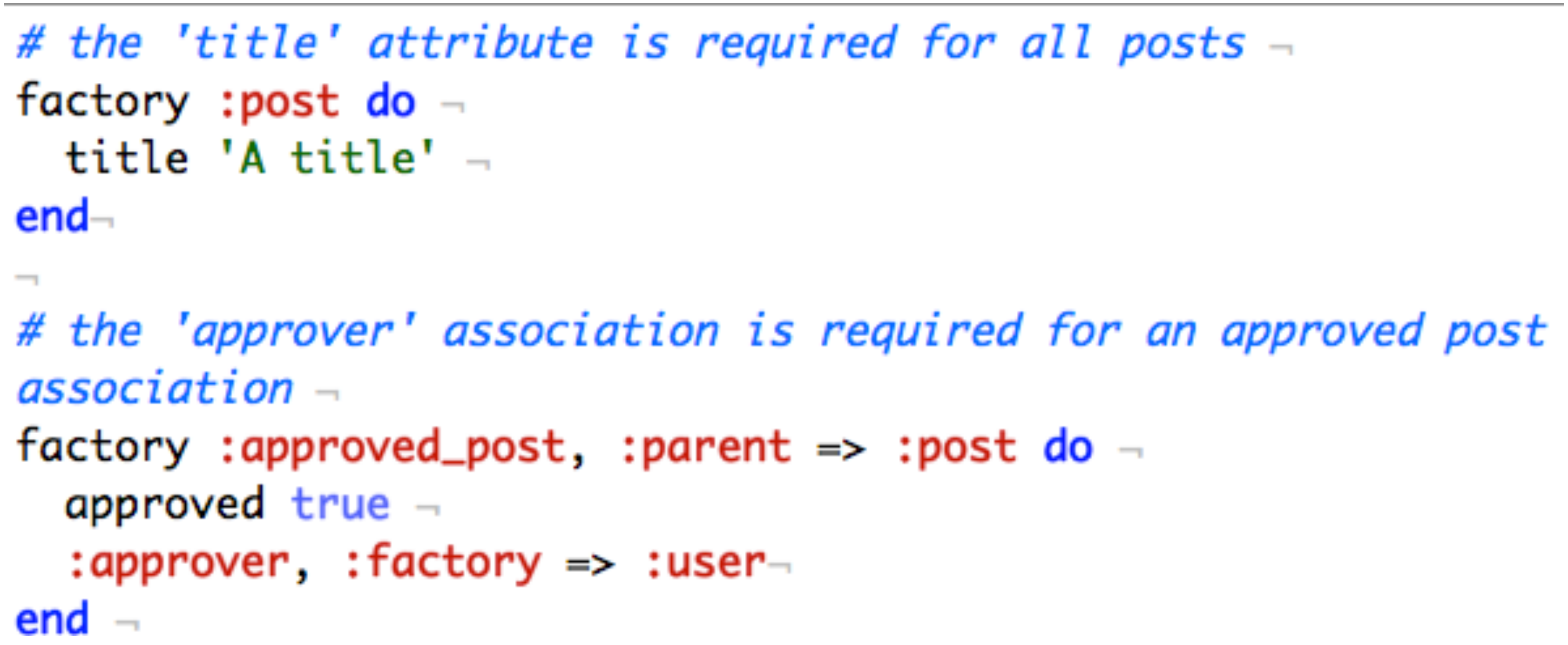
```
factory :post do
  # ...
  author
end

factory :post do
  # ...
  association :author, :factory => :user, :last_name => 'Writely'
end
```

Associations



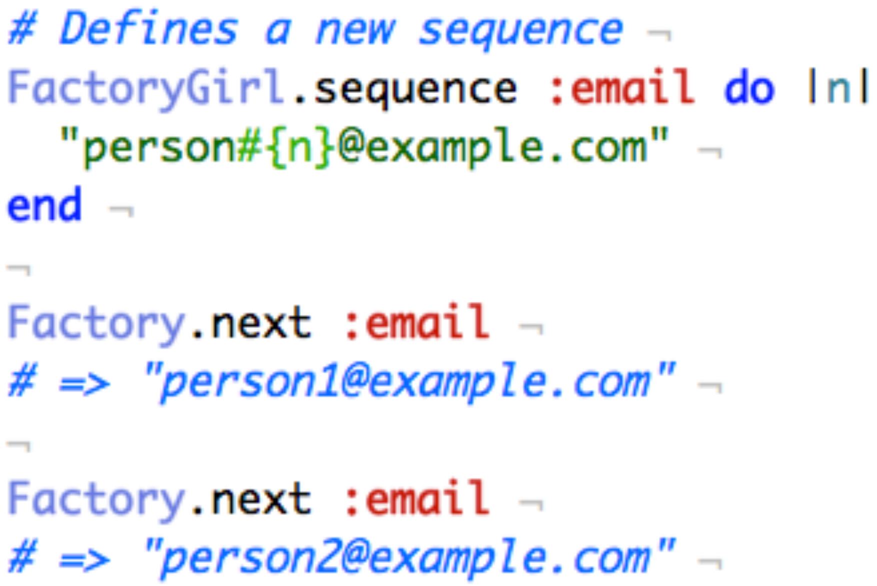
```
# Builds and saves a User and a Post ¬
post = Factory(:post)¬
post.new_record?       # => false ¬
post.author.new_record # => false ¬
¬
# Builds and saves a User, and then builds but does not save a Post
post = Factory.build(:post) ¬
post.new_record?        # => true ¬
post.author.new_record  # => false ¬
```

Inheritance



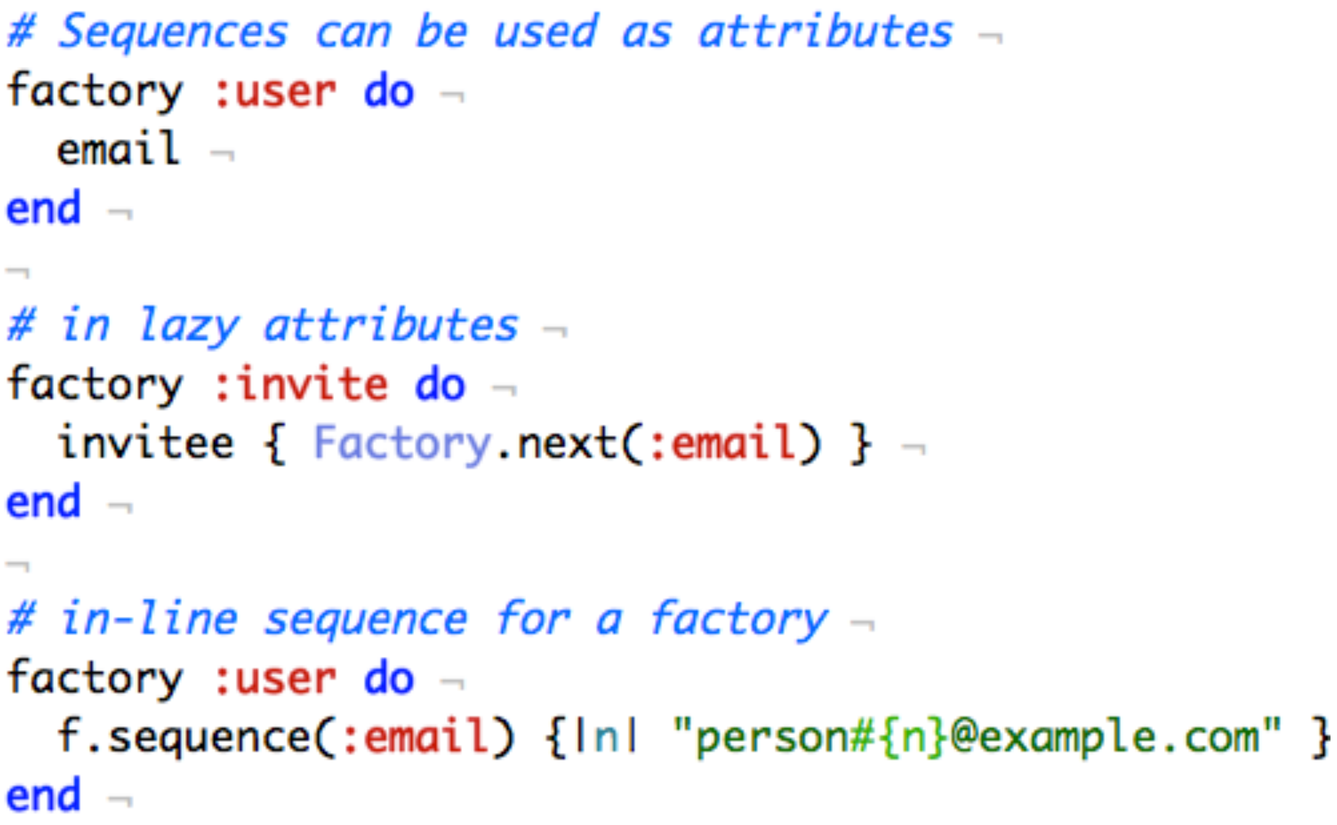
```
# the 'title' attribute is required for all posts ¬
factory :post do ¬
  title 'A title' ¬
end¬
¬
# the 'approver' association is required for an approved post
association ¬
factory :approved_post, :parent => :post do ¬
  approved true ¬
  :approver, :factory => :user¬
end ¬
```

Sequences for Unique Values



```
# Defines a new sequence ¬
FactoryGirl.sequence :email do |n|
  "person#{n}@example.com" ¬
end ¬
¬
Factory.next :email ¬
# => "person1@example.com" ¬
¬
Factory.next :email ¬
# => "person2@example.com" ¬
```

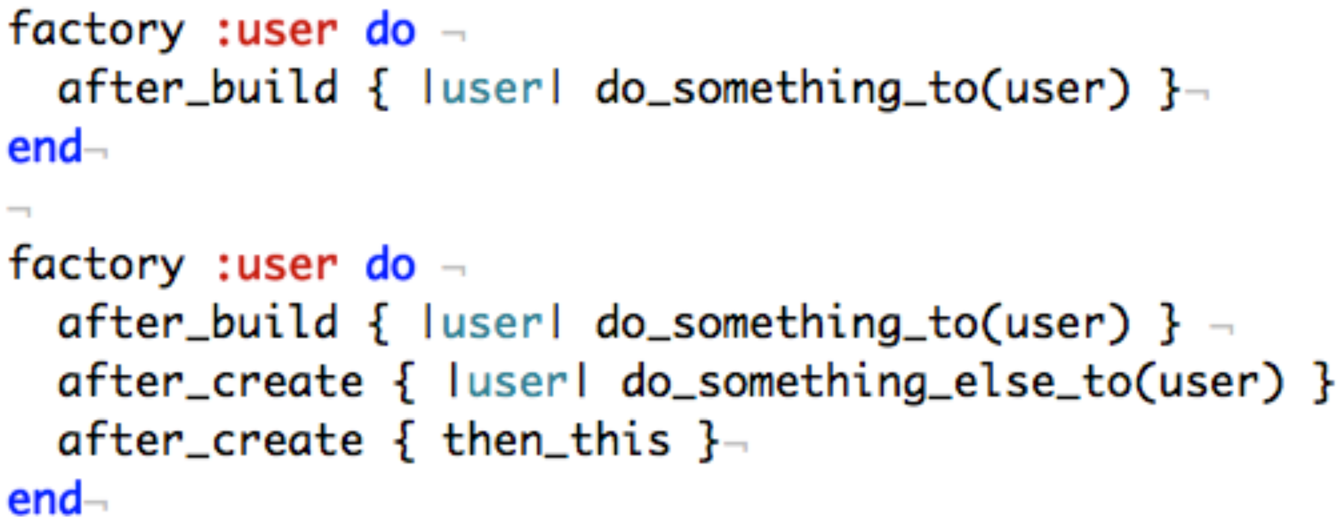
Sequences for Unique Values



```
# Sequences can be used as attributes
factory :user do
  email
end

# in lazy attributes
factory :invite do
  invitee { Factory.next(:email) }
end

# in-line sequence for a factory
factory :user do
  f.sequence(:email) {|n| "person#{n}@example.com" }
end
```

Callbacks

- after_build - called after a factory is built (via Factory.build)
- after_create - called after a factory is saved (via Factory.create)
- after_stub - called after a factory is stubbed (via Factory.stub)



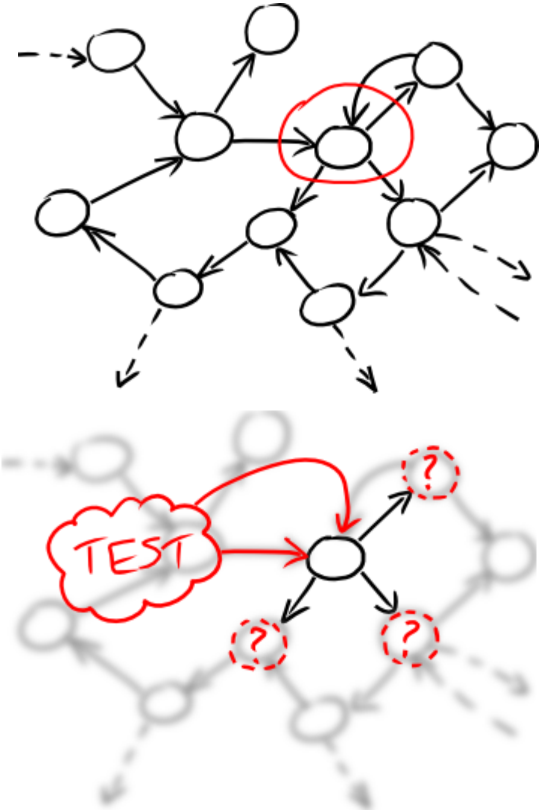
```ruby
factory :user do
  after_build { |user| do_something_to(user) }
end

factory :user do
  after_build { |user| do_something_to(user) }
  after_create { |user| do_something_else_to(user) }
  after_create { then_this }
end
```

Agenda

- Why Behavior-driven Design (BDD)?
- Building Blocks of Tests and BDD
  - Test Data
  - **Test Doubles**
    - **Introduction**
    - Stubs in Detail
    - Mocks in Detail
  - Setup and Teardown
  - Model Tests
  - View Tests
  - Controller Tests
  - …
- Testing Tests
- Outlook

Isolation of Test Cases

- Tests should be independent
- New bug in a model → only tests related to this model should fail
- How to achieve this?
  - □ Don’t share complex test data ✔
  - □ Don’t use complex objects



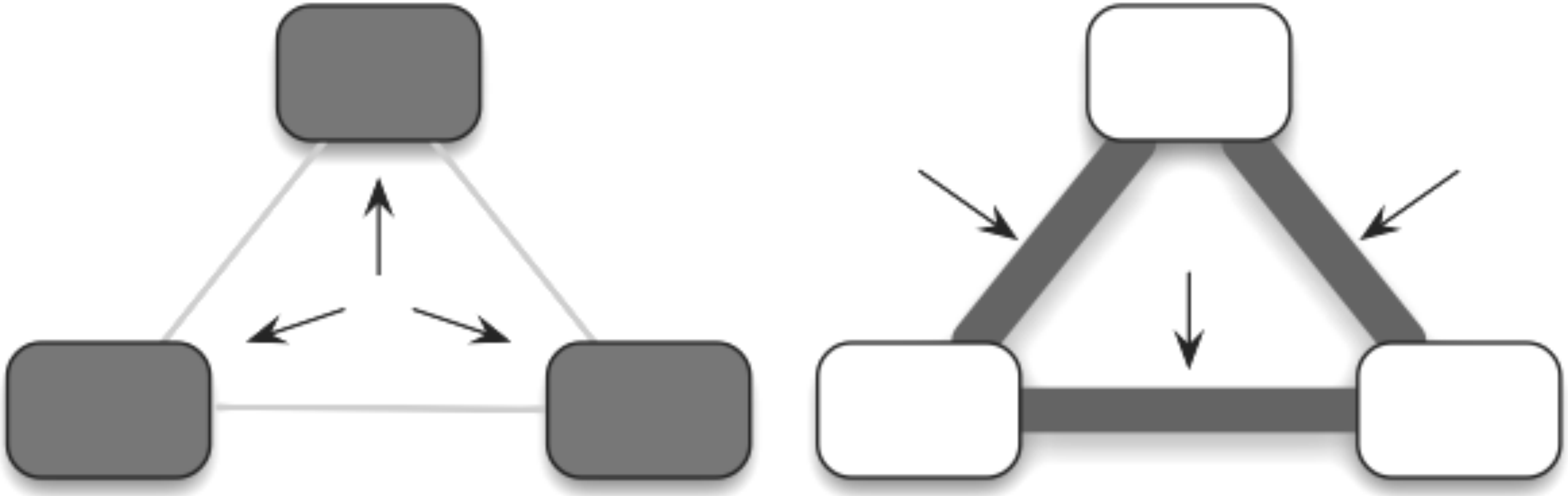
Steve Freeman, Nat Pryce: Growing Object-Oriented Software, Guided by Tests

Test Doubles

- Fake objects used in place of “real” ones
- Purpose: automated testing
- Used when
  - real object is unavailable
  - real object is difficult to
    - access or
    - trigger
  - following a strategy to re-create an application state
  - limiting scope of the test to the object/method currently under test

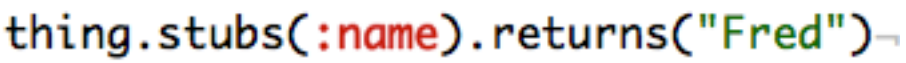
Verifying Behavior During a Test

- Usually: test system state AFTER a test
- With test doubles: test system behavior!



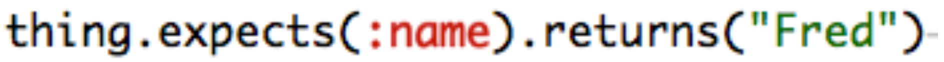
Stubs vs. Mocks

- Stub (passive)
  - □ Returns a predetermined value for a method call
  - □ Does not actually call the method



```
thing.stubs(:name).returns("Fred")
```

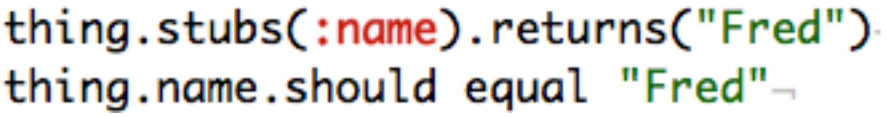
- Mock (more aggressive)
  - □ In addition: set an assertion
  - □ If expectation is not met → test failure



```
thing.expects(:name).returns("Fred")
```

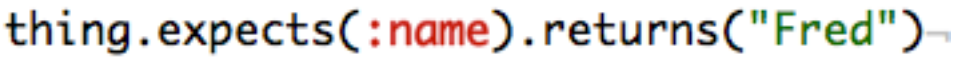
Why to have Mocks?

- Makes sense?



```
thing.stubs(:name).returns("Fred")
thing.name.should equal "Fred"
```

- Makes more sense?



```
thing.expects(:name).returns("Fred")
```

Ruby Test Double Frameworks

- Rspec-mocks (http://github.com/rspec/rspec-mocks)
- Mocha (http://mocha.rubyforge.org/)
- FlexMock (http://flexmock.rubyforge.org/)
- https://www.ruby-toolbox.com/categories/mocking

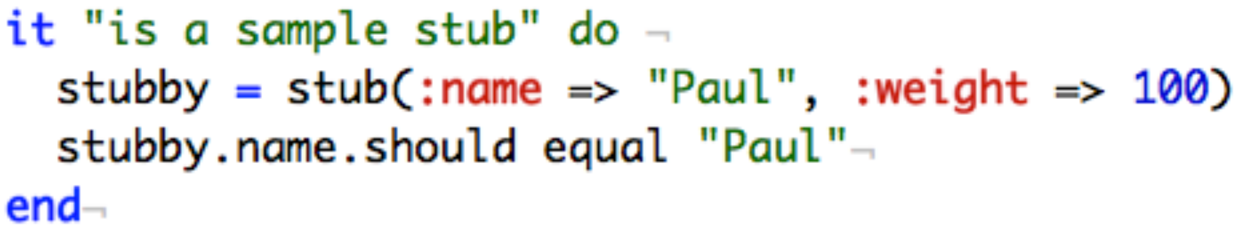
- We'll use Mocha

Agenda

- Why Behavior-driven Design (BDD)?
- Building Blocks of Tests and BDD
  - Test Data
  - **Test Doubles**
    - Introduction
    - **Stubs in Detail**
    - Mocks in Detail
  - Setup and Teardown
  - Model Tests
  - View Tests
  - Controller Tests
  - …
- Testing Tests
- Outlook

Stubs

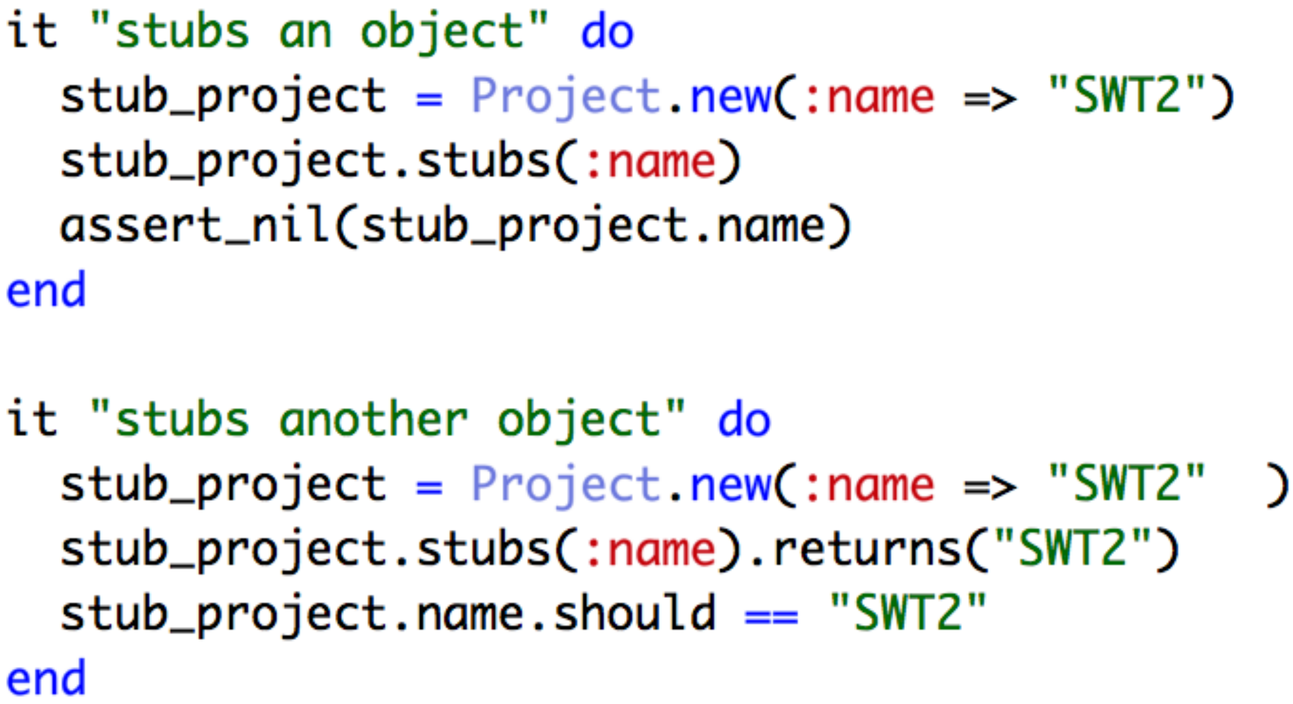
- Replacement for one or many parts of an object
- Normal method call is not happening
- Returns a predefined value if called



```
it "is a sample stub" do
  stubby = stub(:name => "Paul", :weight => 100)
  stubby.name.should equal "Paul"
end
```

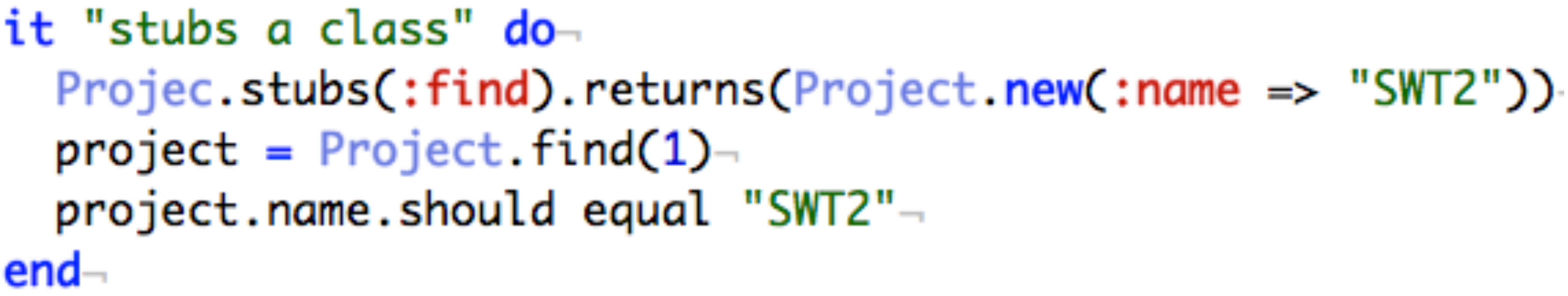
- You can only call stubby.name or stubby.weight
- Else: error
- Or: stub_everything(...) → nil

Stubbing Instances



```
it "stubs an object" do
  stub_project = Project.new(:name => "SWT2")
  stub_project.stubs(:name)
  assert_nil(stub_project.name)
end

it "stubs another object" do
  stub_project = Project.new(:name => "SWT2"  )
  stub_project.stubs(:name).returns("SWT2")
  stub_project.name.should == "SWT2"
end
```

Stubbing Classes

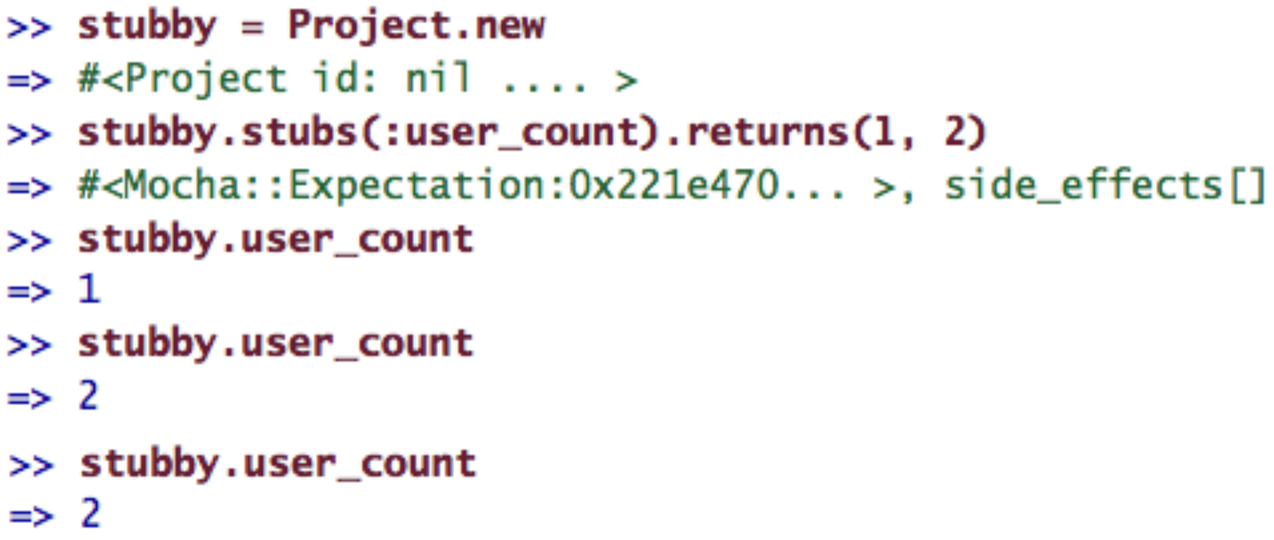


```
it "stubs a class" do
  Projec.stubs(:find).returns(Project.new(:name => "SWT2"))
  project = Project.find(1)
  project.name.should equal "SWT2"
end
```

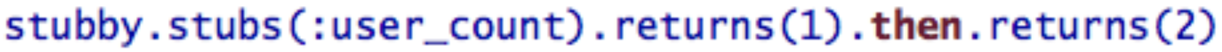
- A specific instance is returned
- Database is not touched

- "find" cannot be verified anymore BUT
- Tests based on "find" can be isolated
- ➔ just test the logic that is under test

Multiple Return Values

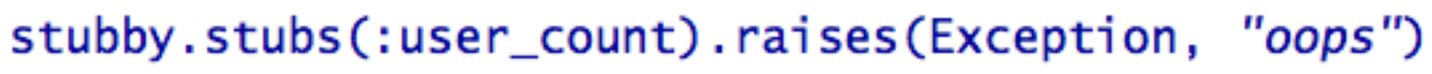


```
>> stubby = Project.new
=> #<Project id: nil .... >
>> stubby.stubs(:user_count).returns(1, 2)
=> #<Mocha::Expectation:0x221e470... >, side_effects[]
>> stubby.user_count
=> 1
>> stubby.user_count
=> 2
>> stubby.user_count
=> 2
```

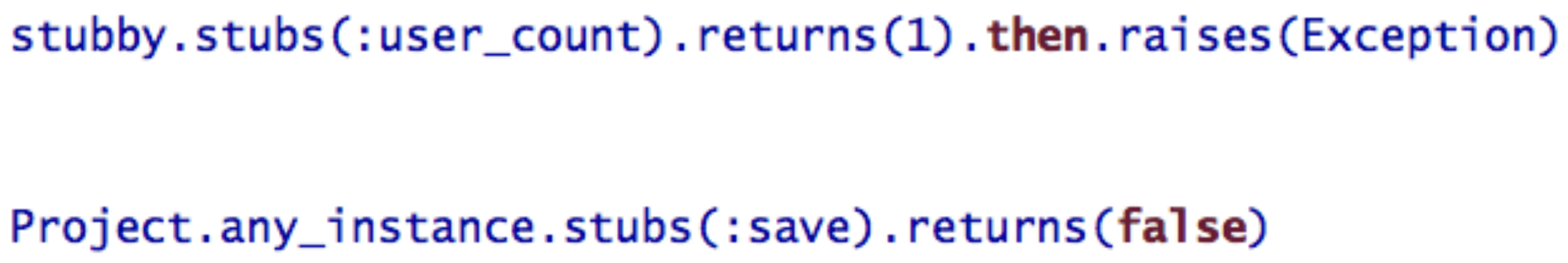


```
stubby.stubs(:user_count).returns(1).then.returns(2)
```

Stub Returns and Raises

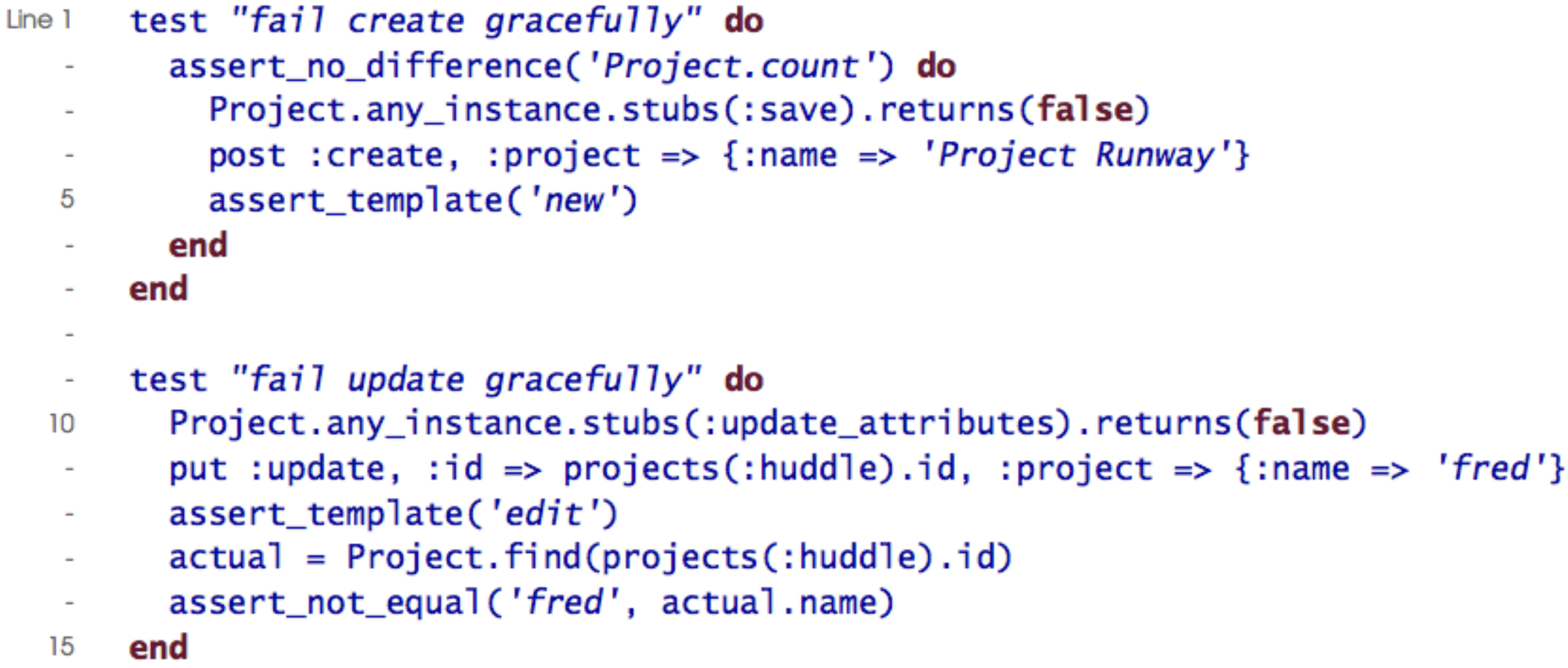


```
stubby.stubs(:user_count).raises(Exception, "oops")
```



```
stubby.stubs(:user_count).returns(1).then.raises(Exception)

Project.any_instance.stubs(:save).returns(false)
```

Examples

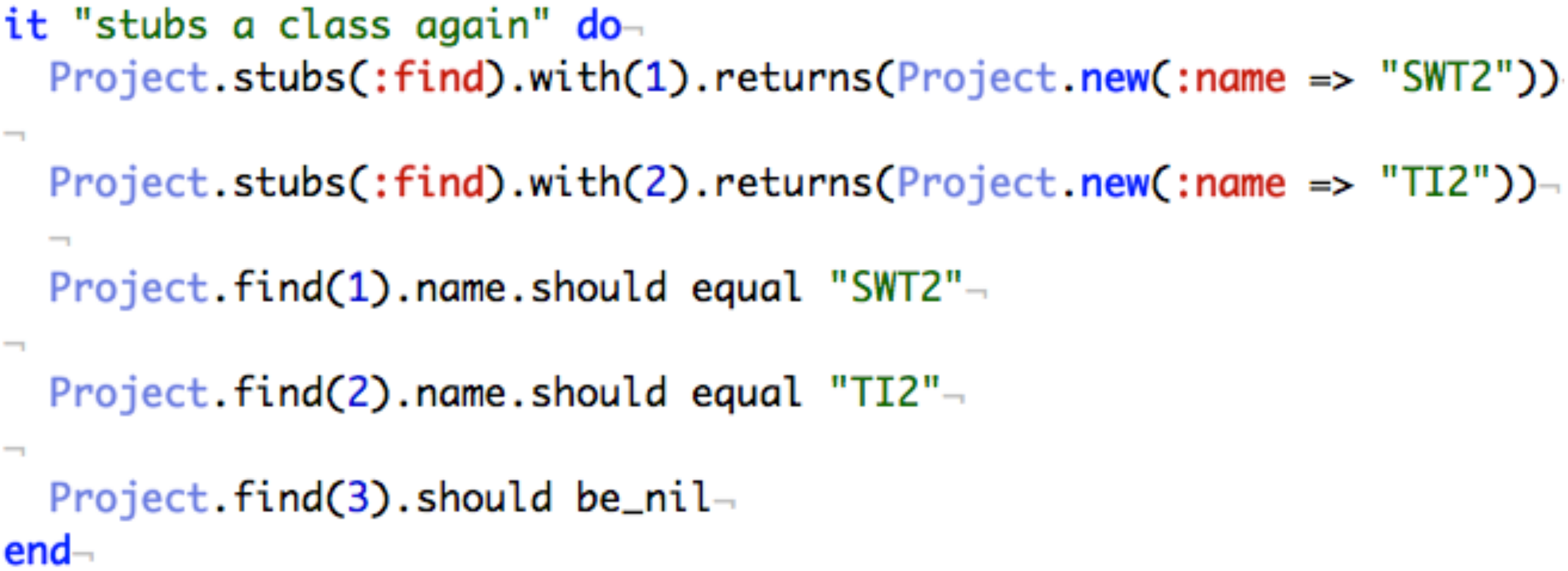


```
test "fail create gracefully" do
  assert_no_difference('Project.count') do
    Project.any_instance.stubs(:save).returns(false)
    post :create, :project => {:name => 'Project Runway'}
    assert_template('new')
  end
end

test "fail update gracefully" do
  Project.any_instance.stubs(:update_attributes).returns(false)
  put :update, :id => projects(:huddle).id, :project => {:name => 'fred'}
  assert_template('edit')
  actual = Project.find(projects(:huddle).id)
  assert_not_equal('fred', actual.name)
end
```

Hints for any_instance

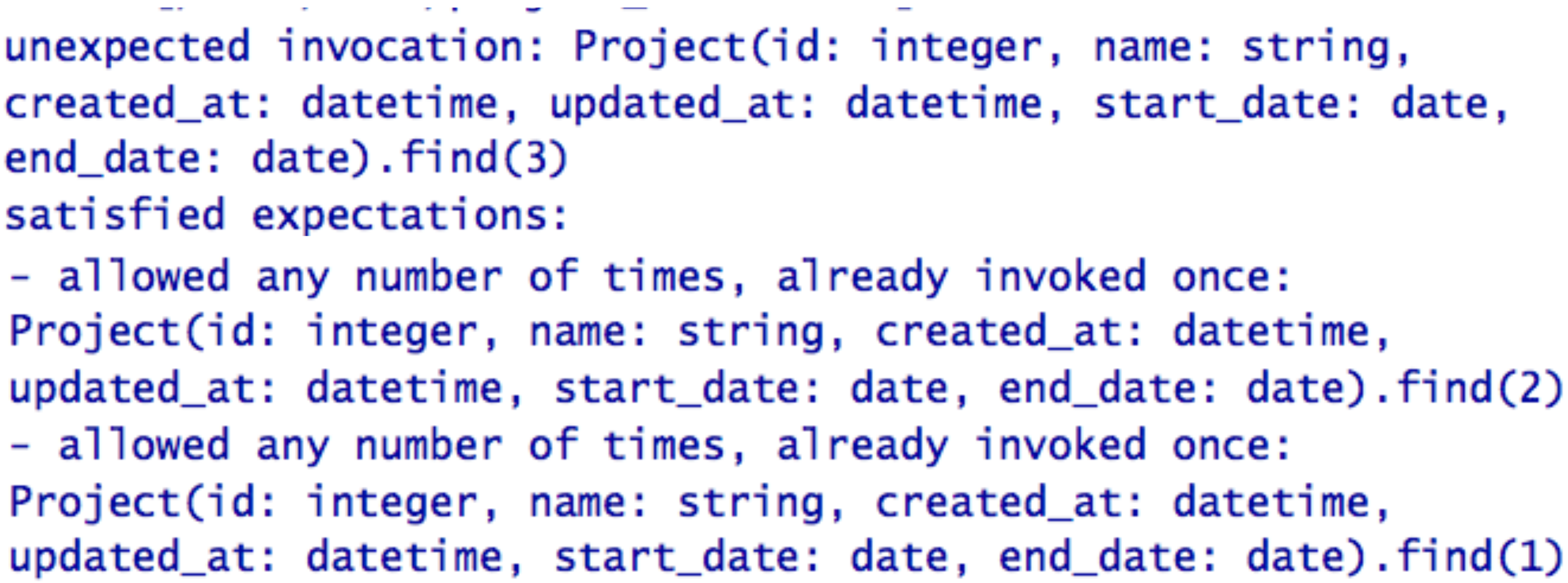
- No guarantee that find returns the exact object you expect
- any_instance is valid only for instances created after you declared the stub (not for fixture data)

Stubs with Parameters



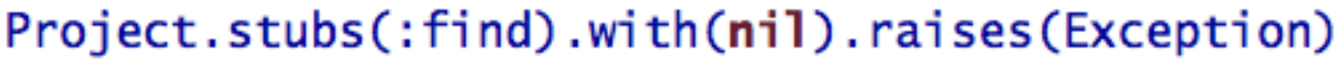
```
it "stubs a class again" do¬
  Project.stubs(:find).with(1).returns(Project.new(:name => "SWT2"))
¬
  Project.stubs(:find).with(2).returns(Project.new(:name => "TI2"))¬
  ¬
  Project.find(1).name.should equal "SWT2"¬
¬
  Project.find(2).name.should equal "TI2"¬
¬
  Project.find(3).should be_nil¬
end¬
```

Stubs with Parameters

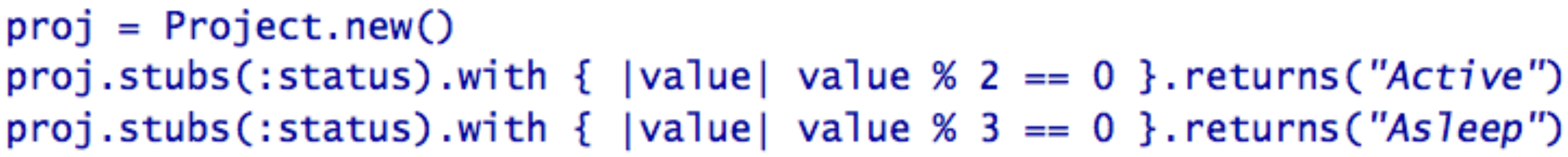


```
unexpected invocation: Project(id: integer, name: string,
created_at: datetime, updated_at: datetime, start_date: date,
end_date: date).find(3)
satisfied expectations:
- allowed any number of times, already invoked once:
Project(id: integer, name: string, created_at: datetime,
updated_at: datetime, start_date: date, end_date: date).find(2)
- allowed any number of times, already invoked once:
Project(id: integer, name: string, created_at: datetime,
updated_at: datetime, start_date: date, end_date: date).find(1)
```

with() Descriptor

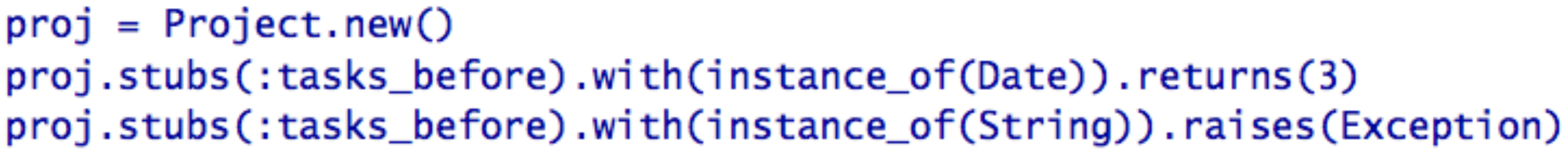


```ruby
Project.stubs(:find).with(nil).raises(Exception)
```

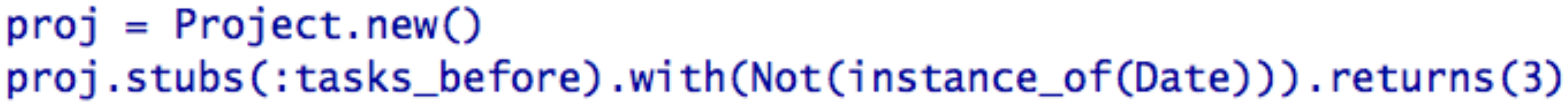


```ruby
proj = Project.new()
proj.stubs(:status).with { |value| value % 2 == 0 }.returns("Active")
proj.stubs(:status).with { |value| value % 3 == 0 }.returns("Asleep")
```

instance_of(), Not

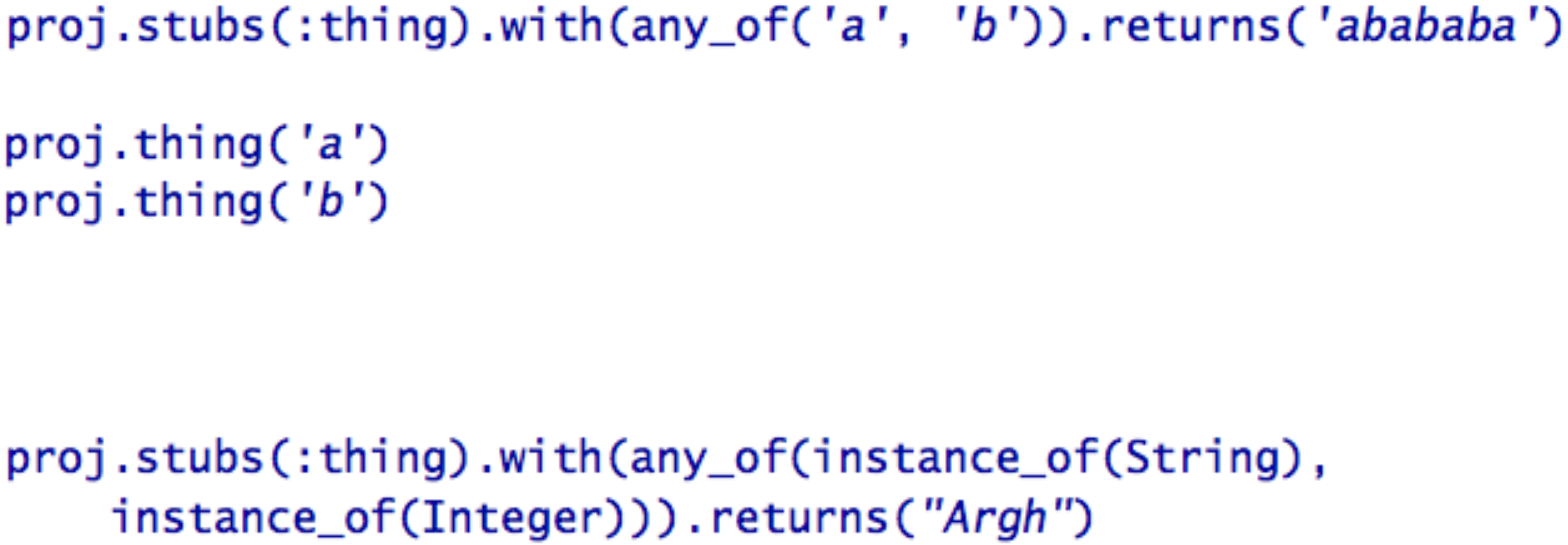


```
proj = Project.new()
proj.stubs(:tasks_before).with(instance_of(Date)).returns(3)
proj.stubs(:tasks_before).with(instance_of(String)).raises(Exception)
```



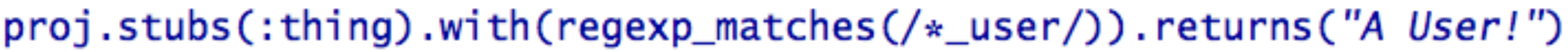
```
proj = Project.new()
proj.stubs(:tasks_before).with(Not(instance_of(Date))).returns(3)
```

any_of()



```
proj.stubs(:thing).with(any_of('a', 'b')).returns('abababa')

proj.thing('a')
proj.thing('b')



proj.stubs(:thing).with(any_of(instance_of(String),
    instance_of(Integer))).returns("Argh")
```

regexp_matches(), …



```
proj.stubs(:thing).with(regexp_matches(/*_user/)).returns("A User!")
```

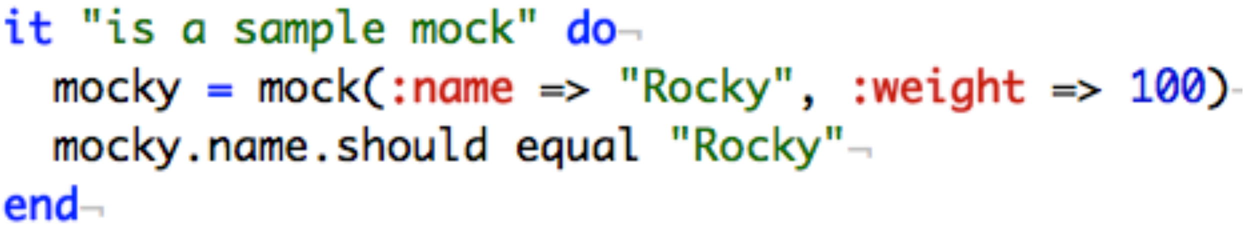
http://mocha.rubyforge.org/

Agenda

- Why Behavior-driven Design (BDD)?
- Building Blocks of Tests and BDD
  - Test Data
  - **Test Doubles**
    - Introduction
    - Stubs in Detail
    - **Mocks in Detail**
  - Setup and Teardown
  - Model Tests
  - View Tests
  - Controller Tests
  - …
- Testing Tests
- Outlook

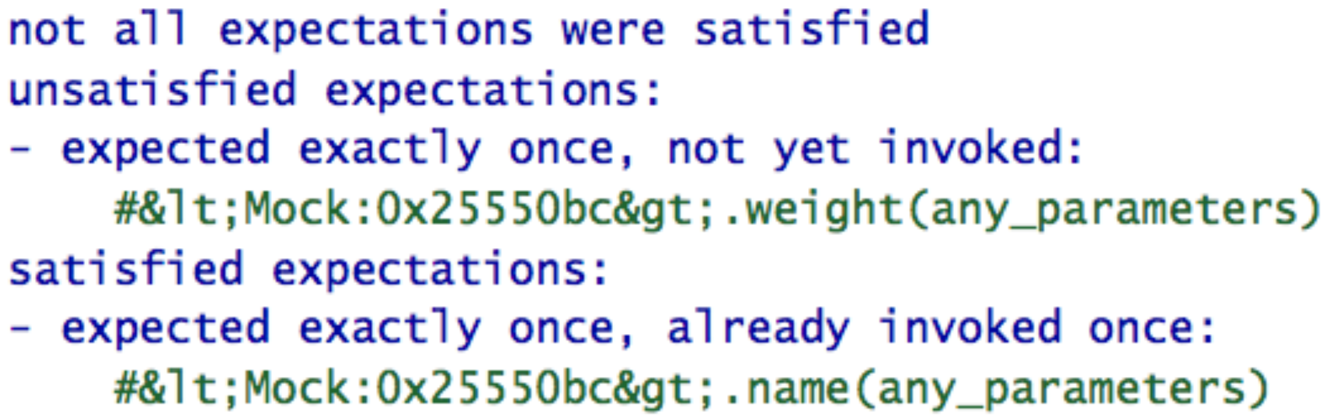
Mocks

- Mock = Stub + attitude
- Demands that mock parameters are called



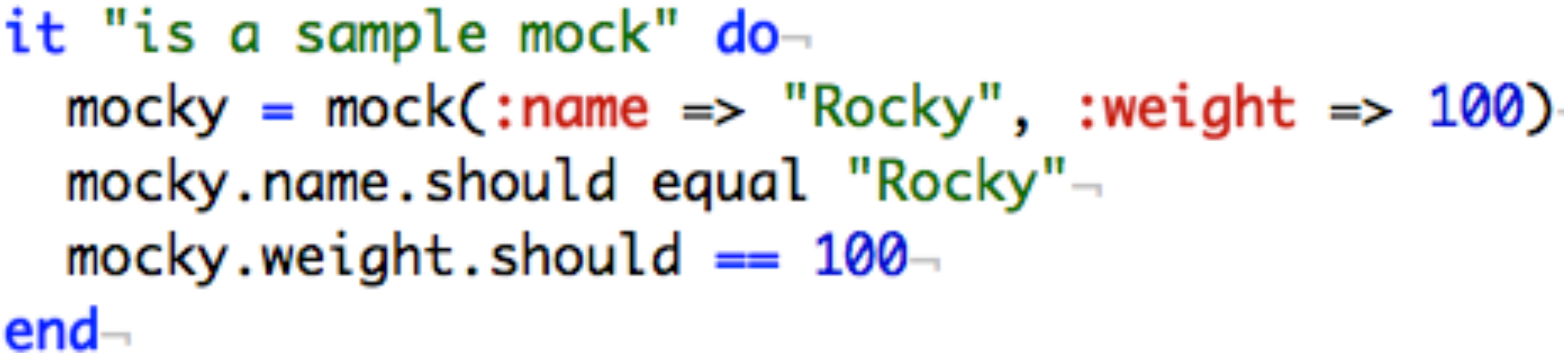
```ruby
it "is a sample mock" do
  mocky = mock(:name => "Rocky", :weight => 100)
  mocky.name.should equal "Rocky"
end
```

Mocks



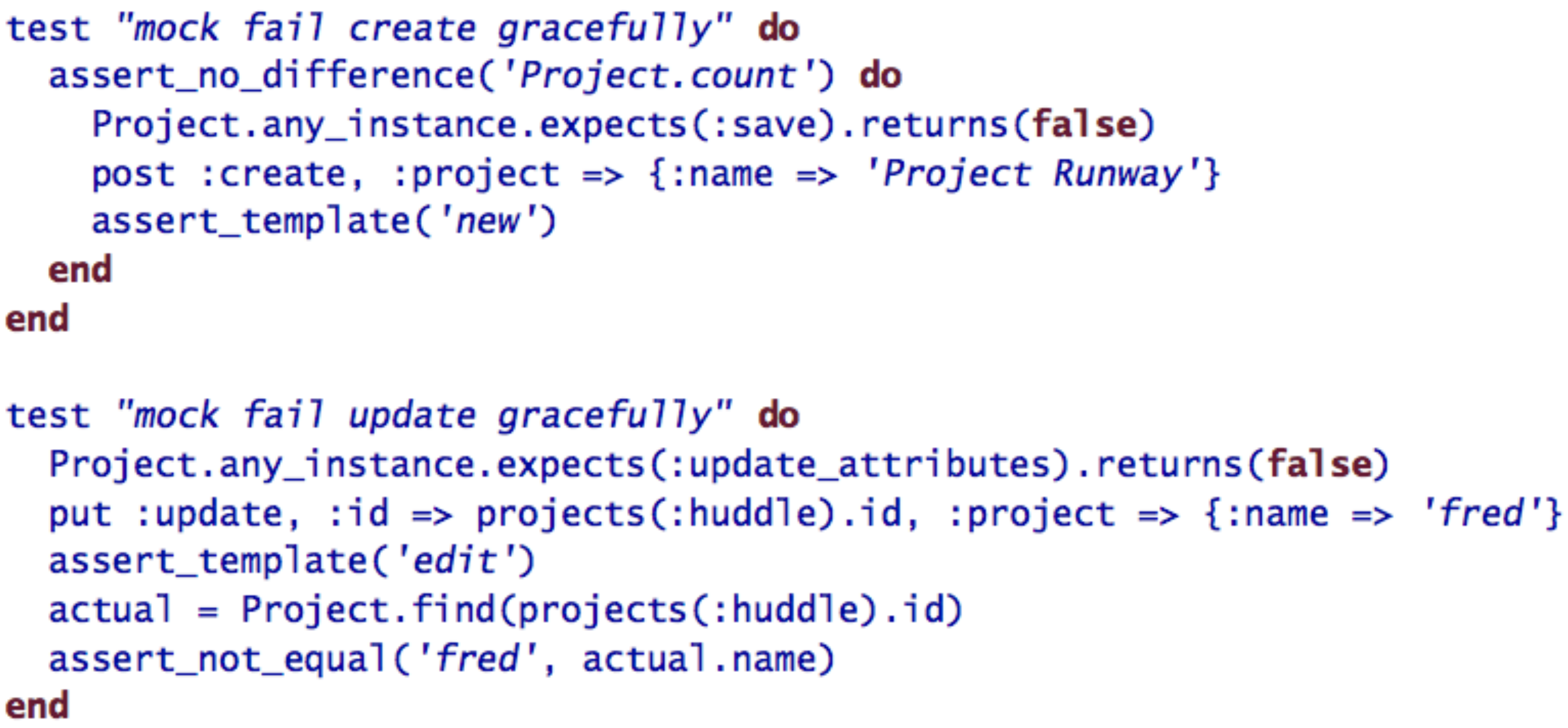
```
not all expectations were satisfied
unsatisfied expectations:
- expected exactly once, not yet invoked:
    #&lt;Mock:0x25550bc&gt;.weight(any_parameters)
satisfied expectations:
- expected exactly once, already invoked once:
    #&lt;Mock:0x25550bc&gt;.name(any_parameters)
```

Mocks



```ruby
it "is a sample mock" do
  mocky = mock(:name => "Rocky", :weight => 100)
  mocky.name.should equal "Rocky"
  mocky.weight.should == 100
end
```

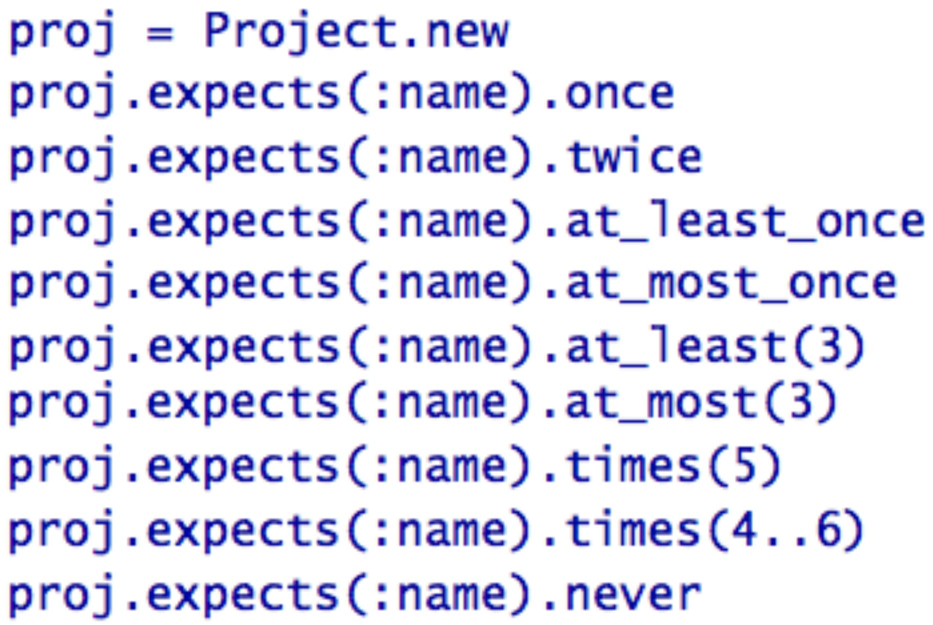
Stub Modifiers are Valid for Mocks



```
test "mock fail create gracefully" do
  assert_no_difference('Project.count') do
    Project.any_instance.expects(:save).returns(false)
    post :create, :project => {:name => 'Project Runway'}
    assert_template('new')
  end
end

test "mock fail update gracefully" do
  Project.any_instance.expects(:update_attributes).returns(false)
  put :update, :id => projects(:huddle).id, :project => {:name => 'fred'}
  assert_template('edit')
  actual = Project.find(projects(:huddle).id)
  assert_not_equal('fred', actual.name)
end
```

How often wants a Mock to be Called?

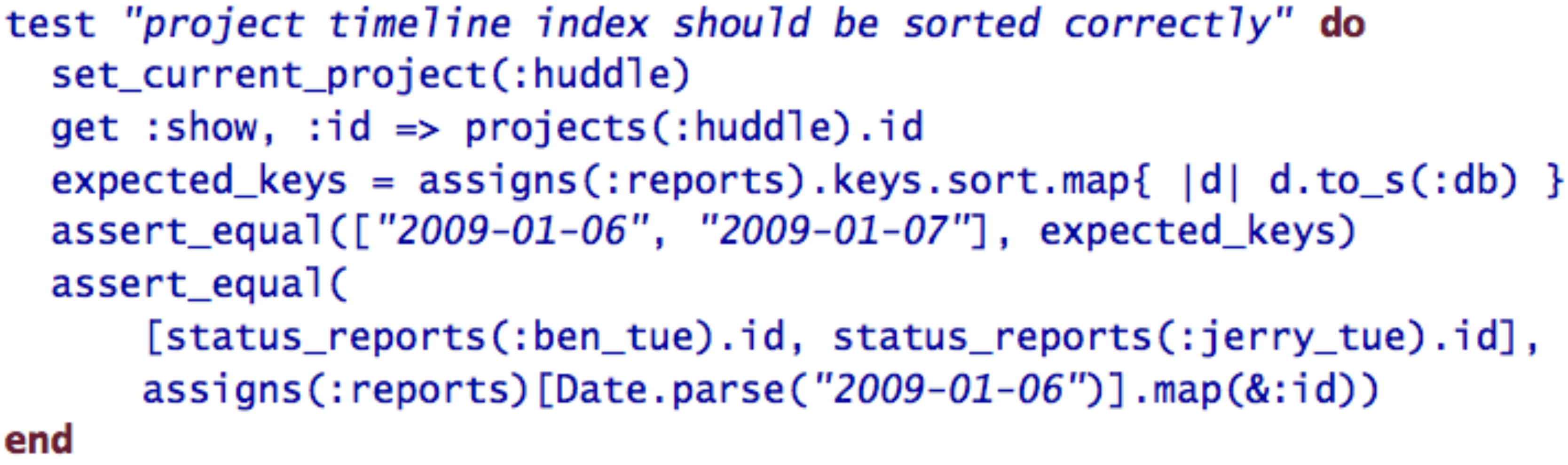
- By default: one



```
proj = Project.new
proj.expects(:name).once
proj.expects(:name).twice
proj.expects(:name).at_least_once
proj.expects(:name).at_most_once
proj.expects(:name).at_least(3)
proj.expects(:name).at_most(3)
proj.expects(:name).times(5)
proj.expects(:name).times(4..6)
proj.expects(:name).never
```

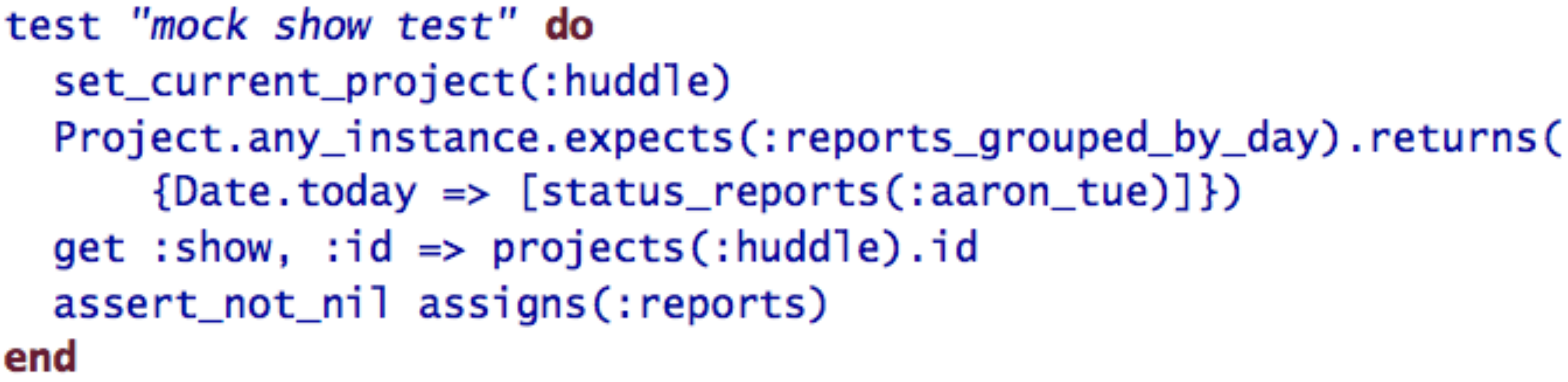
Mock Objects and Behavior-Driven Development

- Example of a controller test



```
test "project timeline index should be sorted correctly" do
  set_current_project(:huddle)
  get :show, :id => projects(:huddle).id
  expected_keys = assigns(:reports).keys.sort.map{ |d| d.to_s(:db) }
  assert_equal(["2009-01-06", "2009-01-07"], expected_keys)
  assert_equal(
      [status_reports(:ben_tue).id, status_reports(:jerry_tue).id],
      assigns(:reports)[Date.parse("2009-01-06")].map(&:id))
end
```

vs.



```
test "mock show test" do
  set_current_project(:huddle)
  Project.any_instance.expects(:reports_grouped_by_day).returns(
      {Date.today => [status_reports(:aaron_tue)]})
  get :show, :id => projects(:huddle).id
  assert_not_nil assigns(:reports)
end
```

Advantages and Disadvantages

- Disadvantages
  - Mismatch between mocked model and real model
    - Data type
    - Semantic
    - ➔ integration tests
  - Risk to test predefined data (non-sense)
  - Tests might depend on internal structures of mocked object → brittle while refactoring
- Advantages
  - The test is focused on behavior
  - Speed
  - Isolation of tests (failure in model does not affect controller test)

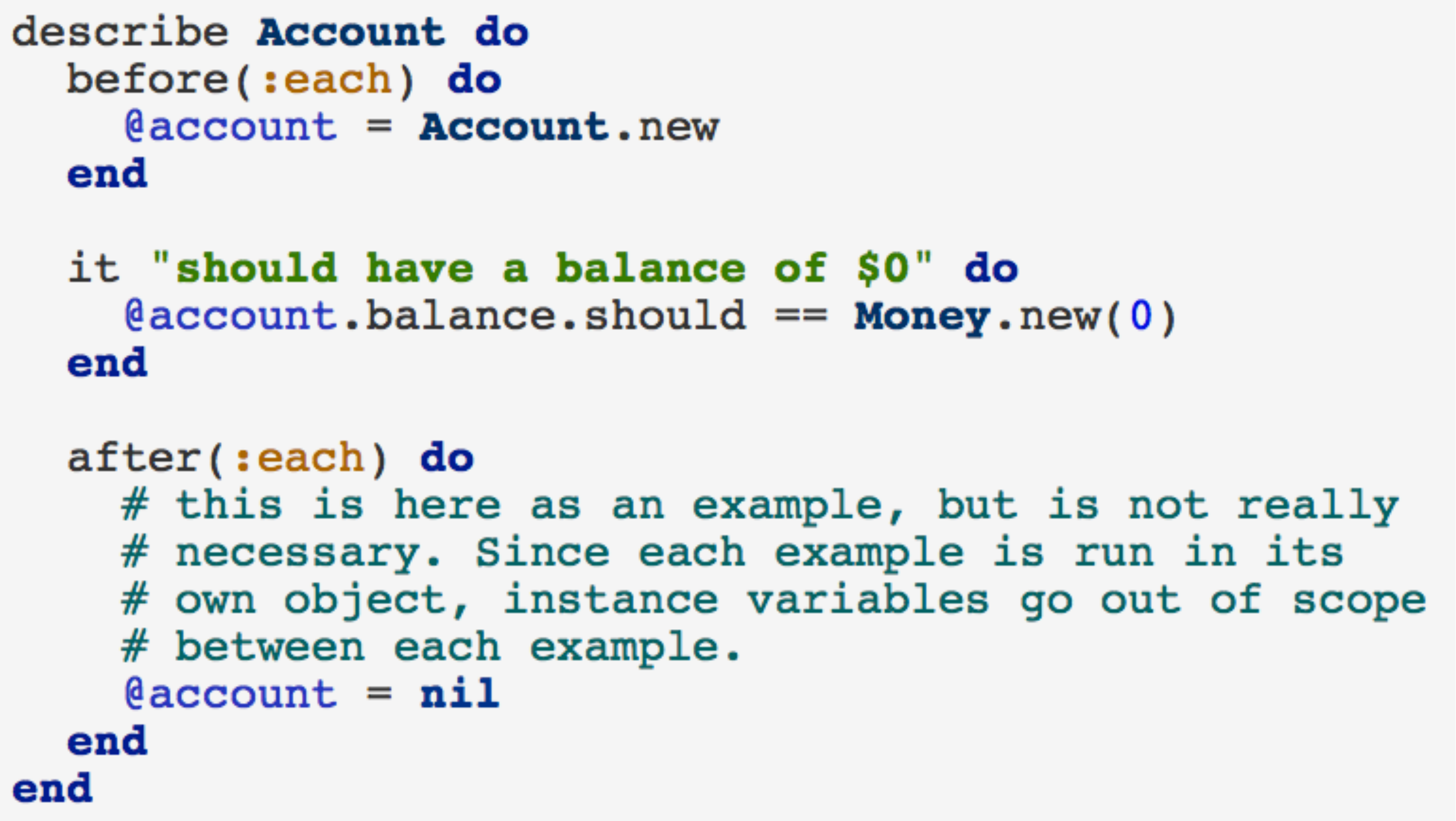
Test Double Dos & Don’ts

- You replace an object because it is hard to create in a test environment ➔ use a stub
- minimize number of mocked methods
- #mocks⇧ ➔ possibility to run out of sync with real implementation⇧
- #mocks⇧ ➔ test too large? Poor object-oriented design?
- Don’t assert a value you set by a test double (false positives)

Agenda

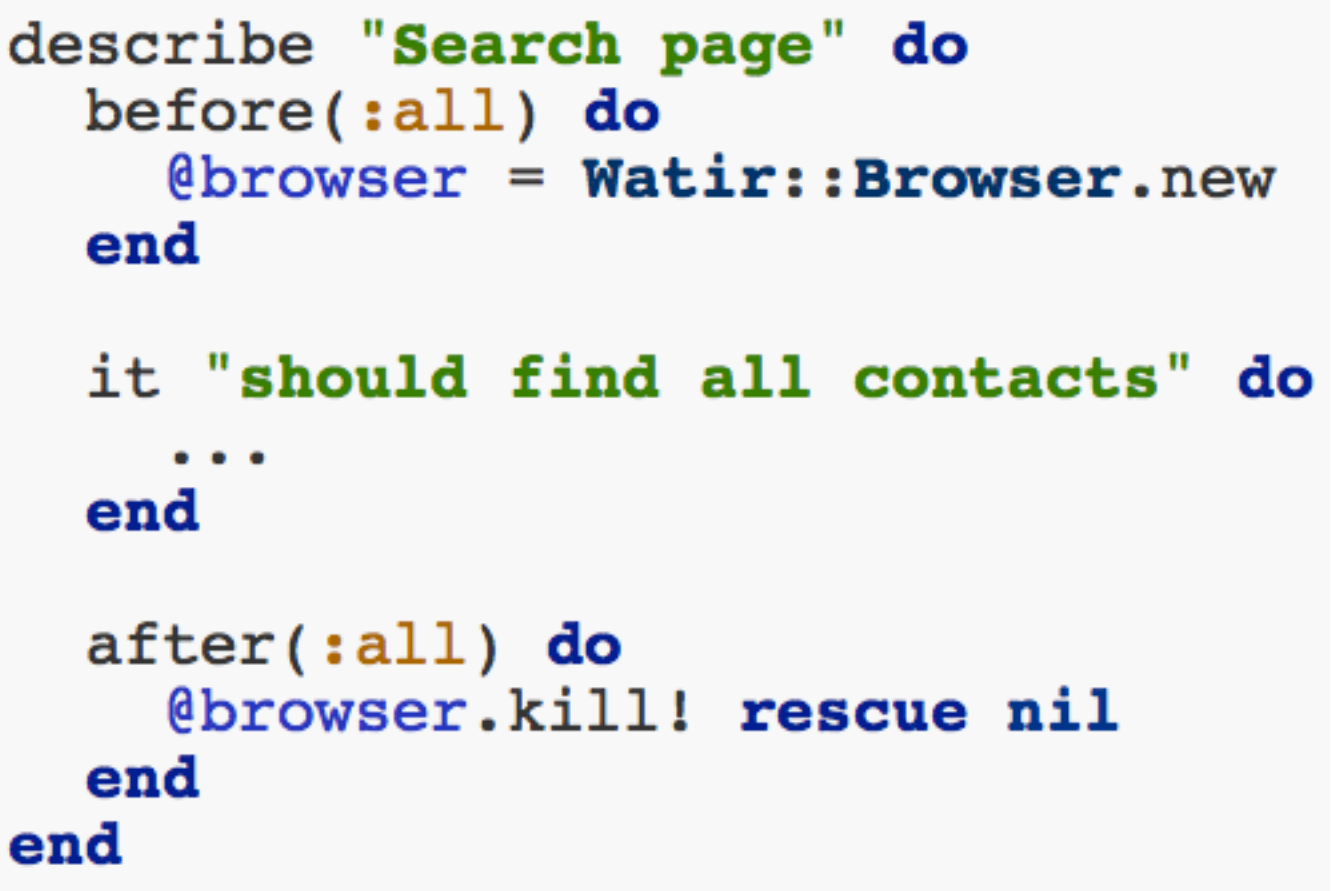
- Why Behavior-driven Design (BDD)?
- Building Blocks of Tests and BDD
  - □ Test Data
  - □ Test Doubles
  - □ **Setup and Teardown**
  - □ Model Tests
  - □ View Tests
  - □ Controller Tests
  - □ …
- Testing Tests
- Outlook

Setup and Teardown
RSpec



```ruby
describe Account do
  before(:each) do
    @account = Account.new
  end

  it "should have a balance of $0" do
    @account.balance.should == Money.new(0)
  end

  after(:each) do
    # this is here as an example, but is not really
    # necessary. Since each example is run in its
    # own object, instance variables go out of scope
    # between each example.
    @account = nil
  end
end
```

Setup and Teardown
RSpec



```
describe "Search page" do
  before(:all) do
    @browser = Watir::Browser.new
  end

  it "should find all contacts" do
    ...
  end

  after(:all) do
    @browser.kill! rescue nil
  end
end
```

Agenda

- Why Behavior-driven Design (BDD)?
- Building Blocks of Tests and BDD
  - ...
  - **Model Tests**
  - View Tests
  - Controller Tests
  - Routing Tests
  - Outgoing Mail Tests
  - Helper Tests
  - Integration and Acceptance Tests
- Testing Tests & Hints for Successful Test Design
- Outlook

Model Tests

- A Rails model
  - accesses data through an ORM
  - implements business logic
  - is “fat”

- Model tests
  - Model tests in Rails = Test::Unit + test data + setup/teardown + test logic + additional assertions
  - Easiest tests to write

Hints for Model Tests

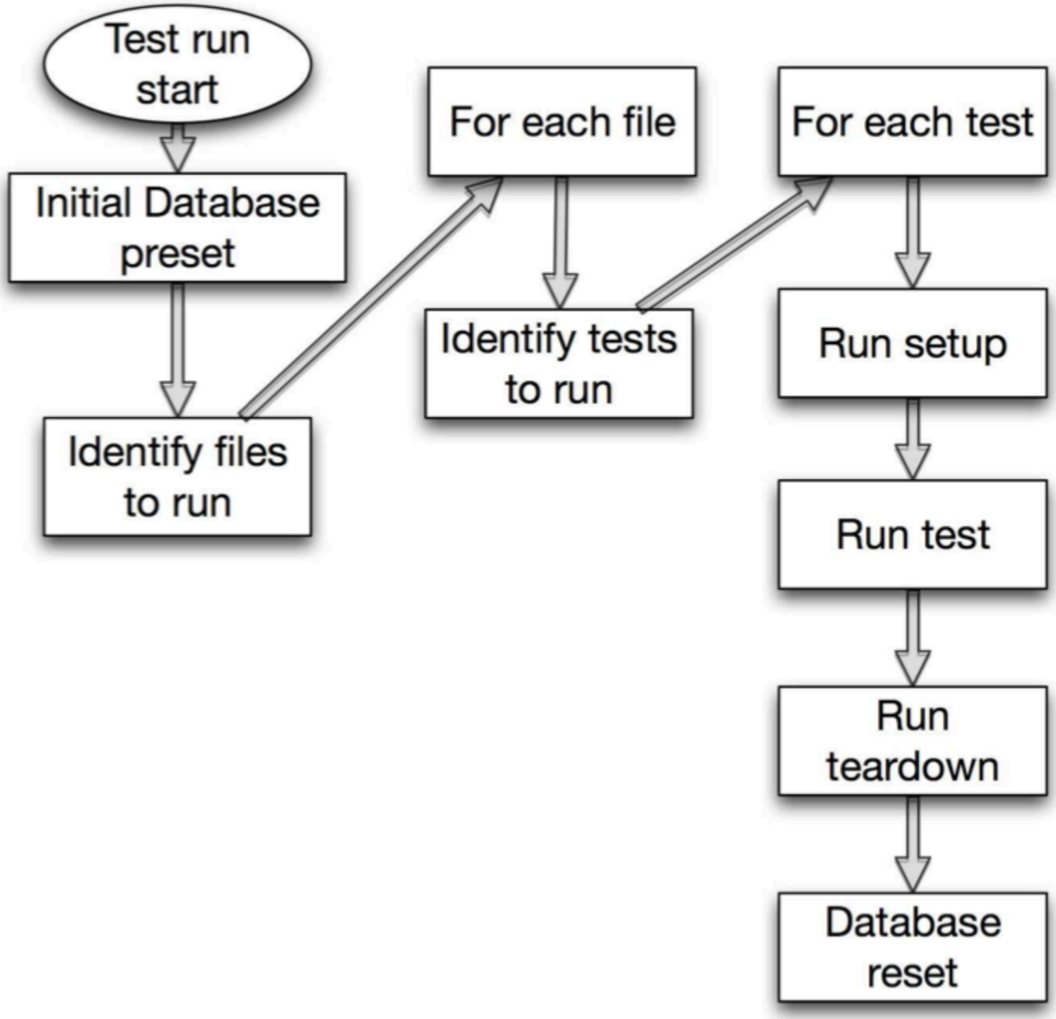
- Tests should cover ~100% of the model code
- Do not test framework functionality like “belongs_to”
- Test your validations
- How many tests? Let tests drive the code → perfect fit
- What comes out?
    - One test for the “happy-path case”
    - One test for each branch
    - Corner cases (nil, wrong values, …) ← if appropriate
- Keep each test small!

How many Assertions per Test?

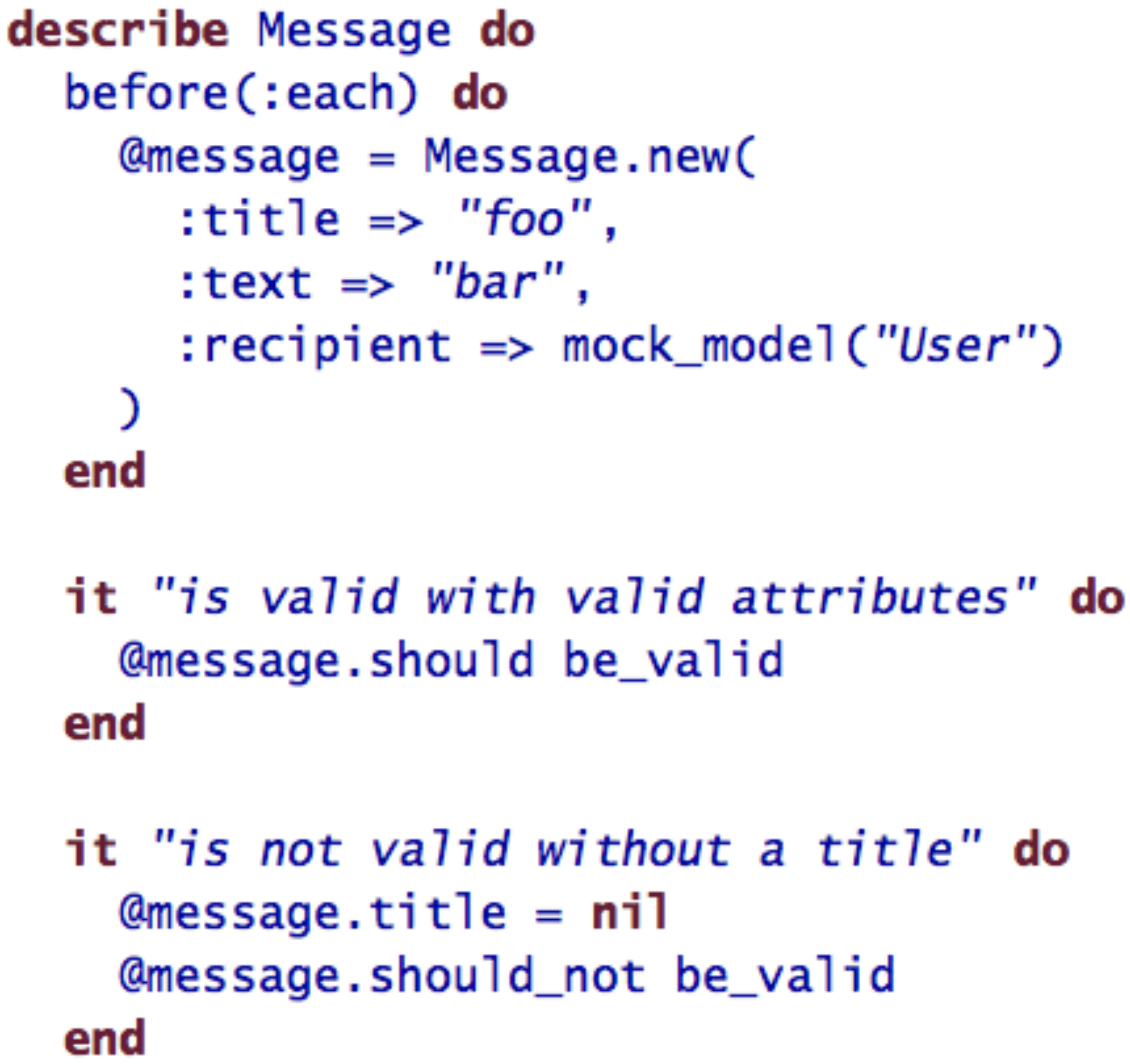
- If 1 call to a model ➔ many changes:
  - #Assertions ⇧ ➔ clarity and cohesion ⇧
  - #Assertions ⇧ ➔ test independece ⇩

  ➔ Use context & describe and have 1 assertion per test

Test Run

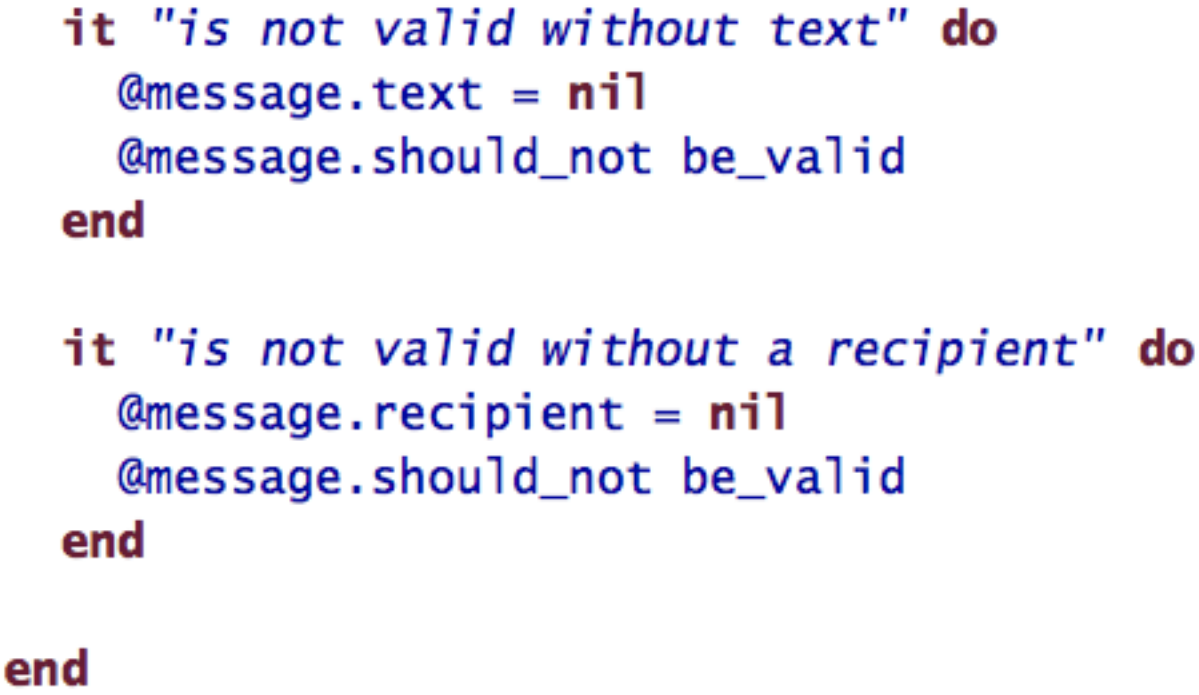

Test run start
Initial Database preset
Identify files to run
For each file
Identify tests to run
For each test
Run setup
Run test
Run teardown
Database reset

Example Rspec Tests



```
describe Message do
  before(:each) do
    @message = Message.new(
      :title => "foo",
      :text => "bar",
      :recipient => mock_model("User")
    )
  end

  it "is valid with valid attributes" do
    @message.should be_valid
  end

  it "is not valid without a title" do
    @message.title = nil
    @message.should_not be_valid
  end
```

Example Rspec Tests



```
  it "is not valid without text" do
    @message.text = nil
    @message.should_not be_valid
  end

  it "is not valid without a recipient" do
    @message.recipient = nil
    @message.should_not be_valid
  end

end
```

RSpec

- Remainder: we use RSpec

http://rspec.info/

Autotest

- Automate testing with Autotest (https://github.com/rspec/rspec/wiki/autotest)
- Run autotest –rails
- Integrate with Growl
- Use FSEvent

Agenda

- Why Behavior-driven Design (BDD)?
- Building Blocks of Tests and BDD
  - ...
  - Model Tests
  - **View Tests**
  - Controller Tests
  - Routing Tests
  - Outgoing Mail Tests
  - Helper Tests
  - Integration and Acceptance Tests
- Testing Tests & Hints for Successful Test Design
- Outlook

View Tests

- A Rails view
  - Has only minimal logic
  - Does never call the database
  - Presents the data given by the controller

- Challenges for view tests
  - Time-intensive
  - How to test look & feel?
  - Brittle w.r.t. re-designs

- What to do?

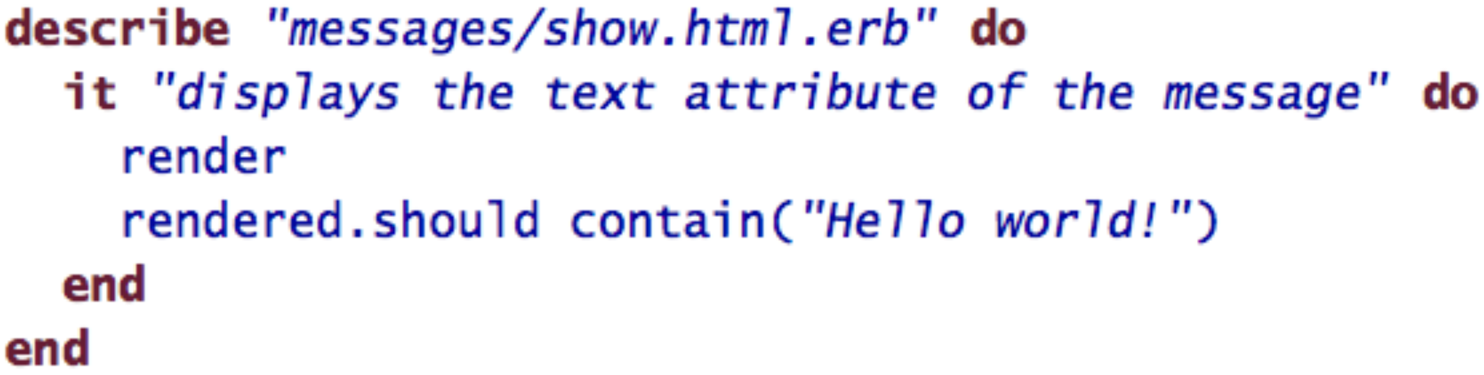
View Tests

- Specify and verify logical and semantic structure

- Goals
  - Validate that view layer runs without error
  - Data gathered by the controller is presented as expected
  - Validate security-based output (e.g., for admins)

- Do not
  - Validate HTML markup
  - Look & feel

Most Basic View Tests



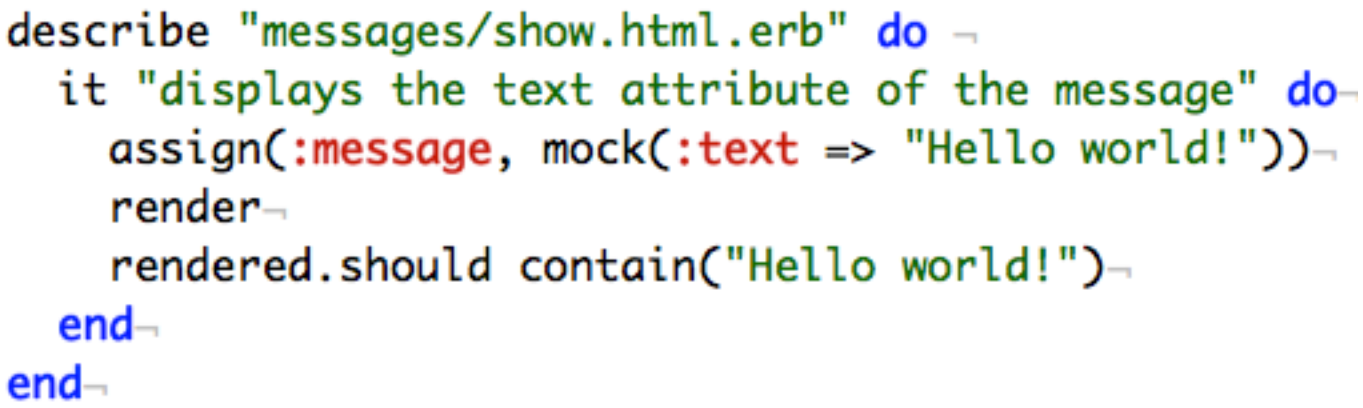
```
describe "messages/show.html.erb" do
  it "displays the text attribute of the message" do
    render
    rendered.should contain("Hello world!")
  end
end
```

Keys to Meaningful View Tests

- Test views semantically with use DOM IDs / CSS classes
- Do not test actual text

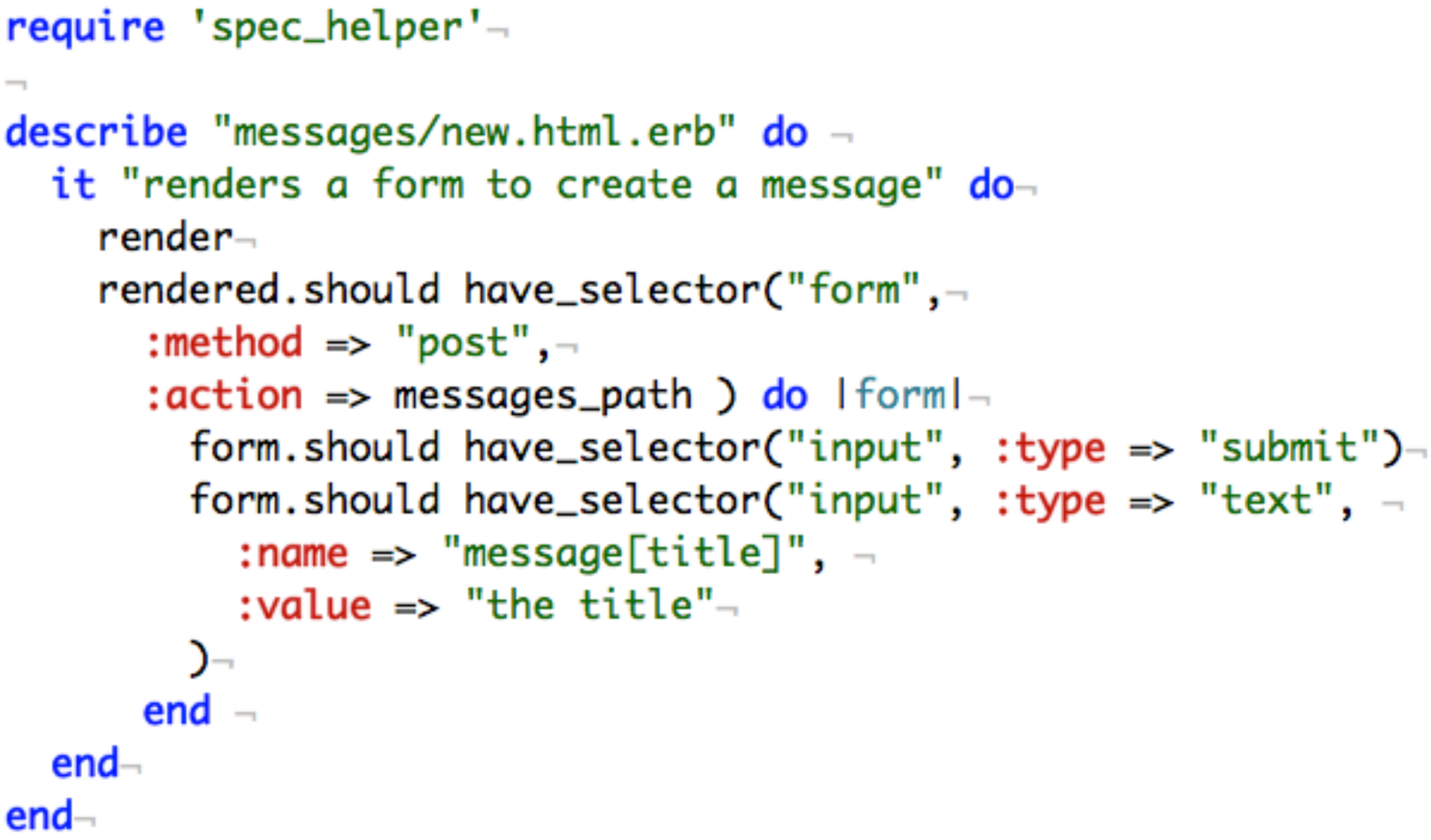
- Side-effect: you validate your HTML

assign()



```
describe "messages/show.html.erb" do
  it "displays the text attribute of the message" do
    assign(:message, mock(:text => "Hello world!"))
    render
    rendered.should contain("Hello world!")
  end
end
```

should have_selector()



```ruby
require 'spec_helper'

describe "messages/new.html.erb" do
  it "renders a form to create a message" do
    render
    rendered.should have_selector("form",
      :method => "post",
      :action => messages_path ) do |form|
        form.should have_selector("input", :type => "submit")
        form.should have_selector("input", :type => "text",
          :name => "message[title]",
          :value => "the title"
        )
      end
  end
end
```

Agenda

- Why Behavior-driven Design (BDD)?
- Building Blocks of Tests and BDD
  - ...
  - Model Tests
  - View Tests
  - **Controller Tests**
  - Routing Tests
  - Outgoing Mail Tests
  - Helper Tests
  - Integration and Acceptance Tests
- Testing Tests & Hints for Successful Test Design
- Outlook

Controller Tests

- A Rails controller
  - Is “skinny”
  - Calls the ORM
  - Calls the model
  - Passes data to the view

- Goal of controller tests
  - Simulate a request
  - Verify the result

- Subclass of ActionController::TestCase (http://api.rubyonrails.org/classes/ActionController/TestCase.html)
- and ActiveSupport:TestCase (http://api.rubyonrails.org/classes/ActiveSupport/TestCase.html)

Controller Tests

- 3 important variables
  - controller
  - request
  - response

- Variables for
  - session – session[:key]
  - controller variables – assigns[:key]
  - flash – flash[:key]

Controller Tests

- Methods for
  - get
  - post
  - put
  - delete
  - xhr (Ajax)

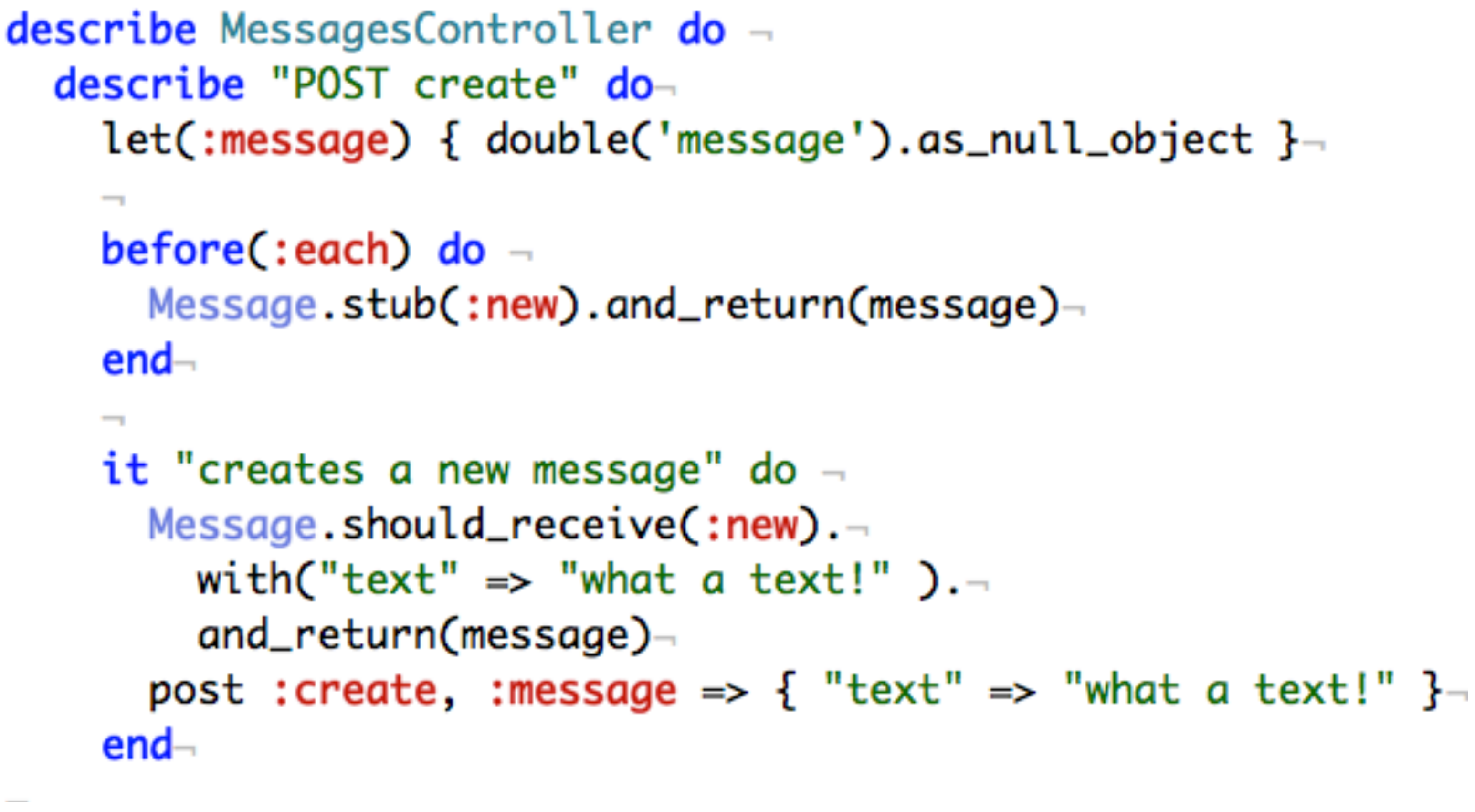
What to test?

- Remember: model functionality is tested in model tests
- Controller tests
  - Verify that user requests trigger
    - model/OER calls
    - that data is forwarded to view
  - Handling of invalid user requests
  - Verifying security roles / role-based access control

Structure of a Controller Test

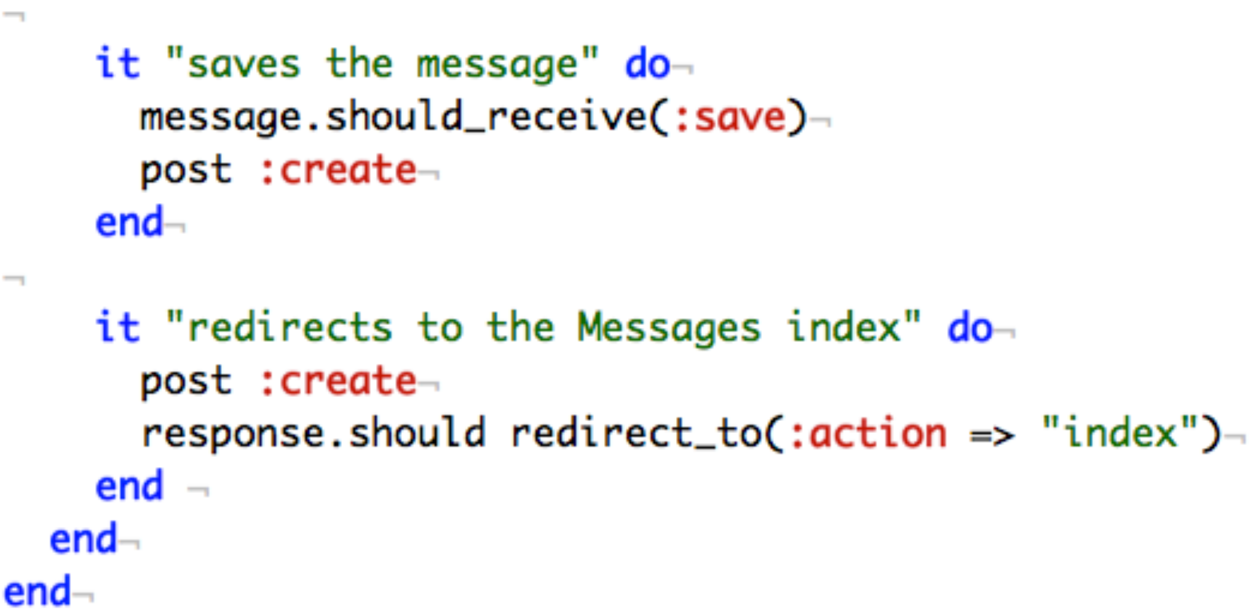
- Setup
    - Prepare data
    - Prepare request/session
- Send request to controller
- Validate controller response

Simple Controller Test



```
describe MessagesController do
  describe "POST create" do
    let(:message) { double('message').as_null_object }

    before(:each) do
      Message.stub(:new).and_return(message)
    end

    it "creates a new message" do
      Message.should_receive(:new).
        with("text" => "what a text!" ).
        and_return(message)
      post :create, :message => { "text" => "what a text!" }
    end
```

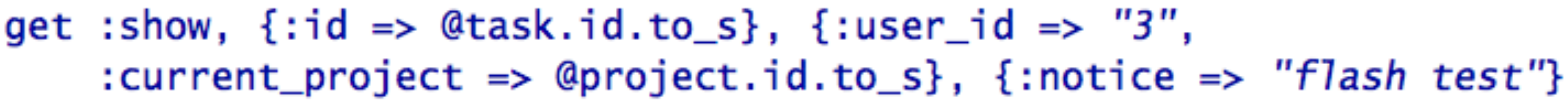
Simple Controller Test



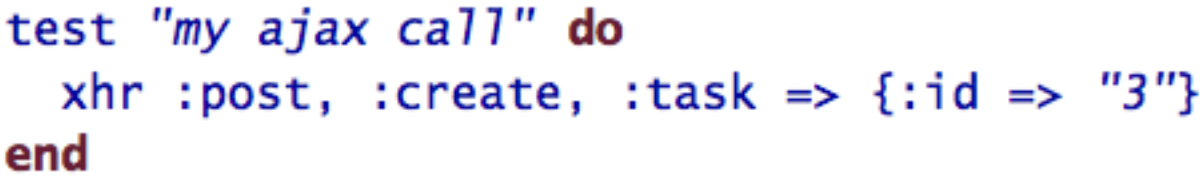
```

    it "saves the message" do
      message.should_receive(:save)
      post :create
    end

    it "redirects to the Messages index" do
      post :create
      response.should redirect_to(:action => "index")
    end
  end
end
```

Structure of a Request

<http_verb> :<method>, <parameters>, <session_data>, <text_flash>



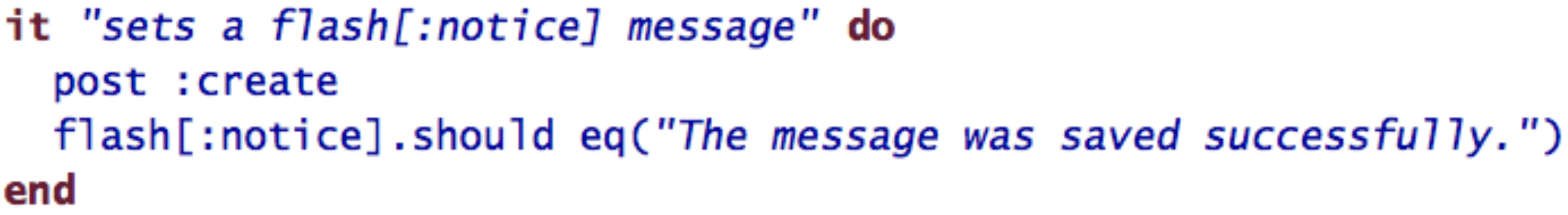
```
get :show, {:id => @task.id.to_s}, {:user_id => "3",
    :current_project => @project.id.to_s}, {:notice => "flash test"}
```



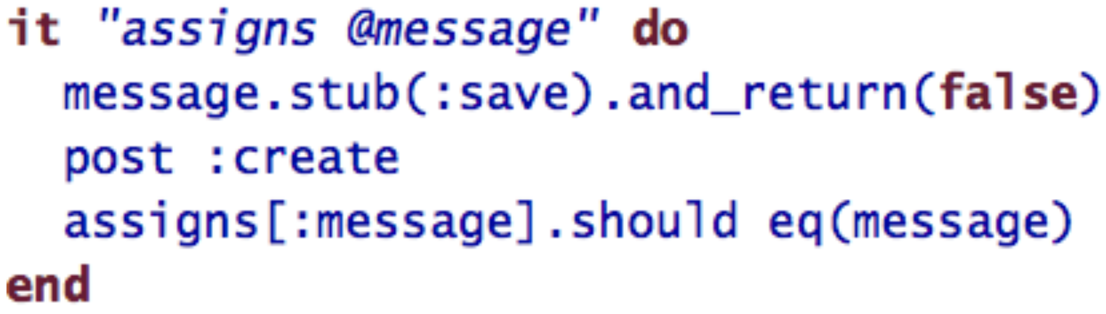
```
test "my ajax call" do
  xhr :post, :create, :task => {:id => "3"}
end
```

format.js

Some More Examples

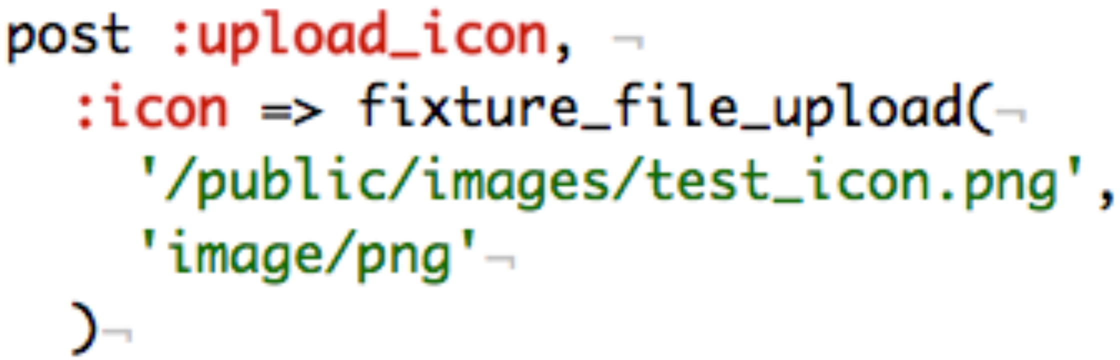


```ruby
it "sets a flash[:notice] message" do
  post :create
  flash[:notice].should eq("The message was saved successfully.")
end
```



```ruby
it "assigns @message" do
  message.stub(:save).and_return(false)
  post :create
  assigns[:message].should eq(message)
end
```

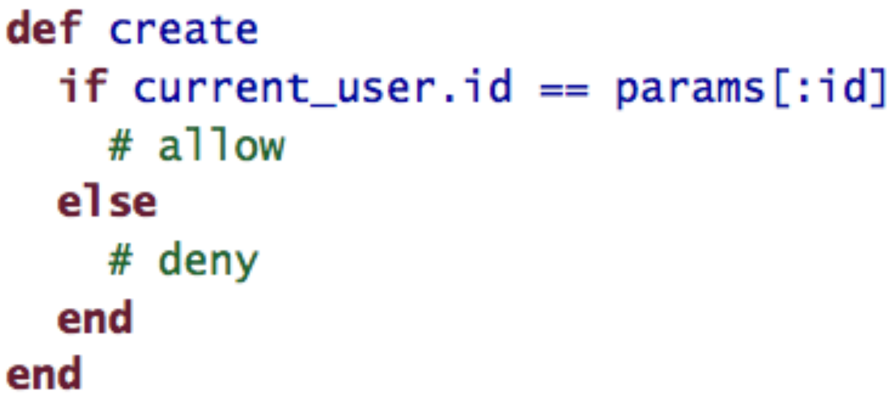
Testing File Uploads



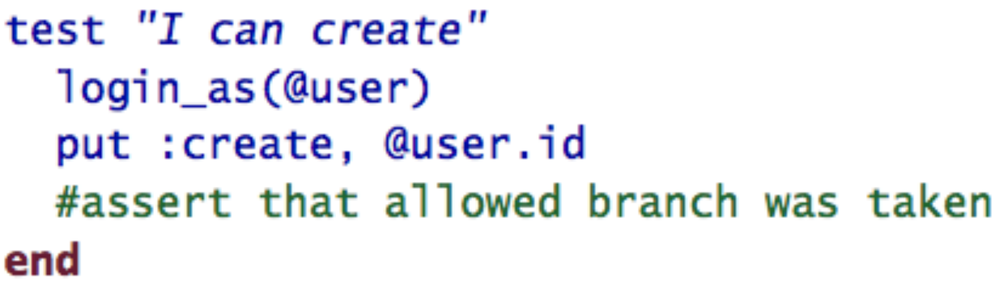
```
post :upload_icon, ¬
  :icon => fixture_file_upload(¬
    '/public/images/test_icon.png',
    'image/png'¬
  )¬
```

Background on Controller Tests

- Controller method is called directly
- Routes are NOT evaluated
- Real request parameters are always strings



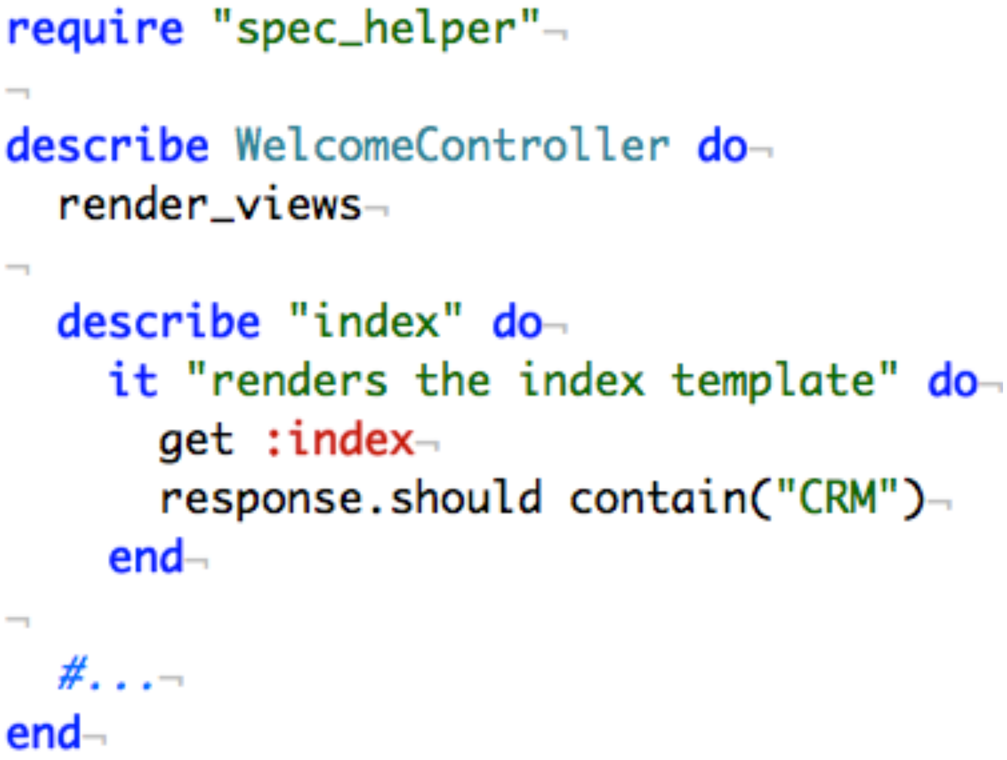
```
def create
  if current_user.id == params[:id]
    # allow
  else
    # deny
  end
end
```



```
test "I can create"
  login_as(@user)
  put :create, @user.id
  #assert that allowed branch was taken
end
```

Background on Controller Tests

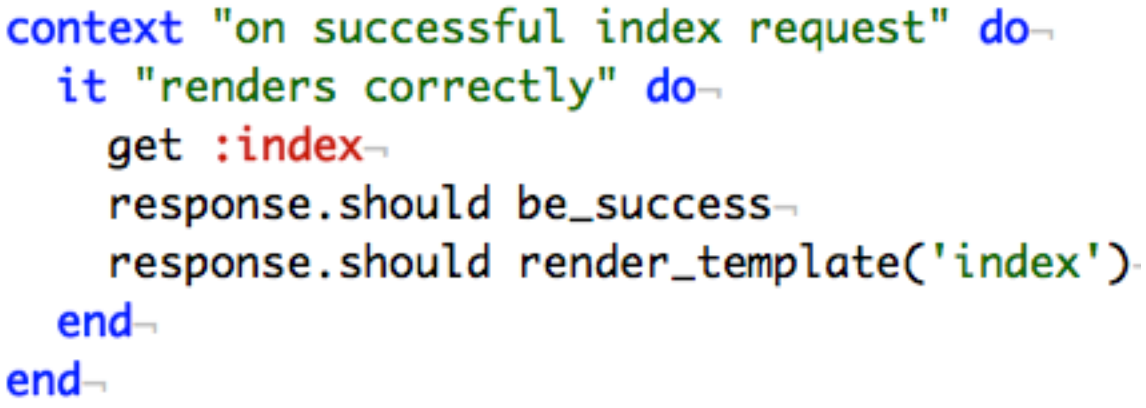
- By default, views are not rendered



```ruby
require "spec_helper"

describe WelcomeController do
  render_views

  describe "index" do
    it "renders the index template" do
      get :index
      response.should contain("CRM")
    end

  #...
end
```

Testing the Controller Response

- HTTP status code
- Correct template
- Assertion methods
    - □ response.should redirect_to(…)
    - □ response.should be_success | be_redirect | …
    - □ response.should render_template(…)



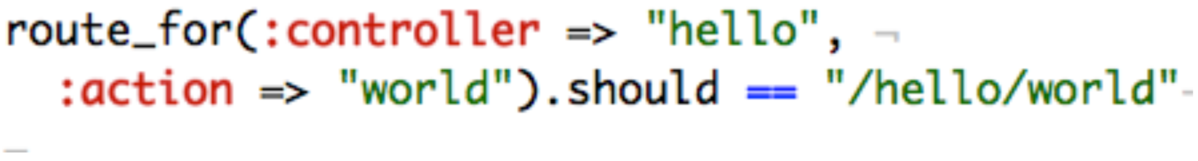
```
context "on successful index request" do
  it "renders correctly" do
    get :index
    response.should be_success
    response.should render_template('index')
  end
end
```

Agenda

- Why Behavior-driven Design (BDD)?
- Building Blocks of Tests and BDD
  - ...
  - Model Tests
  - View Tests
  - Controller Tests
  - **Routing Tests**
  - Outgoing Mail Tests
  - Helper Tests
  - Integration and Acceptance Tests
- Testing Tests & Hints for Successful Test Design
- Outlook

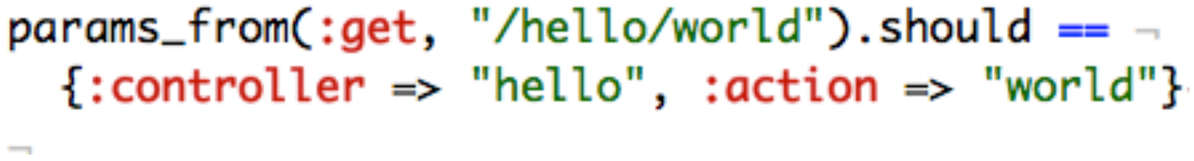
Route Tests

- route_for



```
route_for(:controller => "hello", 
  :action => "world").should == "/hello/world"
```

- params_from



```
params_from(:get, "/hello/world").should == 
  {:controller => "hello", :action => "world"}
```

Agenda

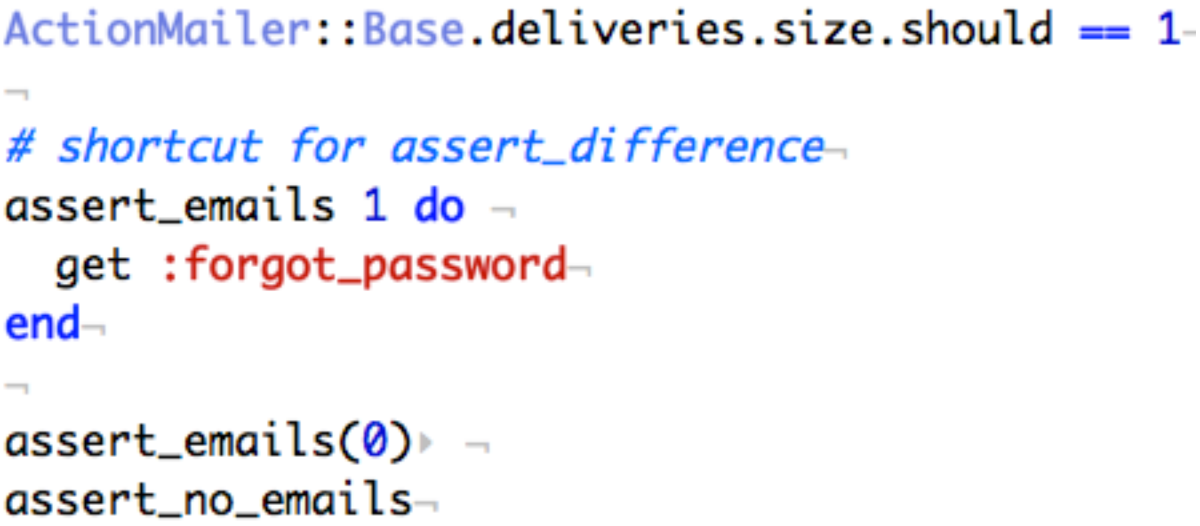
- Why Behavior-driven Design (BDD)?
- Building Blocks of Tests and BDD
  - …
  - Model Tests
  - View Tests
  - Controller Tests
  - Routing Tests
  - **Outgoing Mail Tests**
  - Helper Tests
  - Integration and Acceptance Tests
- Testing Tests & Hints for Successful Test Design
- Outlook

Outgoing Mail Tests

- What to validate?
  - Application sends mail when expected
  - Email content is what you expect

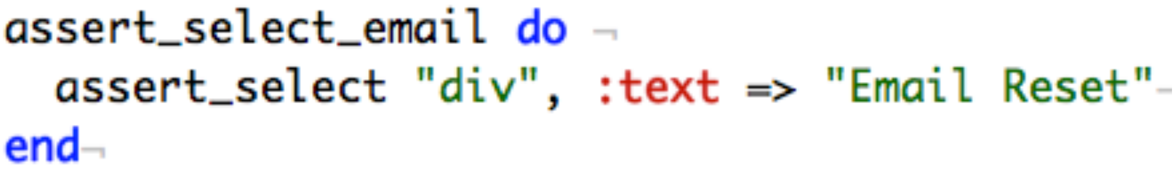
- Enable testing
  - Uncomment line 26 in config/environments/test.rb
  - config.action_mailer.delivery_method = :test
  - In mail test setup: "ActionMailer::Base.deliveries.clear"

First Steps in Outgoing Mail Tests



```
ActionMailer::Base.deliveries.size.should == 1

# shortcut for assert_difference
assert_emails 1 do
  get :forgot_password
end

assert_emails(0)
assert_no_emails
```

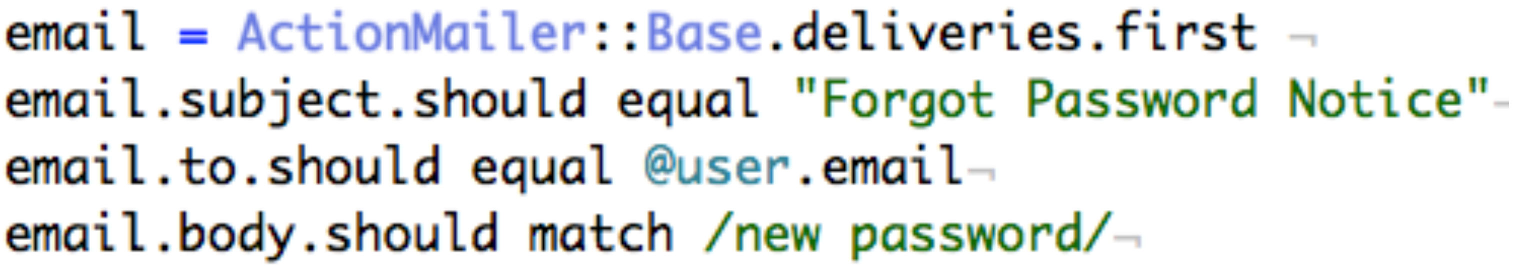
Validating Content



```
assert_select_email do
  assert_select "div", :text => "Email Reset"
end
```

- Applied to the body of each email in ActionMailer::Base.deliveries
- Only text/html emails

Complete Outgoing Mail Test

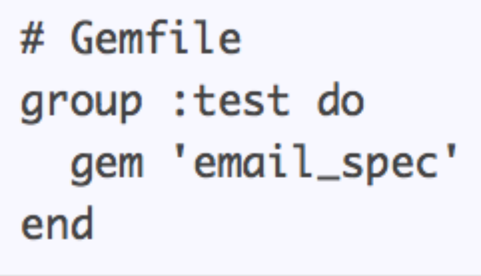


```
email = ActionMailer::Base.deliveries.first
email.subject.should equal "Forgot Password Notice"
email.to.should equal @user.email
email.body.should match /new password/
```

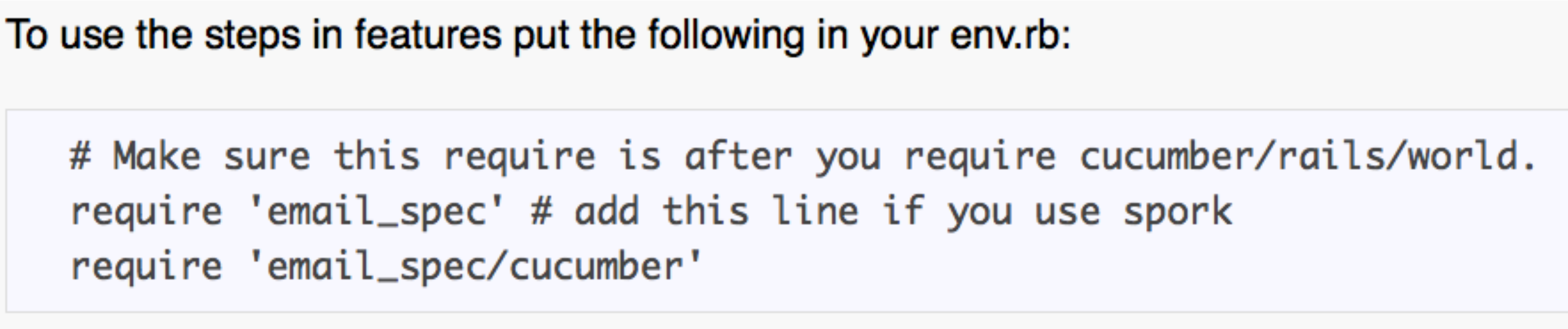
More insights about mail spec at http://tmail.rubyforge.org/

Mail Tests in RSpec & Cucumber

- email-spec (https://github.com/bmabey/email-spec)
- Installation

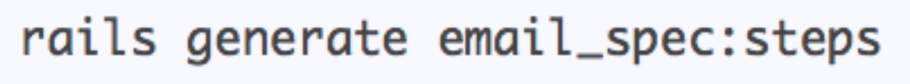


```
# Gemfile
group :test do
  gem 'email_spec'
end
```



To use the steps in features put the following in your env.rb:

```
  # Make sure this require is after you require cucumber/rails/world.
  require 'email_spec' # add this line if you use spork
  require 'email_spec/cucumber'
```



```
rails generate email_spec:steps
```

Cucumber Mail Scenarios

- Clear the email queue (done automatically by email_spec)

- Execute steps that sends an email

- Check the user received an/no/[0-9] emails
- Open the email
- Inspect the email contents
- Interact with the email (e.g. click links)

Cucumber Mail Scenarios

- Given a clear email queue | no emails have been sent

- When ...

- Then I|they|address should receive an|no|\d+ emails
- Then I|they|address should have an|no|\d+ emails
- Then I|they|address should receive an|no|\d+ emails with subject ”subject”
- Then I|they|address should receive an email with the following body:

Cucumber Mail Scenarios

- When I|they|address opens the email ← last recent one
- When I|they|address opens the email with subject ”subject”
- When I|they|address opens the email with text ”text”

- Then I|they should see ”text" in the email subject
- Then I|they should see \regex/ in the email subject

- Then I|they should see ”text" in the email body
- Then I|they should see \regex/ in the email body

Cucumber Mail Scenarios

- Then I|they should see the email delivered from ”text”
- Then I|they should see ”header_text" in the email ”header_name
- Then I|they should see \regex/ in the email ”header_name" header

- Then I should see it is a multi-part email
- Then I|they should see ”text" in the email html part body
- Then I|they should see ”text" in the email text part body

Cucumber Mail Scenarios

- Then I|they should see an|no|\d+ attachments with the email
- Then there should be an|no|\d+ attachments named ”filename”
- Then attachment \d+ should be named ”filename”
- Then there should be an|no|\d+ attachments of type ”content_type
- Then attachment (\d+) should be of type ”content_type”

- Then all attachments should not be blank
- Then show me a list of email attachments

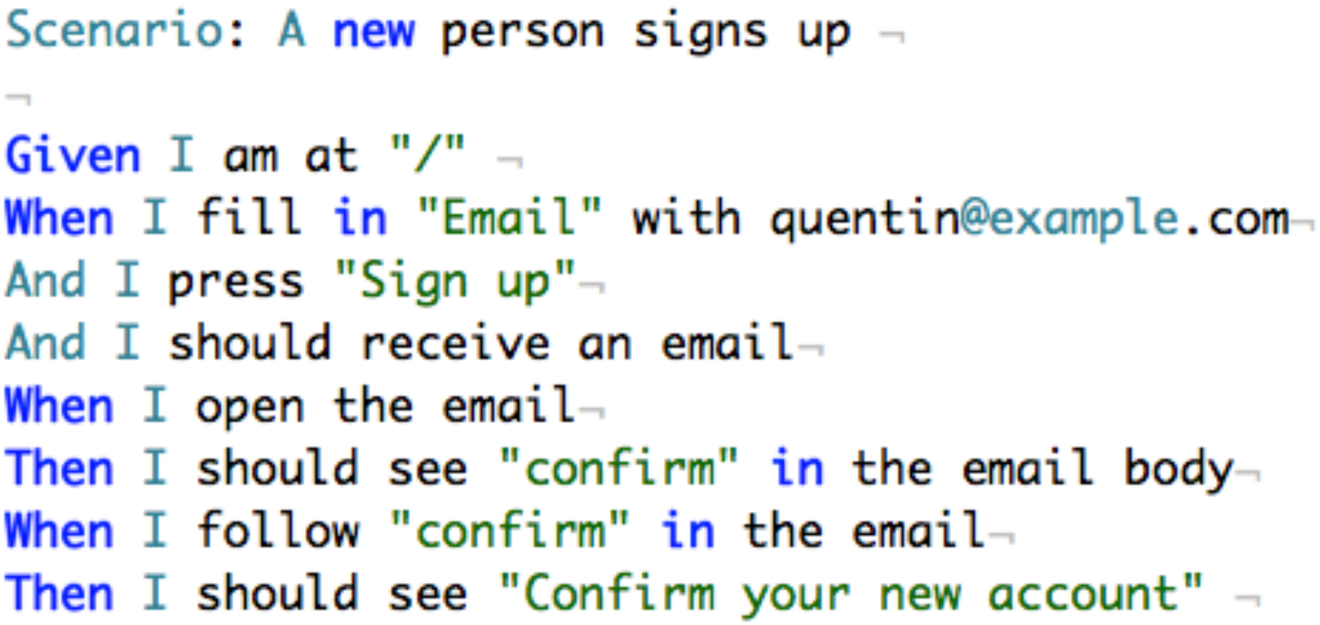
Cucumber Mail Scenarios

- When I|they follow ”link" in the email
- When I|they click the first link in the email

- # Debugging, Rails and OSx ATM since EmailViewer uses RAILS_ROOT and OSx's 'open' command.

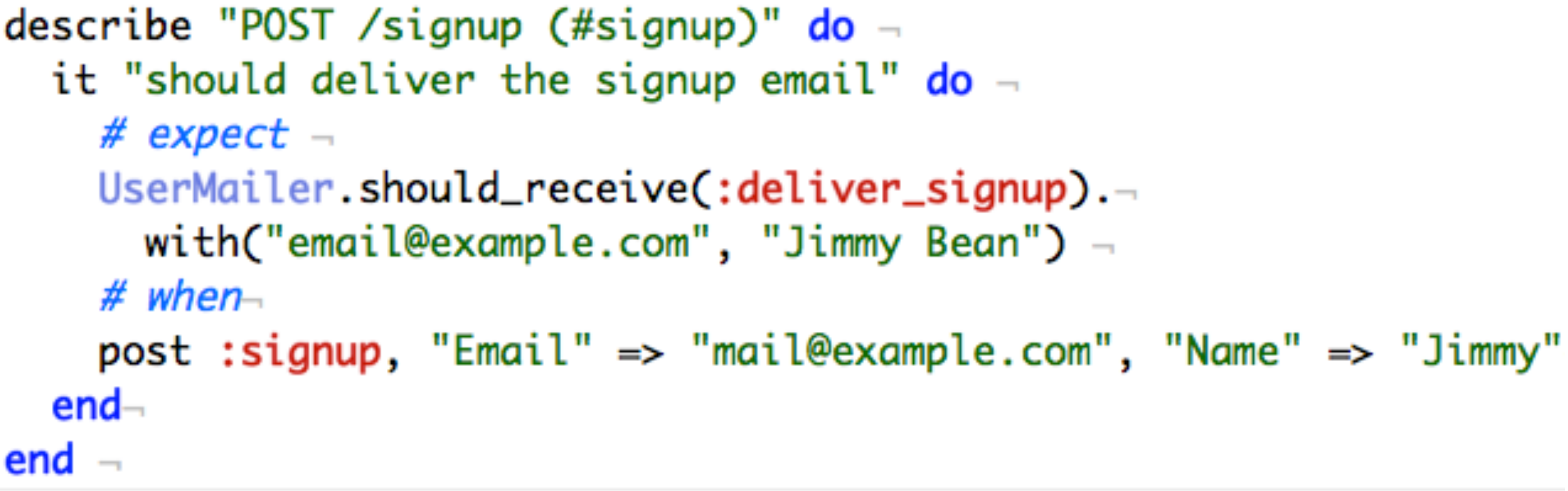
- Then save and open current email
- Then save and open all text emails
- Then save and open all html emails
- Then save and open all raw emails

Cucumber Example Scenario



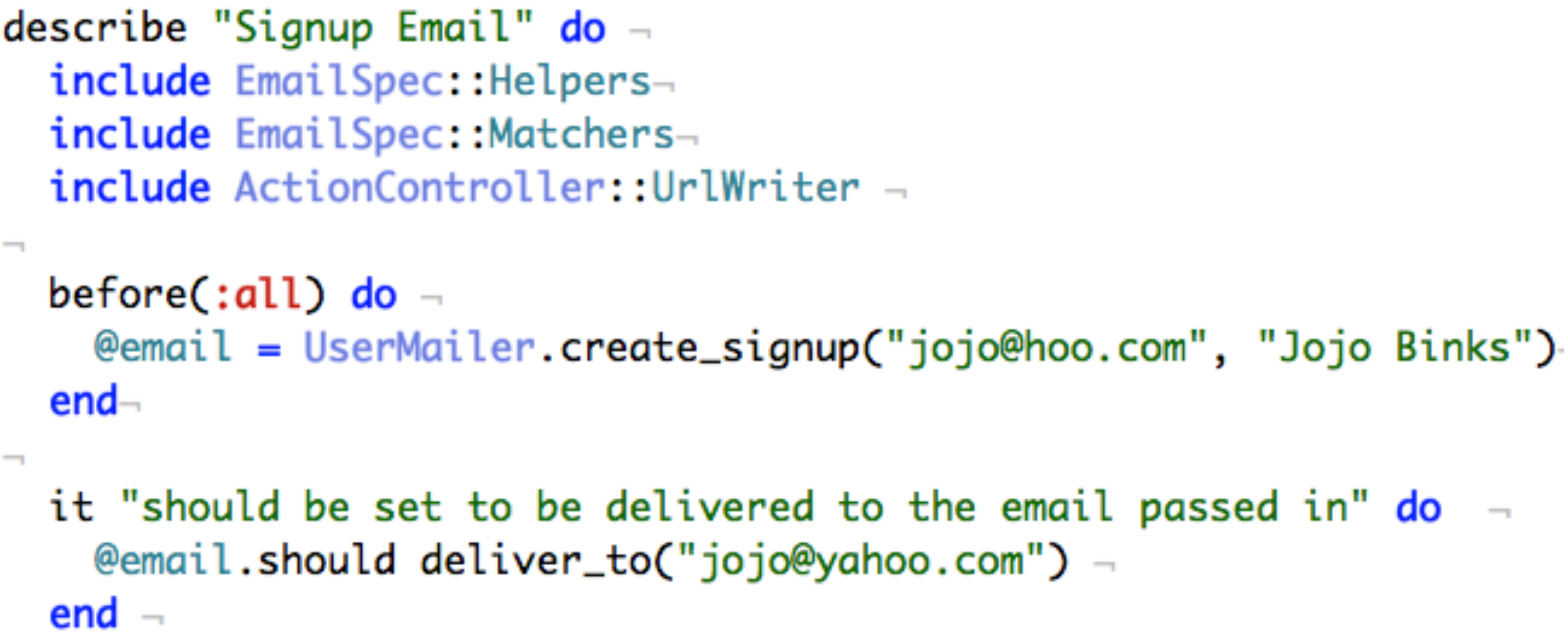
```
Scenario: A new person signs up

Given I am at "/"
When I fill in "Email" with quentin@example.com
And I press "Sign up"
And I should receive an email
When I open the email
Then I should see "confirm" in the email body
When I follow "confirm" in the email
Then I should see "Confirm your new account"
```

RSpec Example Test



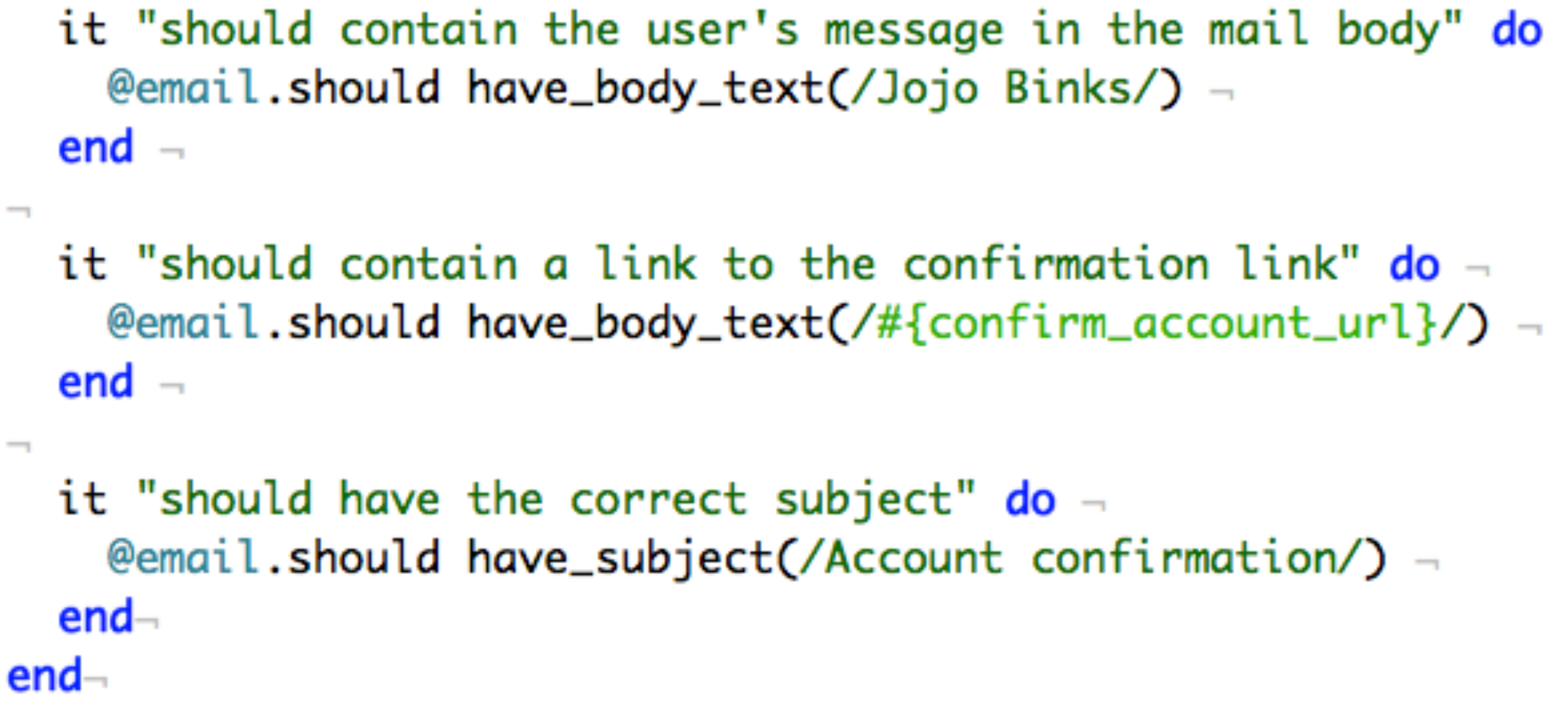
```
describe "POST /signup (#signup)" do
  it "should deliver the signup email" do
    # expect
    UserMailer.should_receive(:deliver_signup).
      with("email@example.com", "Jimmy Bean")
    # when
    post :signup, "Email" => "mail@example.com", "Name" => "Jimmy"
  end
end
```

RSpec Example Test 2



```ruby
describe "Signup Email" do
  include EmailSpec::Helpers
  include EmailSpec::Matchers
  include ActionController::UrlWriter

  before(:all) do
    @email = UserMailer.create_signup("jojo@hoo.com", "Jojo Binks")
  end

  it "should be set to be delivered to the email passed in" do
    @email.should deliver_to("jojo@yahoo.com")
  end
```

RSpec Example Test 2



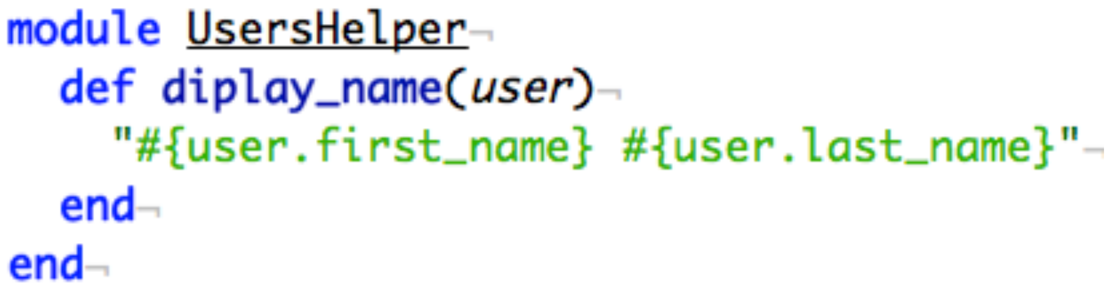
```
  it "should contain the user's message in the mail body" do
    @email.should have_body_text(/Jojo Binks/)
  end

  it "should contain a link to the confirmation link" do
    @email.should have_body_text(/#{confirm_account_url}/)
  end

  it "should have the correct subject" do
    @email.should have_subject(/Account confirmation/)
  end
end
```

Agenda

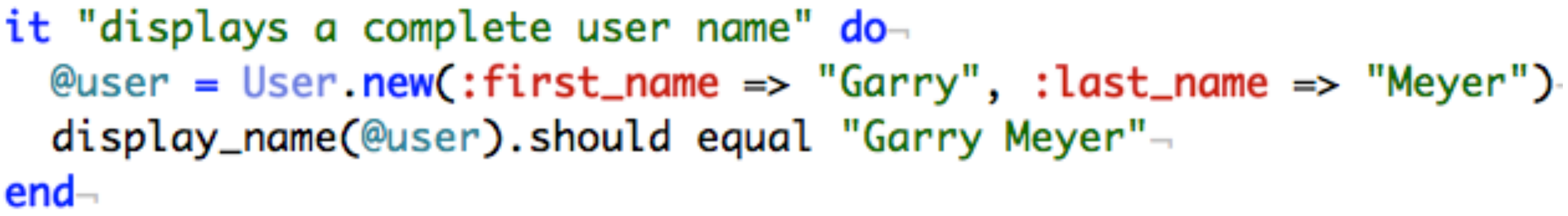
- Why Behavior-driven Design (BDD)?
- Building Blocks of Tests and BDD
  - …
  - Model Tests
  - View Tests
  - Controller Tests
  - Routing Tests
  - Outgoing Mail Tests
  - **Helper Tests**
  - Integration and Acceptance Tests
- Testing Tests & Hints for Successful Test Design
- Outlook

Helper Tests

- Helpers are filled with “the rest”
- Used as mediator between views and models or views and controllers
- (Complex) view logic is moved to helpers

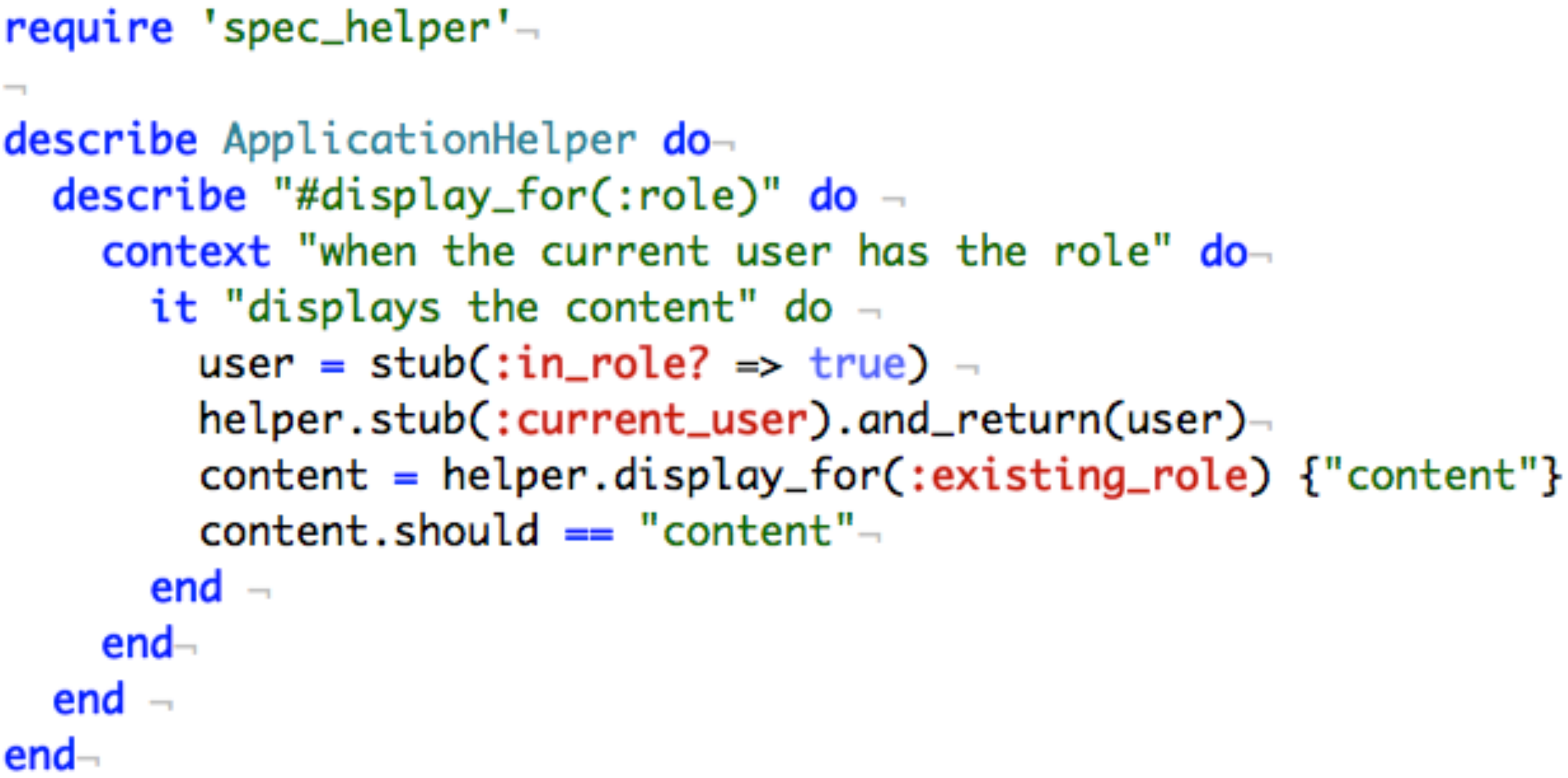


```ruby
module UsersHelper
  def diplay_name(user)
    "#{user.first_name} #{user.last_name}"
  end
end
```



```ruby
it "displays a complete user name" do
  @user = User.new(:first_name => "Garry", :last_name => "Meyer")
  display_name(@user).should equal "Garry Meyer"
end
```

Helper Tests



```
require 'spec_helper'¬
¬
describe ApplicationHelper do¬
  describe "#display_for(:role)" do ¬
    context "when the current user has the role" do¬
      it "displays the content" do ¬
        user = stub(:in_role? => true) ¬
        helper.stub(:current_user).and_return(user)¬
        content = helper.display_for(:existing_role) {"content"}
        content.should == "content"¬
      end ¬
    end¬
  end ¬
end¬
```

Agenda

- Why Behavior-driven Design (BDD)?
- Building Blocks of Tests and BDD
  - □ …
  - □ Model Tests
  - □ View Tests
  - □ Controller Tests
  - □ Routing Tests
  - □ Outgoing Mail Tests
  - □ Helper Tests
  - □ **Integration and Acceptance Tests**
- Testing Tests & Hints for Successful Test Design
- Outlook

Integration Tests

- Written by a developer for a developer

- Test communication of controllers via sessions/cookies
- Verify end-to-end behavior
- Make controller calls
- Verify that everything is okay

- Similar to controller tests, BUT
  - Not tied to one controller
  - 1..n sessions for different users

Methods

- url_for(object_instance)

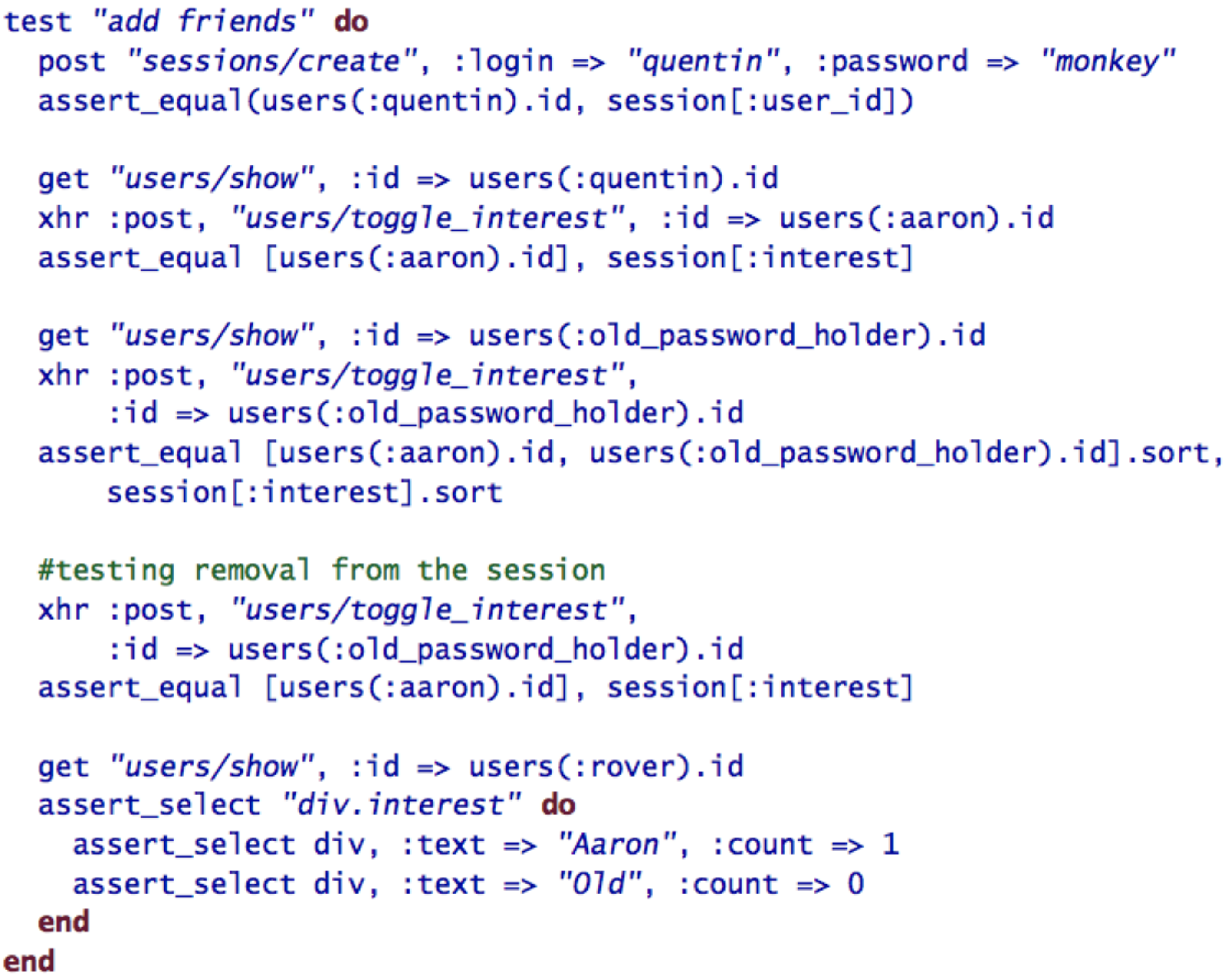
- via_redirect
- post_via_redirect
- follow_redirect!

- https!
- https!(false)

- host!(epic.hpi.uni-potsdam.de)

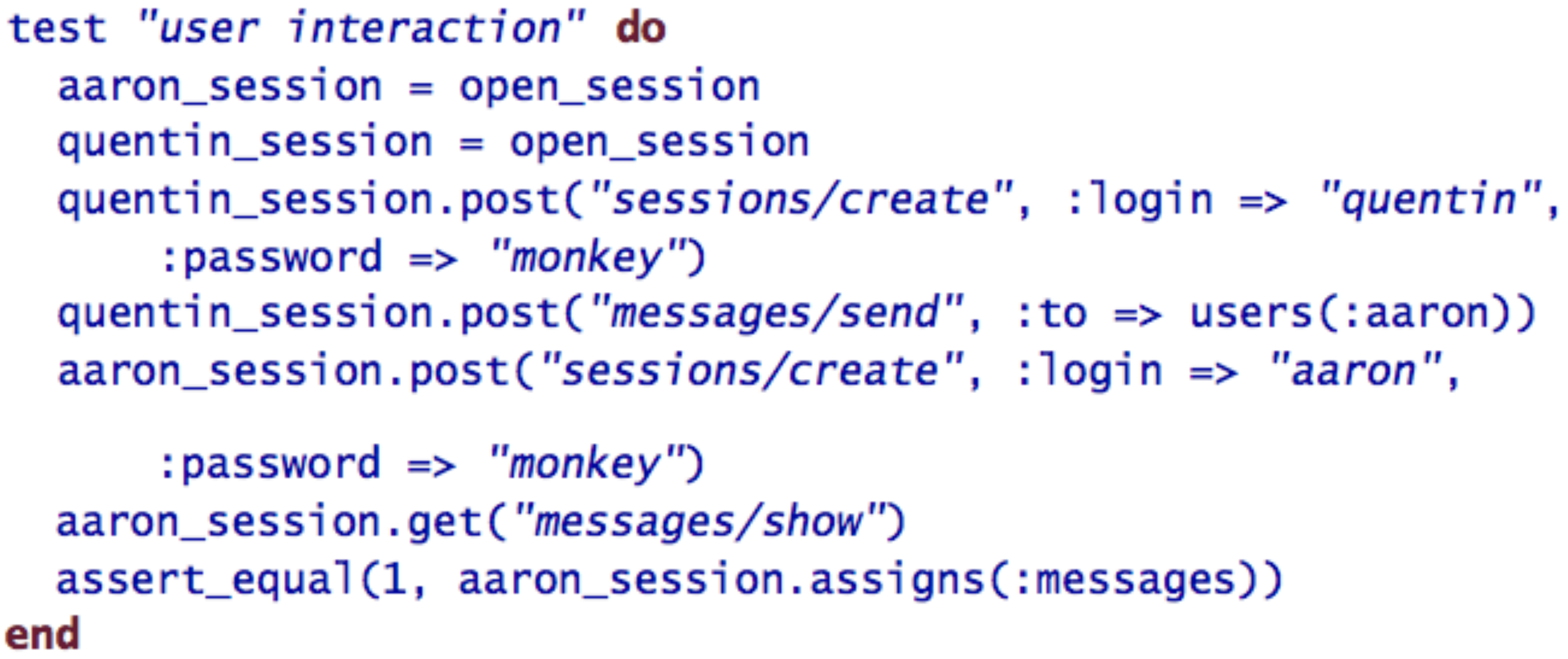
- Set initial state: session[:user_id] = 3

Test::Unit



```
test "add friends" do
  post "sessions/create", :login => "quentin", :password => "monkey"
  assert_equal(users(:quentin).id, session[:user_id])

  get "users/show", :id => users(:quentin).id
  xhr :post, "users/toggle_interest", :id => users(:aaron).id
  assert_equal [users(:aaron).id], session[:interest]

  get "users/show", :id => users(:old_password_holder).id
  xhr :post, "users/toggle_interest",
      :id => users(:old_password_holder).id
  assert_equal [users(:aaron).id, users(:old_password_holder).id].sort,
      session[:interest].sort

  #testing removal from the session
  xhr :post, "users/toggle_interest",
      :id => users(:old_password_holder).id
  assert_equal [users(:aaron).id], session[:interest]

  get "users/show", :id => users(:rover).id
  assert_select "div.interest" do
    assert_select div, :text => "Aaron", :count => 1
    assert_select div, :text => "Old", :count => 0
  end
end
```

Multiple Session Example with Test::Unit



```
test "user interaction" do
  aaron_session = open_session
  quentin_session = open_session
  quentin_session.post("sessions/create", :login => "quentin",
      :password => "monkey")
  quentin_session.post("messages/send", :to => users(:aaron))
  aaron_session.post("sessions/create", :login => "aaron",

      :password => "monkey")
  aaron_session.get("messages/show")
  assert_equal(1, aaron_session.assigns(:messages))
end
```

Webrat & Capybara

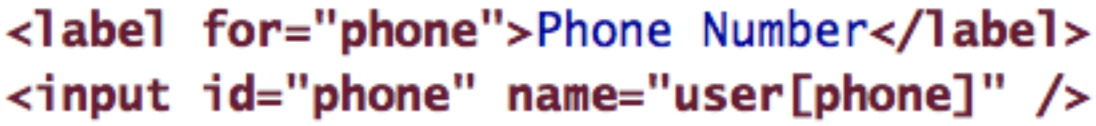
- Webrat and Capybara are very similar
- We’ll use Capybara

- DSL for
  - “browsing the Internet”
  - Acceptance testing

- Capybara is case-sensitive (due to Xpath backend)

Finding Elements

- Finding elements
  - □ label
  - □ DOM ID
  - □ form field name



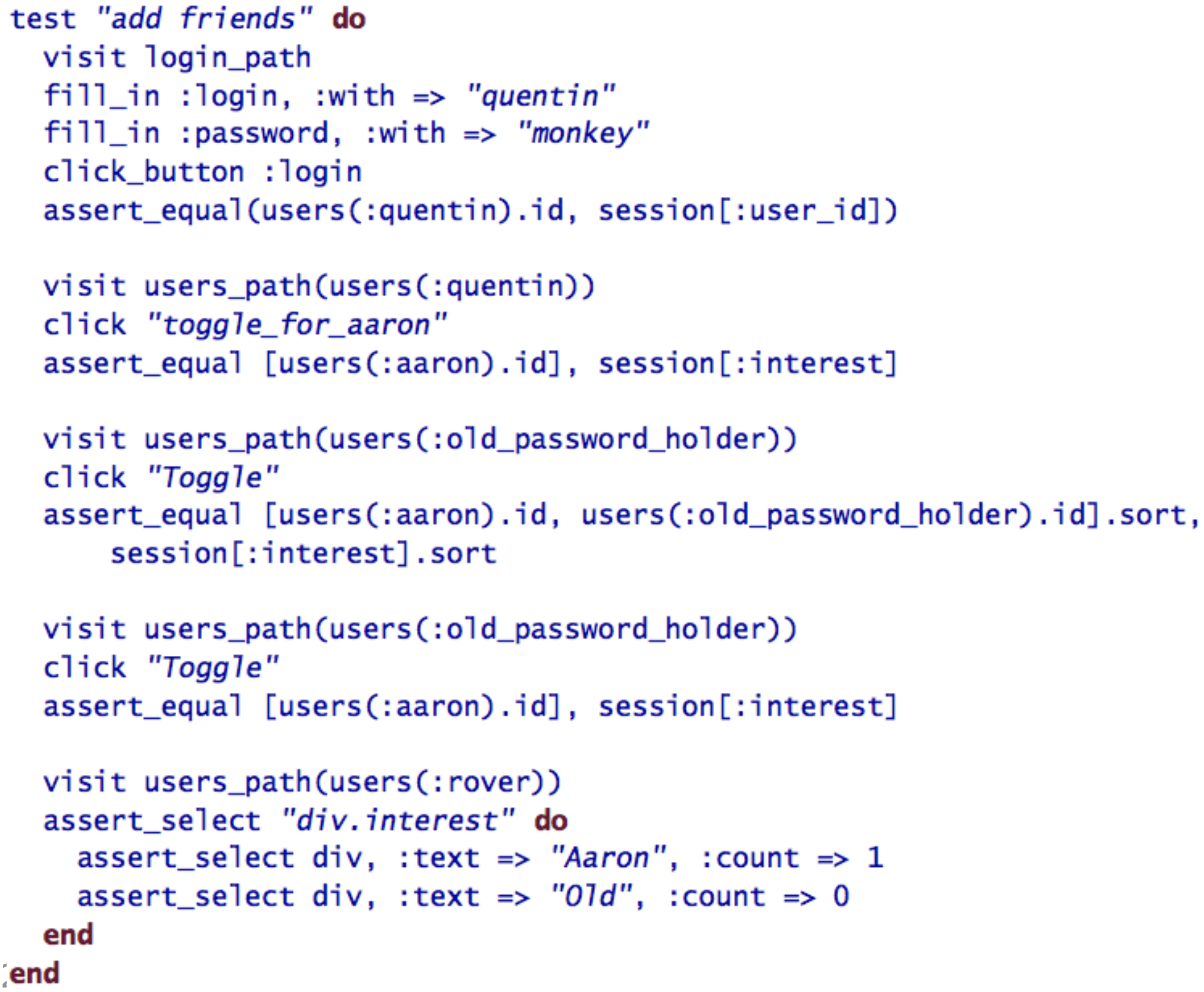
```
<label for="phone">Phone Number</label>
<input id="phone" name="user[phone]" />
```

- “Phone Number”
- “phone”
- “user[phone]”

10 Capybara Methods

- attach_file(field_locator, path, content_type = nil)
- check(field_locator)
- choose(field_locator)
- click_button(value)
- click_link(text_or_title_or_id, options = {})
- fill_in(field_locator, options = {})
- save_and_open_page()
- select(option_text, options = {})
- uncheck(field_locator)
- visit(url = nil, http_method = :get, data = {})

Clarity ⇧



```
test "add friends" do
  visit login_path
  fill_in :login, :with => "quentin"
  fill_in :password, :with => "monkey"
  click_button :login
  assert_equal(users(:quentin).id, session[:user_id])

  visit users_path(users(:quentin))
  click "toggle_for_aaron"
  assert_equal [users(:aaron).id], session[:interest]

  visit users_path(users(:old_password_holder))
  click "Toggle"
  assert_equal [users(:aaron).id, users(:old_password_holder).id].sort,
      session[:interest].sort

  visit users_path(users(:old_password_holder))
  click "Toggle"
  assert_equal [users(:aaron).id], session[:interest]

  visit users_path(users(:rover))
  assert_select "div.interest" do
    assert_select div, :text => "Aaron", :count => 1
    assert_select div, :text => "Old", :count => 0
  end
end
```

Cucumber

- Features
  - Gherkin (i.e. the language Cucumber understands)
  - Title
  - Narrative
  - Scenarios
    - Title
    - Steps
- cucumber command
  - Parses steps
  - Map them to step definitions
- Step definition
  - Written in Ruby (or other languages)
  - “implements” a step
  - https://www.ruby-toolbox.com/categories/Cucumber_Steps for helpful add-ons and predefined steps

Gherkin

- Feature
- Background
- Scenario
- Scenario Outline
- Scenarios
- Given
- When
- Then
- And / But
- |
- “””
- #

Predefined Steps

- Given I am on the homepage | <RESTresource>s page
- When I go to (.+)
- When I press ”button” (within ”selector”)
- When I follow ”link” (within ”selector”)
- When I fill in ”field" with ”value” (within ”selector”)
- When I fill in ”value" for “field” (within ”selector”)
- When I fill in the following:

Account Number	5002
Expiry date	2009-11-01
Note	Nice guy

Predefined Steps

- When I select ”value" from ”field” (within “selector”)
- When I check|uncheck “field” (within “selector”)
- When I choose “field” (within “selector”)
- When I attach the file ”path" to “field” (within ”selector”)
- Then I should see JSON:
- Then I should (not) see “text” (within “selector")
- Then I should (not) see \regex/ (within “selector”)

Predefined Steps

- Then the ”field" field (within ”selector”)
  should (not) contain ”value”
- Then the ”label" checkbox (within ”selector”)
  should (not) be checked
- Then I should be on the <RESTresource>s page
- Then I should have the following query string:
- Then show me the page

Declarative vs. Imperative

- Scenario: transfer money (declarative)
  - Given I have $100 in checking
  - And I have $20 in savings
  - When I transfer $15 from checking to savings
  - Then I should have $85 in checking
  - And I should have $35 in savings

[RSpec Book]

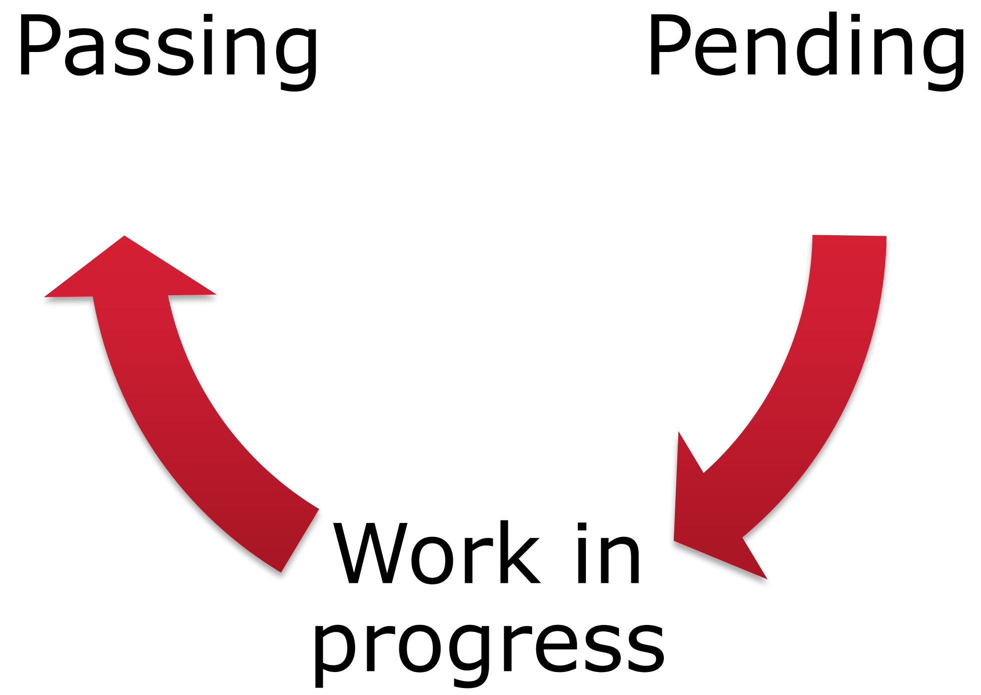
- Scenario: transfer money (imperative)
  - Given I have $100 in checking
  - And I have $20 in savings
  - When I go to the transfer form
  - And I select "Checking" from "Source Account”
  - And I select "Savings" from "Target Account”
  - And I fill in "Amount" with "15”
  - And I press "Execute Transfer”
  - Then I should see that I have $85 in checking
  - And I should see that I have $35 in savings

Organizing Features

- ./features/*.feature
- ./features/<epic>/*.feature

- cucumber features
- Cucumber features/<epic>

Tagging



Passing

Pending

Work in progress

Tags

- @wip
- Cucumber --tags @wip

- @wip @piw
- AND: cucumber --tags @wip --tags @piw
- OR: cucumber --tags @wip,@piw
- NOT: cucumber --tags @wip ~@piw

- Run certain scenarios / features
- Only run in certain environments
- Relate epics

Calling Steps From Steps

- When /I transfer (.*) from (.*) to (.*)/ do |amount, source, target|
  - When "I select #{source} as the source account”
  - When "I select #{target} as the target account”
  - When "I set #{amount} as the amount”
  - When "I click transfer”
- end

- When /I transfer (.*) from (.*) to (.*)/ do |amount, source, target|
  - steps %Q{
    - When I select #{source} as the source account
    - And I select #{target} as the target account
    - And I set #{amount} as the amount And I click transfer
  - }
- end

Background

- Feature: invite friends
    - Background: Logged in
        - Given I am logged in as "Aslak”
        - And the following people exist:
            - | name | friend? |
            - | David | yes |
            - | Vidkun | no |
    - Scenario: Invite someone who is already a friend
    - Scenario: Invite someone who is not a friend
    - Scenario: Invite someone who is not a friend who doesn't have an account

Multi-Line Text

```
Scenario: pending implementation
    Given a file named "example_without_block_spec.rb" with:
        """
        describe "an example" do
            it "has not yet been implemented”
        end
        """
    When I run "spec example_without_block_spec.rb”
    Then the exit code should be 0
    And the stdout should include
        """
        Pending:

        an example has not yet been implemented \(Not Yet Implemented\)
        .\/example_without_block_spec.rb:2

         Finished in ([\d\.]*) seconds

         1 example, 0 failures, 1 pending
        """
```

Cucumber Configuration

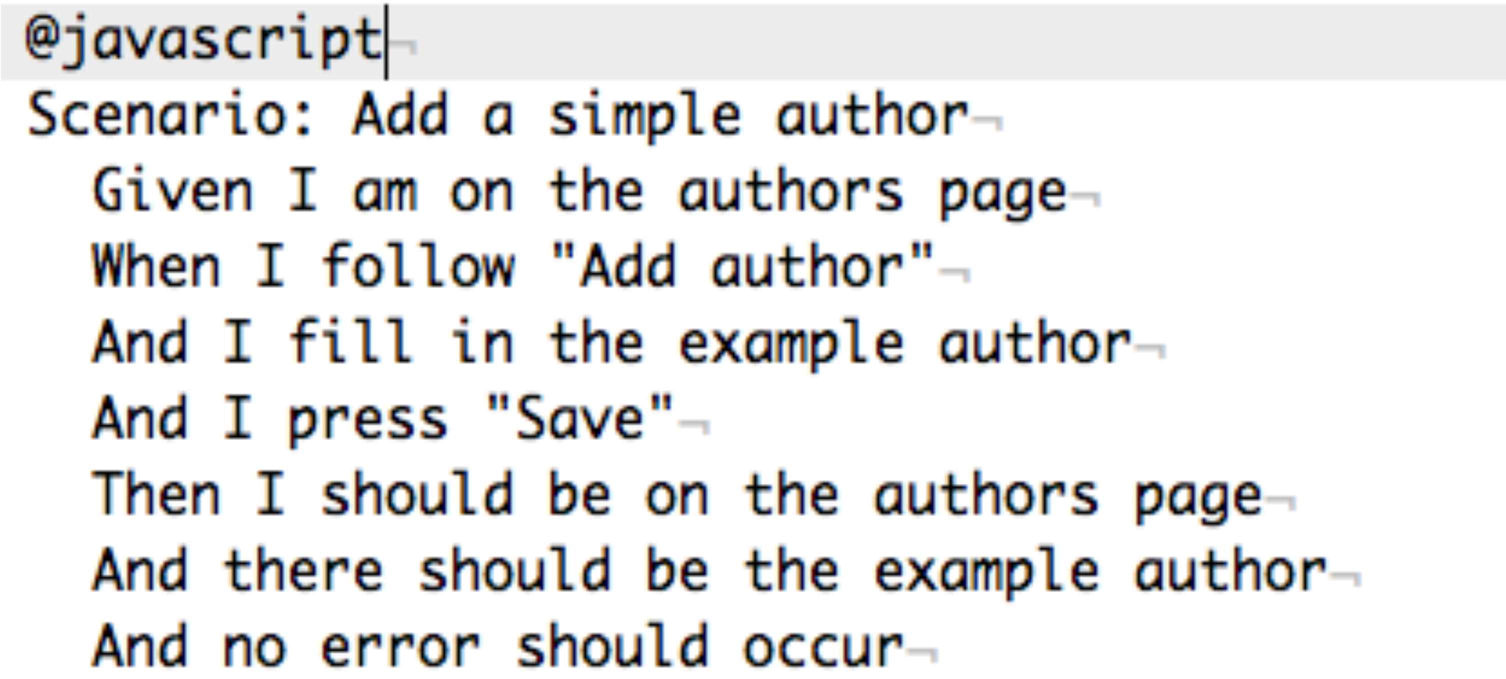
- config/cucumber.yml
- Define profiles
  - wip: --tags @wip features
  - cucumber –p wip
- Define default format

Capybara and Ajax

- Capybara uses
  - Celerity (http://celerity.rubyforge.org/) or
  - Culerity (https://github.com/langalex/culerity/)
  - To run JS without a browser

https://github.com/jnicklas/capybara

Capybara and Ajax



```
@javascript
Scenario: Add a simple author
  Given I am on the authors page
  When I follow "Add author"
  And I fill in the example author
  And I press "Save"
  Then I should be on the authors page
  And there should be the example author
  And no error should occur
```

Agenda

- Behavior-Driven Development of MasterMind
- Why Behavior-driven Design (BDD)?
- Building Blocks of Tests and BDD
- **Testing Tests & Hints for Successful Test Design**
- Outlook

Testing Tests

- Test coverage
- Fault seeding
- Mutation testing

Tests Coverage

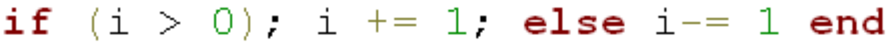
- Most commonly used metric for evaluating test suite quality

- Test coverage = executed code during test suite run / all code *100
- 85 loc / 100 loc = 85% test coverage

1. Absence of code coverage indicates a potential problem
2. Existence of code coverage means very little
3. In combination with good testing practices, coverage might say something about test suite reach

How to Measure Coverage?

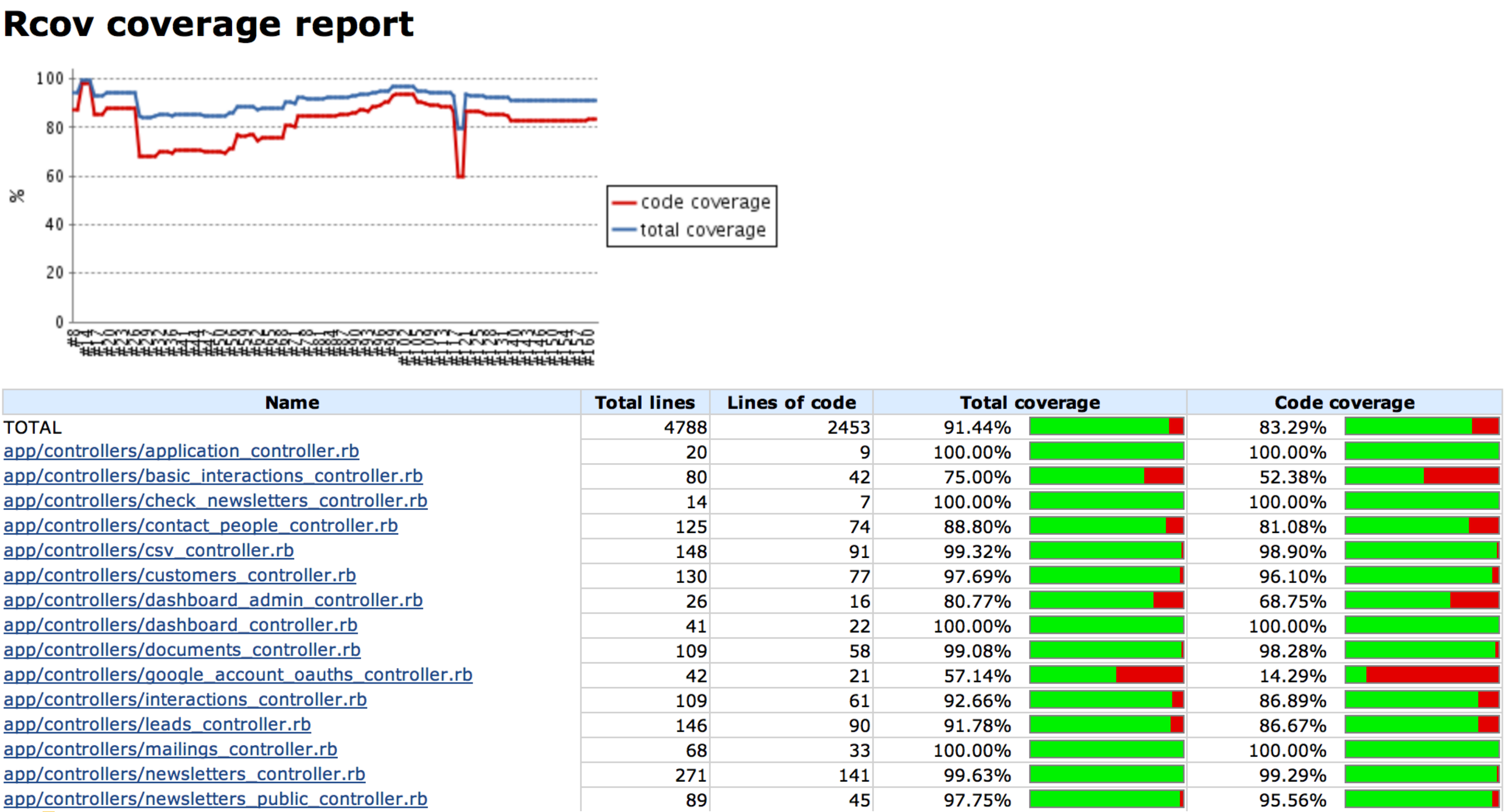
- Most useful approaches
  - □ Line coverage
  - □ Branch coverage
- Tool
  - □ SimpleCov (https://github.com/colszowka/simplecov) - Ruby 1.9
  - □ Rcov (https://github.com/relevance/rcov) for 1.8
  - □ Uses line coverage



```
if (i > 0); i += 1; else i-= 1 end
```

  - □ ➔ 100% code coverage although 1 branch wasn’t executed

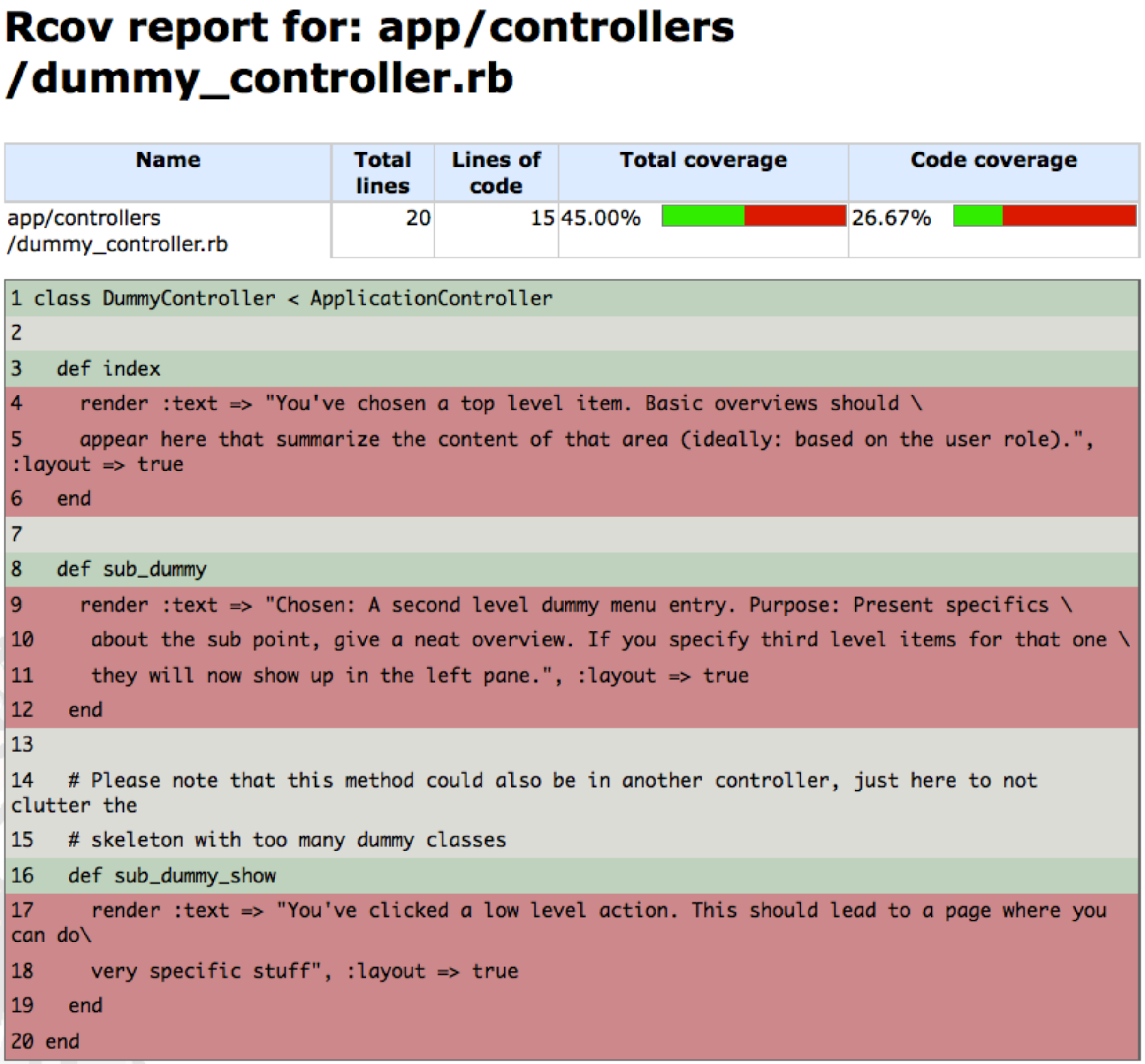
Rcov / SimpleCov



Rcov coverage report

Name	Total lines	Lines of code	Total coverage	Code coverage
TOTAL	4788	2453	91.44%	83.29%
app/controllers/application_controller.rb	20	9	100.00%	100.00%
app/controllers/basic_interactions_controller.rb	80	42	75.00%	52.38%
app/controllers/check_newsletters_controller.rb	14	7	100.00%	100.00%
app/controllers/contact_people_controller.rb	125	74	88.80%	81.08%
app/controllers/csv_controller.rb	148	91	99.32%	98.90%
app/controllers/customers_controller.rb	130	77	97.69%	96.10%
app/controllers/dashboard_admin_controller.rb	26	16	80.77%	68.75%
app/controllers/dashboard_controller.rb	41	22	100.00%	100.00%
app/controllers/documents_controller.rb	109	58	99.08%	98.28%
app/controllers/google_account_oauths_controller.rb	42	21	57.14%	14.29%
app/controllers/interactions_controller.rb	109	61	92.66%	86.89%
app/controllers/leads_controller.rb	146	90	91.78%	86.67%
app/controllers/mailings_controller.rb	68	33	100.00%	100.00%
app/controllers/newsletters_controller.rb	271	141	99.63%	99.29%
app/controllers/newsletters_public_controller.rb	89	45	97.75%	95.56%

Rcov / SimpleCov



Rcov report for: app/controllers /dummy_controller.rb

Name	Total lines	Lines of code	Total coverage		Code coverage	
app/controllers /dummy_controller.rb	20	15	45.00%	🟩🟥	26.67%	🟩🟥

```
class DummyController < ApplicationController

  def index
    render :text => "You've chosen a top level item. Basic overviews should \
    appear here that summarize the content of that area (ideally: based on the user role).", :layout => true
  end

  def sub_dummy
    render :text => "Chosen: A second level dummy menu entry. Purpose: Present specifics \
     about the sub point, give a neat overview. If you specify third level items for that one \
     they will now show up in the left pane.", :layout => true
  end

  # Please note that this method could also be in another controller, just here to not clutter the
  # skeleton with too many dummy classes
  def sub_dummy_show
     render :text => "You've clicked a low level action. This should lead to a page where you can do\
     very specific stuff", :layout => true
  end
end
```

How Much is Enough?

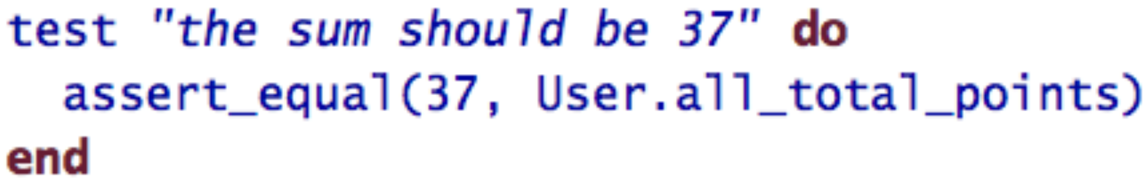
- 100% code coverage says nothing
- 0% says much
- Almost 100% is a side effect of BDD

5 Habits of Highly Successful Tests

- Independence
  - of external test data
  - of other tests (or test order)
- Repeatability
  - Same results each test run
  - Problems
    - date (Timecop)
    - random numbers (try to avoid them)

5 Habits of Highly Successful Tests

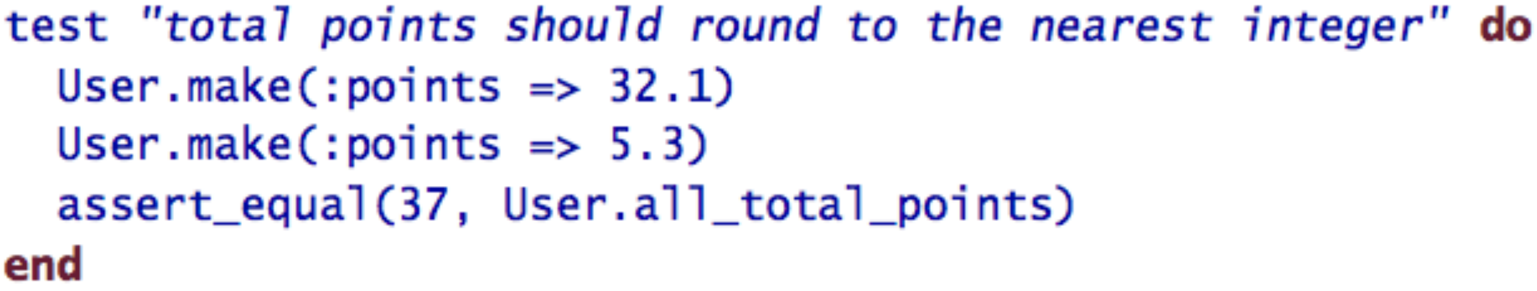
- Clarity
  - □ Test purpose should be immediately understandable
  - □ Readability
  - □ How does the test fit into the larger test suite?
  - □ Worst case:



```
test "the sum should be 37" do
  assert_equal(37, User.all_total_points)
end
```

5 Habits of Highly Successful Tests

- Clarity
  - …
  - Better:

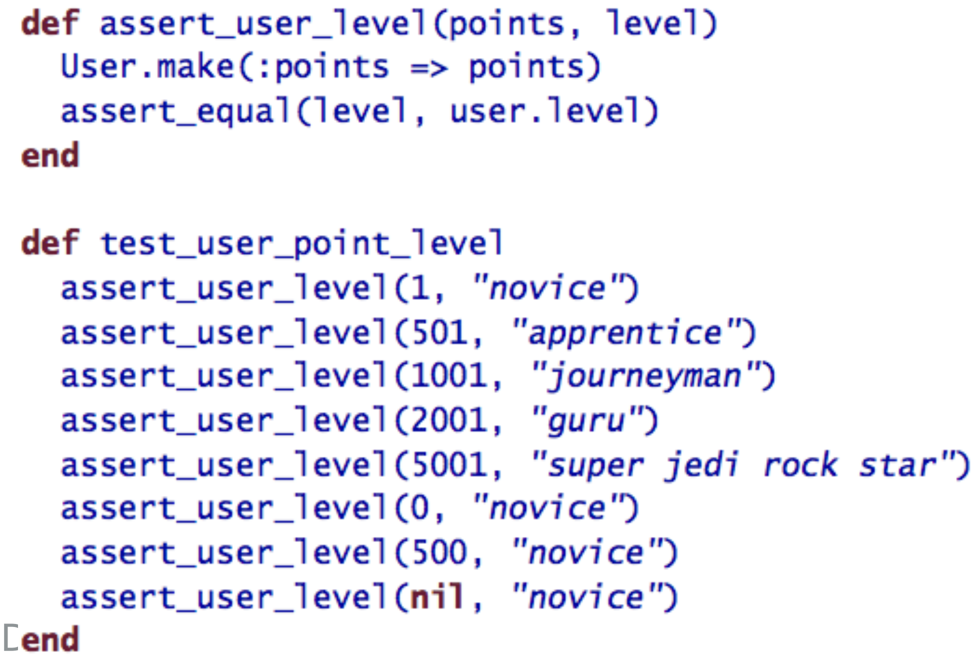


```
test "total points should round to the nearest integer" do
  User.make(:points => 32.1)
  User.make(:points => 5.3)
  assert_equal(37, User.all_total_points)
end
```

  - “Debugging is harder than coding”
  - Tests should be simple

5 Habits of Highly Successful Tests

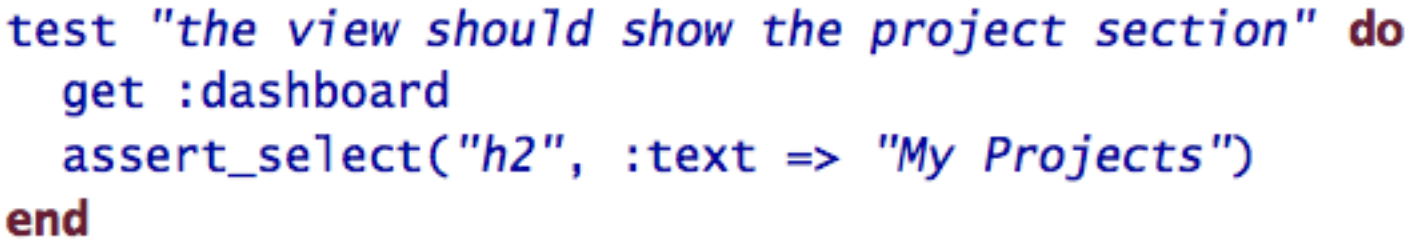
- Conciseness
  - □ Use the minimum amount of code and objects
  - □ Clear beats concise
  - □ Writing the minimum amount of tests
  - □ → tests will be faster



```
def assert_user_level(points, level)
  User.make(:points => points)
  assert_equal(level, user.level)
end

def test_user_point_level
  assert_user_level(1, "novice")
  assert_user_level(501, "apprentice")
  assert_user_level(1001, "journeyman")
  assert_user_level(2001, "guru")
  assert_user_level(5001, "super jedi rock star")
  assert_user_level(0, "novice")
  assert_user_level(500, "novice")
  assert_user_level(nil, "novice")
end
```

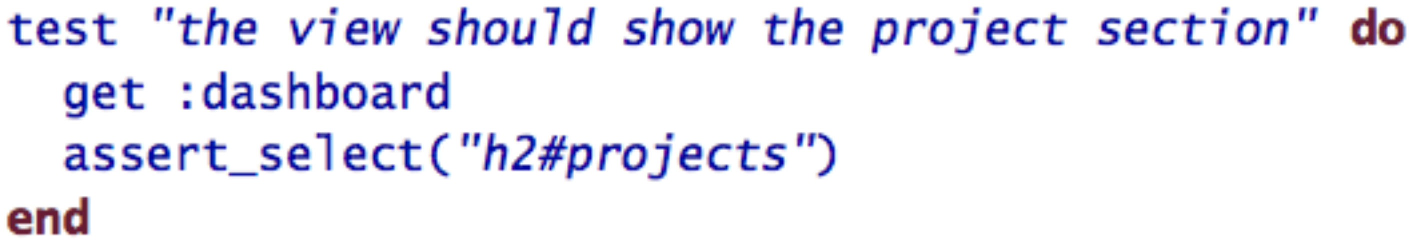
5 Habits of Highly Successful Tests

- Robustness
  - □ Tests the logic as intended
  - □ Code is correct → tests passes
  - □ Code is wrong → test does not pass
  - □ Example: view testing



```
test "the view should show the project section" do
  get :dashboard
  assert_select("h2", :text => "My Projects")
end
```

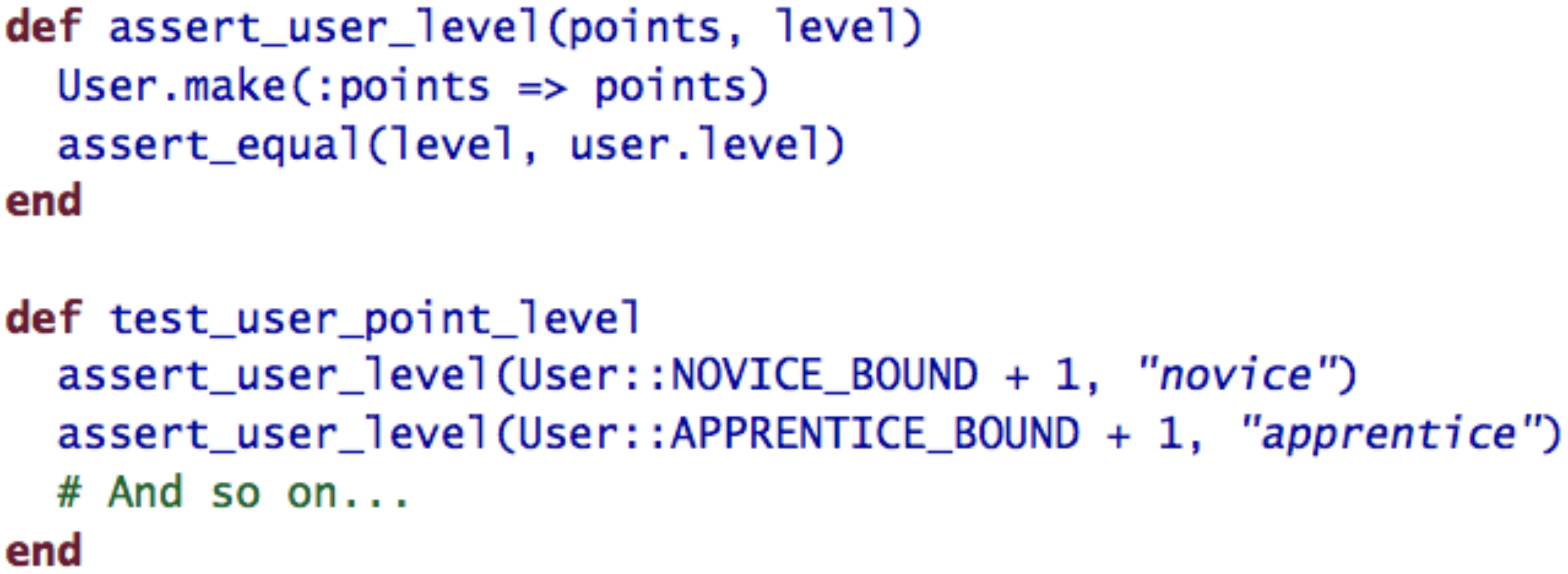
vs.



```
test "the view should show the project section" do
  get :dashboard
  assert_select("h2#projects")
end
```

5 Habits of Highly Successful Tests

- Robustness



```
def assert_user_level(points, level)
  User.make(:points => points)
  assert_equal(level, user.level)
end

def test_user_point_level
  assert_user_level(User::NOVICE_BOUND + 1, "novice")
  assert_user_level(User::APPRENTICE_BOUND + 1, "apprentice")
  # And so on...
end
```

  - □ But be aware of false positives

Troubleshooting

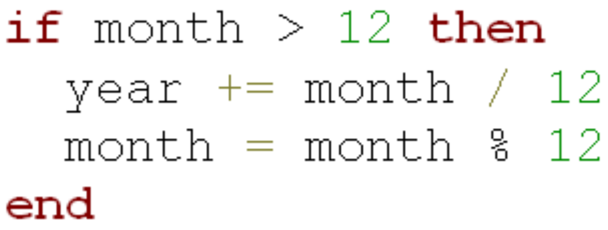
- Reproduce the error
- What has changed?
- Isolate the failure
- thing.inspect (p thing)
- Add assertions/prints to your test
- Rails.logger.error
- save_and_open_page
- Explain to someone else

Manual Fault Seeding

- Introduce a fault into your program
- Run tests
- Minimum 1 test should fail
- Warning: do not leave the fault in the software!

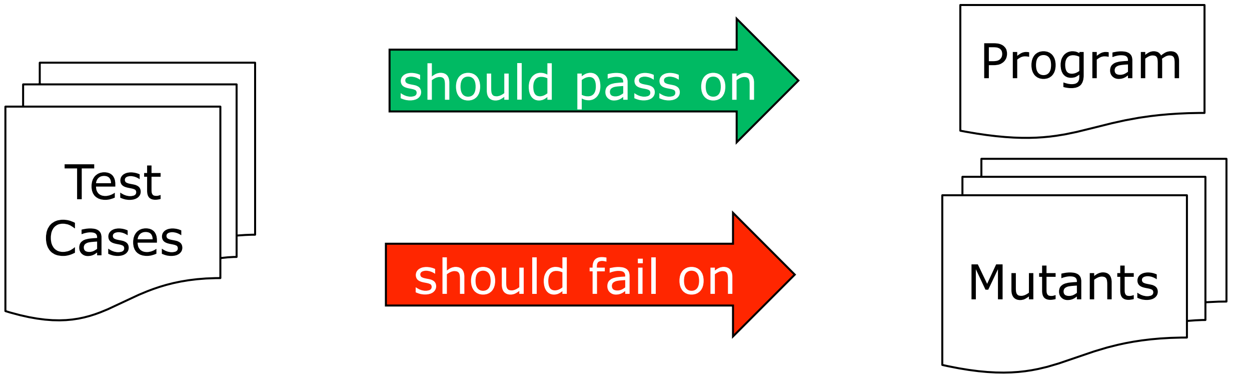
Mutation Testing

- Mutant: Slightly modified version of the program under test, differing from it by a small, syntactic change



```
if month > 12 then
  year += month / 12
  month = month % 12
end
```

To create mutants, replace:
if → if not
12 → 13
= → <



Test Cases

should pass on

should fail on

Program

Mutants

Mutation Testing

- Ruby tool: Heckle (http://ruby.sadi.st/Heckle.html)

1. Your tests should pass
2. You run Heckle to change your code
3. Test(s) should fail
4. Write tests for surviving mutants if useful

Outlook

- Lego Exercise
- Project Introduction
- Assignment until Nov 5, 11:59pm CET
  - Build teams of ~5 people
  - Name the PO of your team
  - Send info to swt2_2012_orga@lists.hpi.uni-potsdam.de