Hasso Plattner Institut

IT Systems Engineering | Universität Potsdam

BDD and Testing (in Rails)

Software Engineering II

Agenda

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

Agenda

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

Goals of Automated Developer Testing

Feature 1: Website registration

Developer 1 (no TDD/BDD, browser-based testing)	Developer 2 (with TDD/BDD, almost no browser testing)
Minute 5: working registration page Minute 8: feature is tested (3 times)	Minute 5: working test Minute 10: working implementation Minute 10.30: feature is tested (3 times)

Assumptions: 1min manual testing, 10s automatic test

Goals of Automated Developer Testing

Feature 2: Special case for feature 1

Developer 1 (no TDD/BDD, browser-based testing)	Developer 2 (with TDD/BDD, almost no browser testing)
Minute 11: implemented	Minute 12.30: test ready
Minute 14: tested (3 times)	Minute 15.30: implemented
	Minute 16.00: tested (3 times)

Goals of Automated Developer Testing

Feature 2: Special case for feature 1

Developer 1 (no TDD/BDD, browser-based testing)	Developer 2 (with TDD/BDD, almost no browser testing)
Minute 11: implemented	Minute 12.30: test ready
Minute 14: tested (3 times)	Minute 15.30: implemented
Minute 17: refactoring ready	Minute 16.00: tested (3 times)
Minute 19: tested feature 1	Minute 19: refactoring ready
Minute 21: tested feature 2	Minute 19.10: tested
Minute 22: committed	Minute 20.10: committed

Goals of Automated Testing

- Finding errors faster
- Better code (correct, robust, maintainable)
- Automated testing are used more frequently
- 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

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

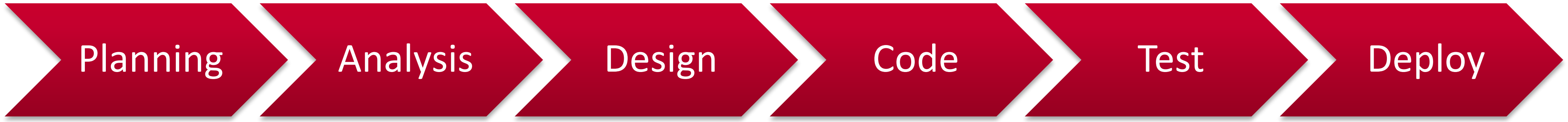
How Traditional Projects Fail

- 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



- 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

This makes sense for a bridge, ship, or a building
but Software (and Lego) are EASY to change!

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 starting with 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

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

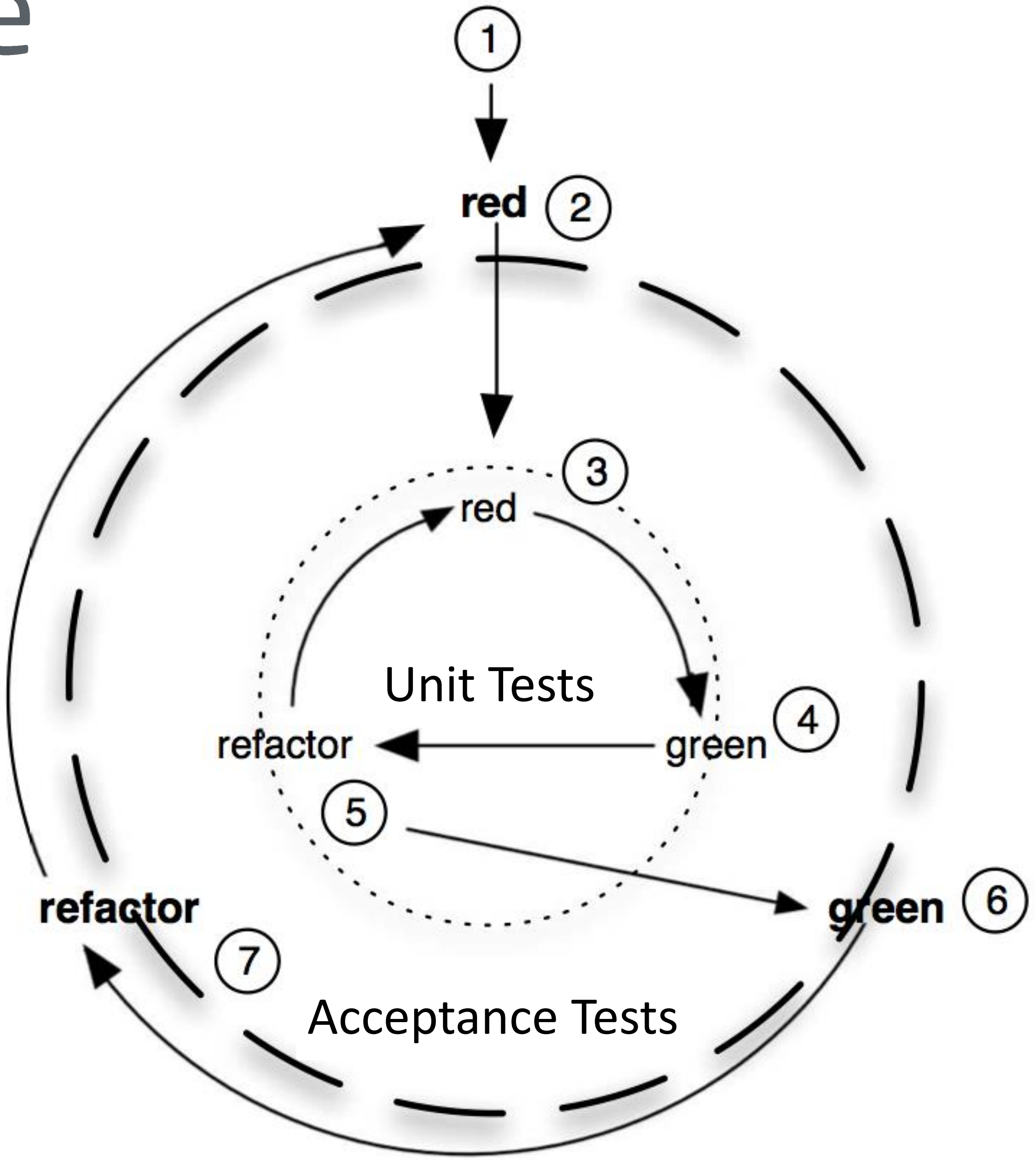
Writing Software that Matters

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

Principles

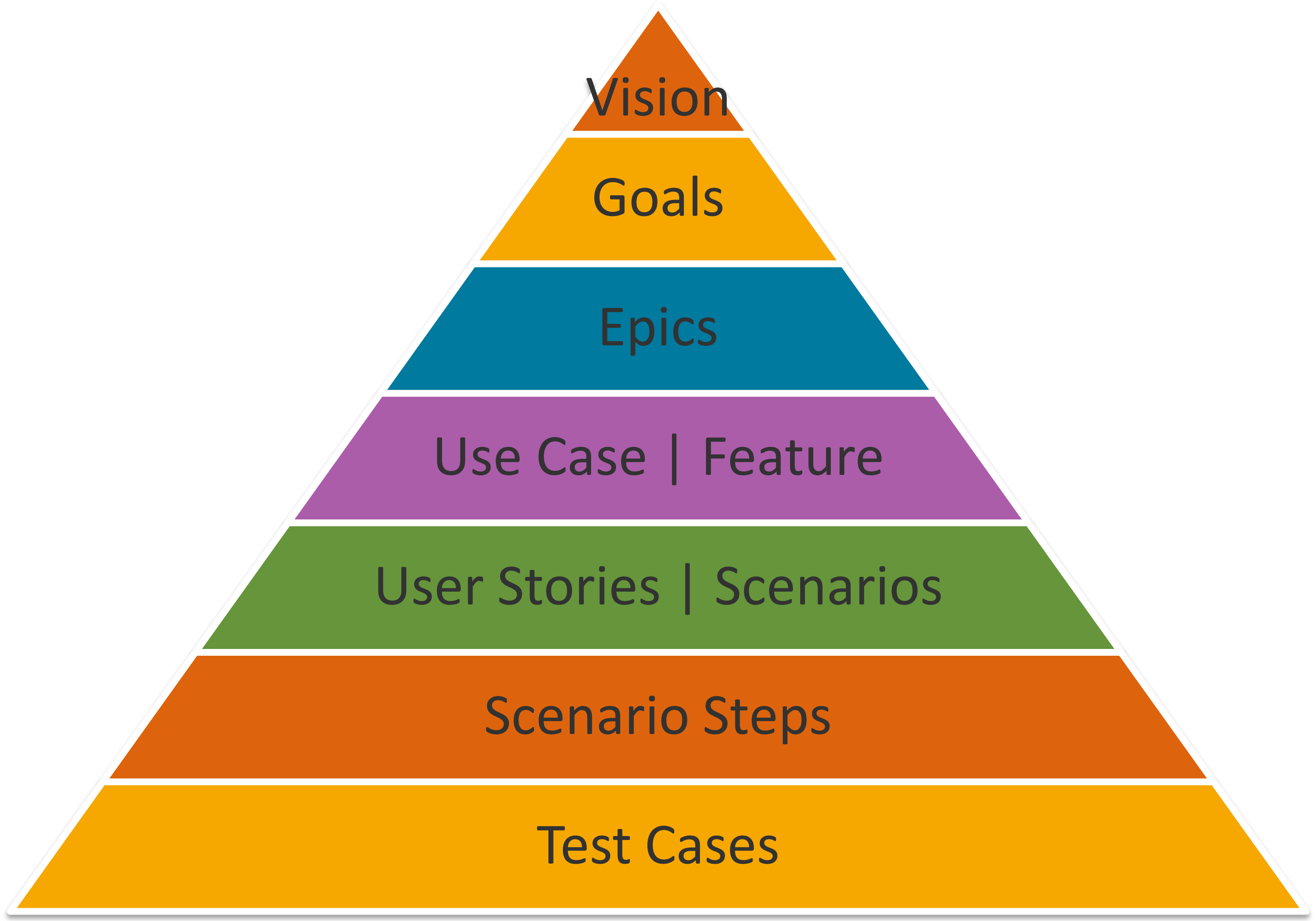
1. Enough is enough
2. Deliver stakeholder value
3. It's all behavior

BDD Cycle



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

Maximum BDD Pyramid



Vision

All Stakeholders, one statement

- 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

- How the vision will be achieved.
- Examples
  - Easier ordering process
  - Better access to suppliers’ information

Epics

- Huge themes / feature sets are described as an “epic”
- Too high level to start coding but 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 (e.g., no database-only stories)
- User stories are a token for conversations

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

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 (Cucumber)
  - scenario “”; <steps>; end (RSpec)

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

BDD Implementations

Behavior-driven development (BDD)

- Story-based definition of application behavior
- Definition of features (feature injection)
- Driven by business value (outside-in)

Cucumber

- Write test cases in a domain-specific language
- Pro: Readable by non-technicians
- Cons:
    - Translation to Ruby
    - directory structure

RSpec

- Integration tests written in plain Ruby
- Pro: No translation overhead
- Con: Barely readable by domain experts

Cucumber Example

```
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 "Add"
    Then there should be the example author
    And I should be on the authors page
```

Cucumber Overview

- Given – When – Then
- Features are located in `features/*.feature`
- Each line is a “step” that is implemented in Ruby (Capybara)
- Steps are located in features/step_definitions/
- Interpreted via regular expressions

http://github.com/jnicklas/capybara

RSpec Example

```
feature "Author Management"
  scenario "should be possible to add an author and after clicking on
'add' it should appear on the next page, which shows the overview"
        visit authors_path
        click_on "add_author"
        fill_in :name, :with "Hemmingway"
        click_on "Add"
        page.should have_content("Hemmingway")
  end
end
```

Verdict?

- Discussion 1: Which one is easier to understand ?
  - By programmers
  - By business stakeholders

- Discussion 2: Which is easier to implement?

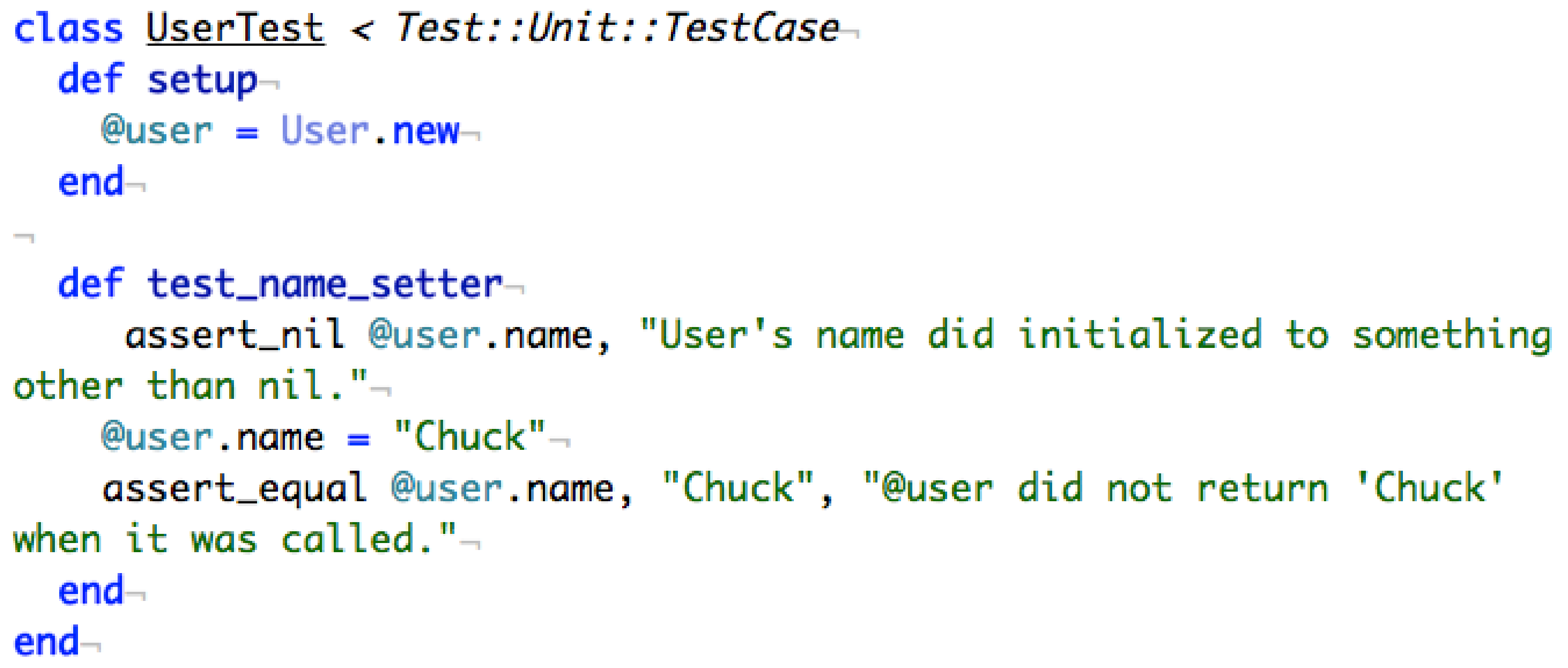
- Discussion 3: Which one to choose?
  - In this project?
  - In other projects?

More opinions:
http://www.jackkinsella.ie/2011/09/26/why-bother-with-cucumber-testing.html
http://cukes.info

Agenda

Test::Unit vs. RSpec

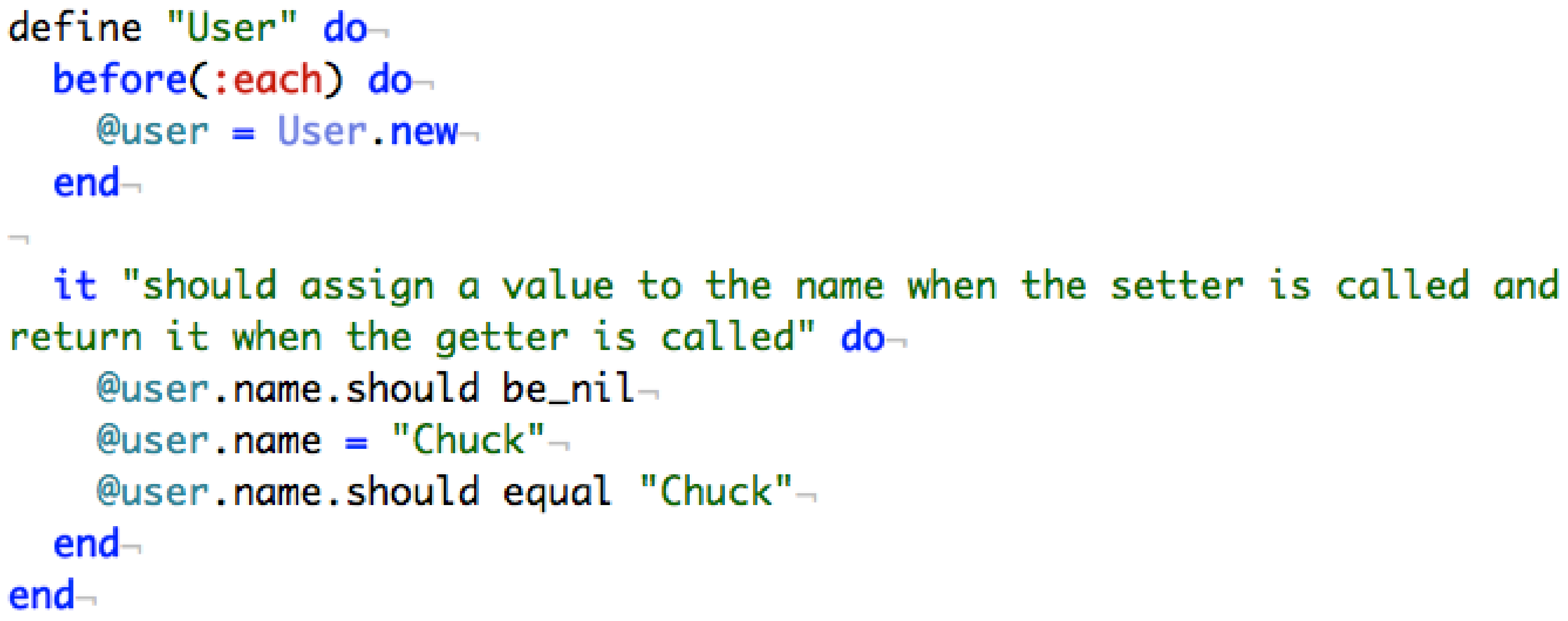
- Test::Unit comes with Ruby



```
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

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

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

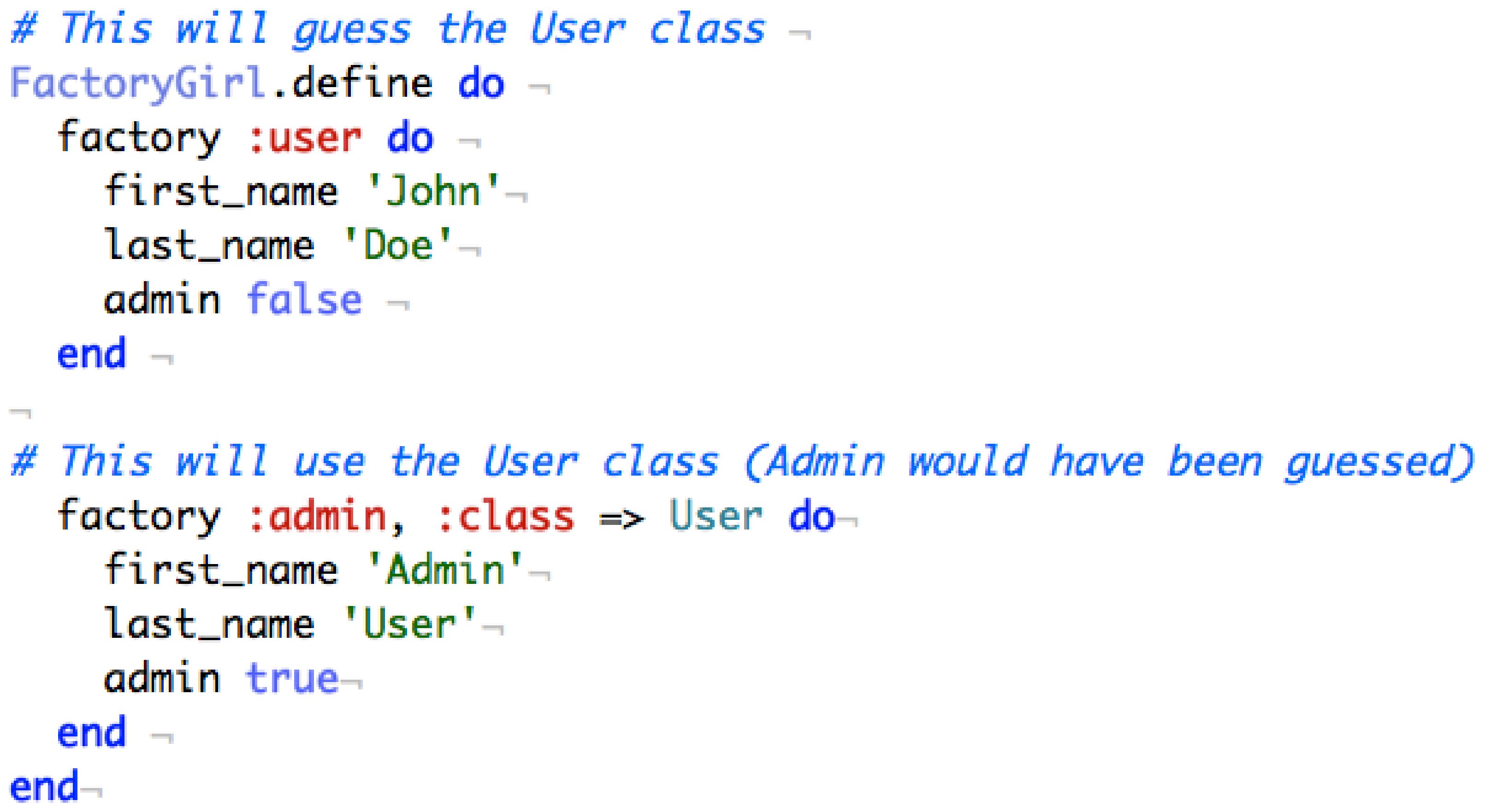
- Local (defined as closely as possible to the test)
- Compact (easy and quick to generate; even complex data sets)
- Robust (independent to other tests)

➔ Data factories

Data Factories

- Blueprint for sample instances
- Rails tool support
  - Factory Girl (our choice)
  - Machinist
  - Fabrication
  - FixtureBuilder
  - ObjectDaddy
  - …
  - https://www.ruby-toolbox.com/categories/rails_fixture_replacement
- Similar structure
  - Syntax for creating the factory blueprint
  - API for creating new objects

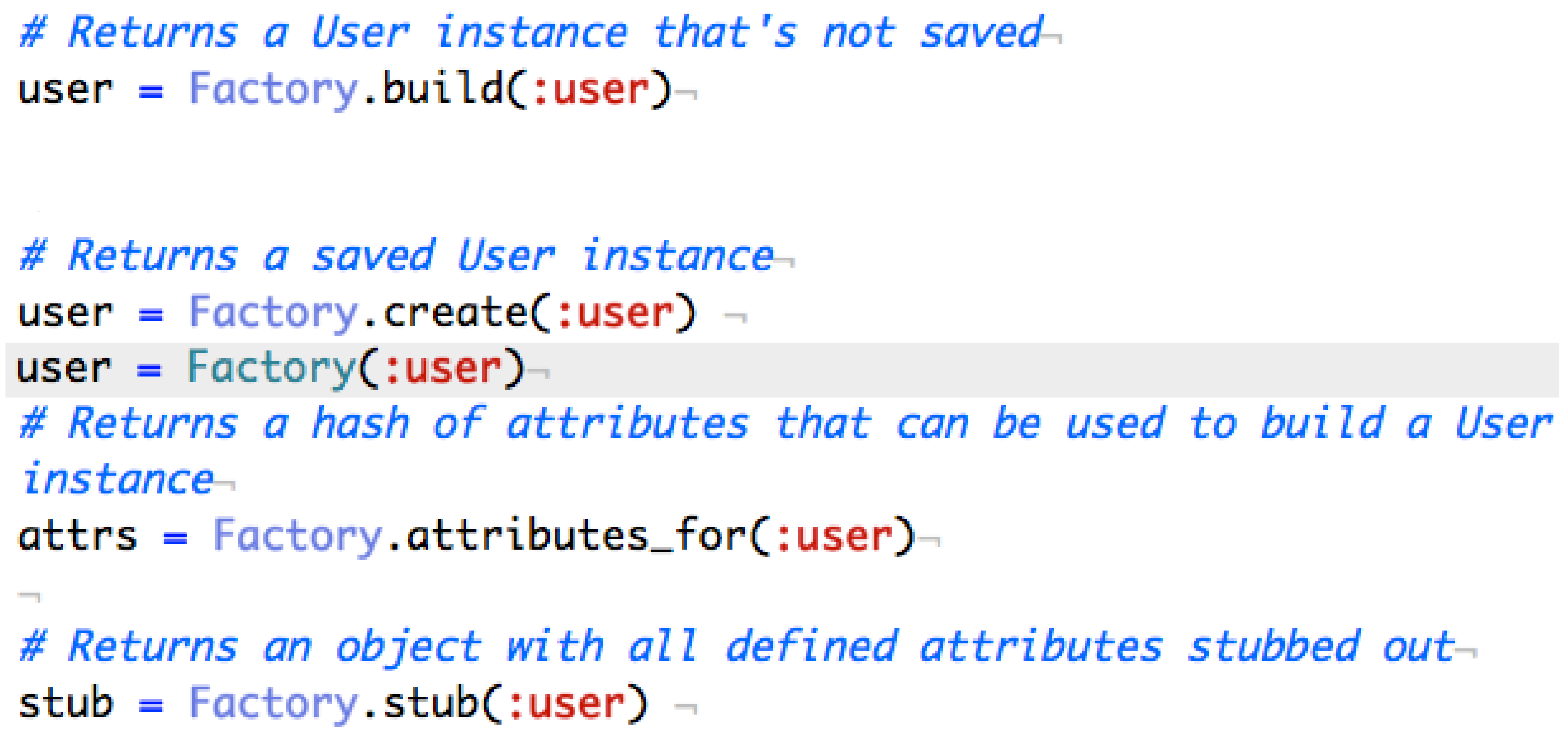
Defining Factories



```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
```

Using Factories

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



```ruby
# 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)
```

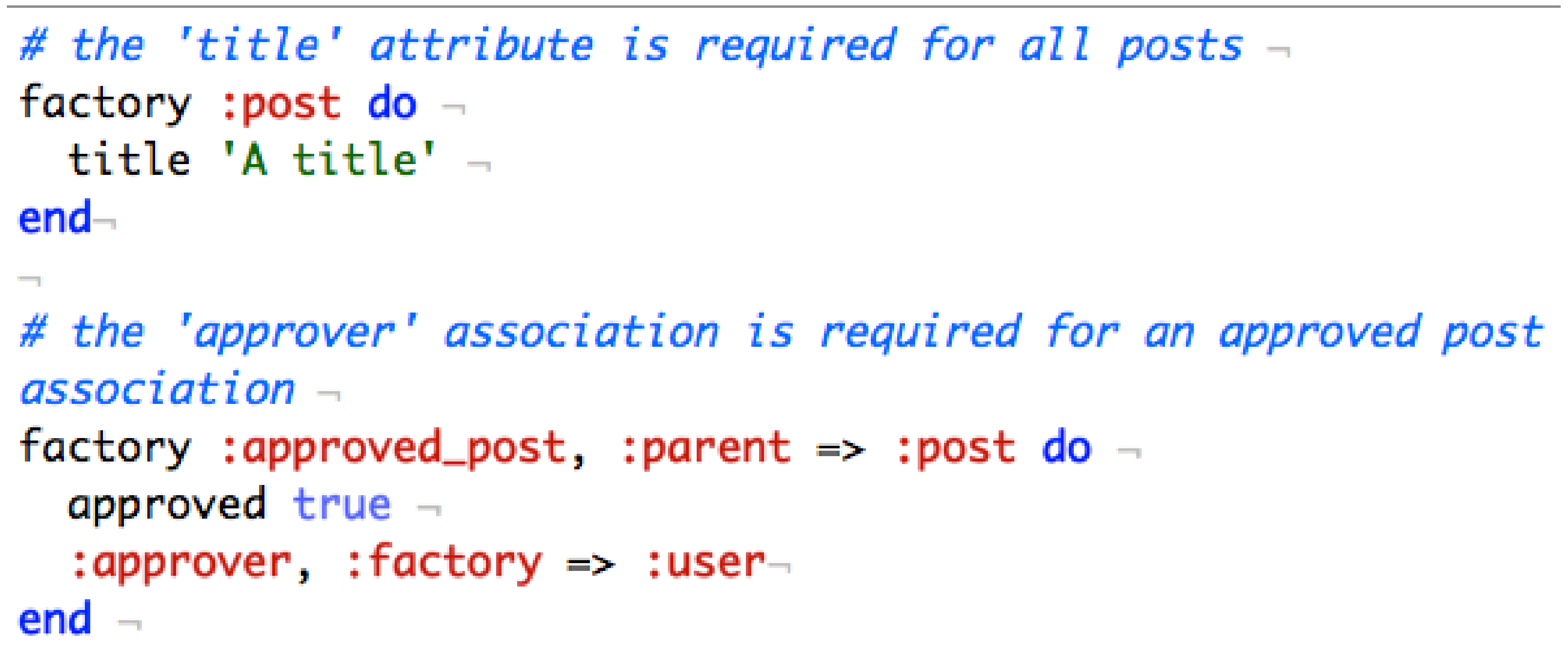
Attributes

```ruby
#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

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

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

# 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
```

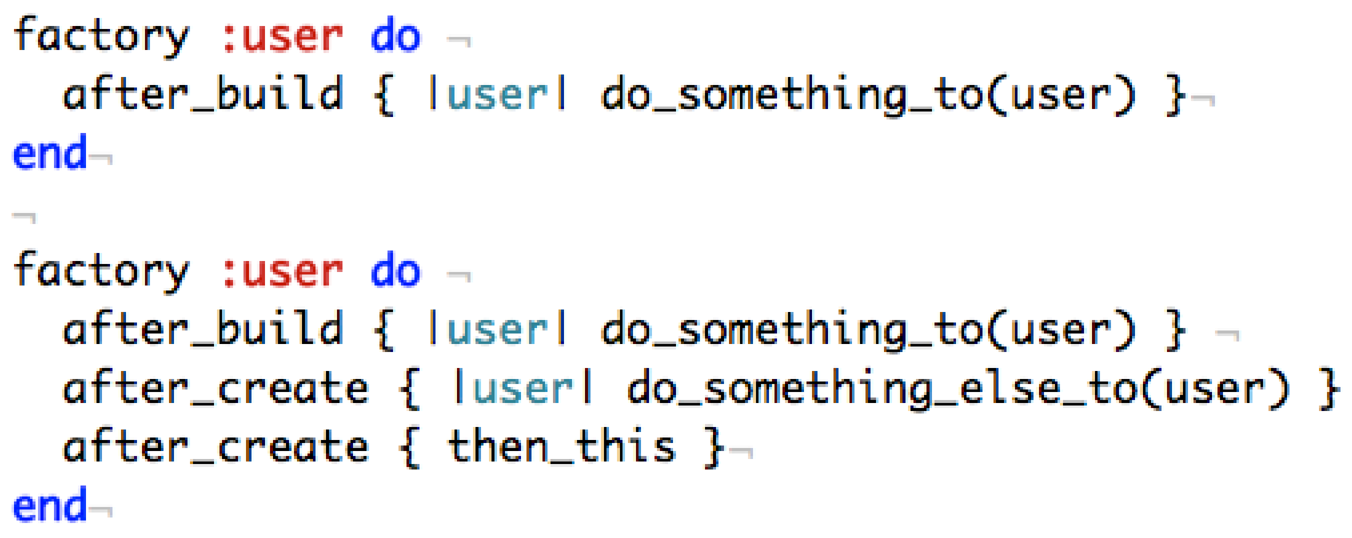
Sequences for Unique Values

```
# 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 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)



```
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

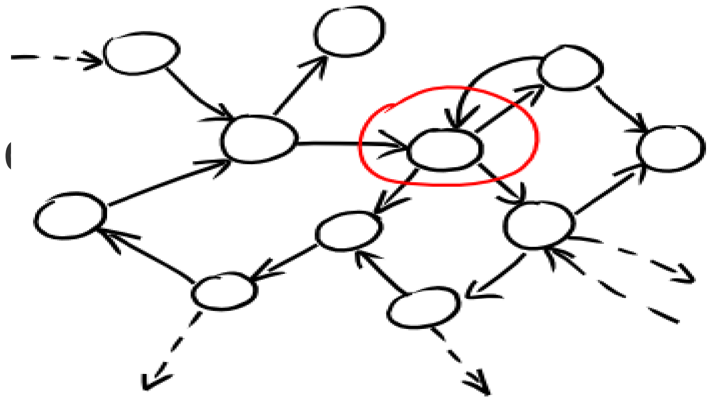
Isolation of Test Cases

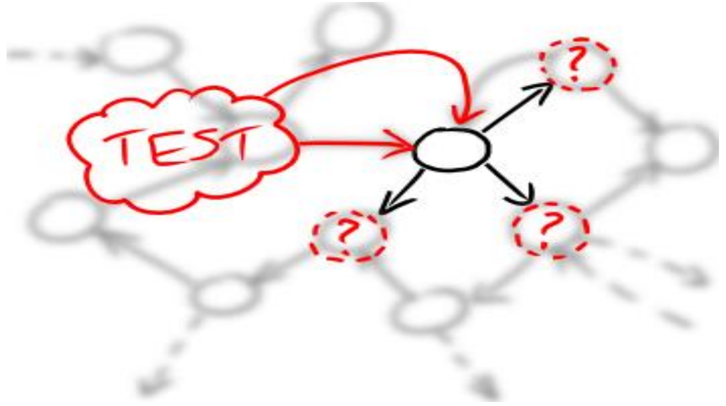
Tests should be independent

New bug in a model → only tests related to this model shoul

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

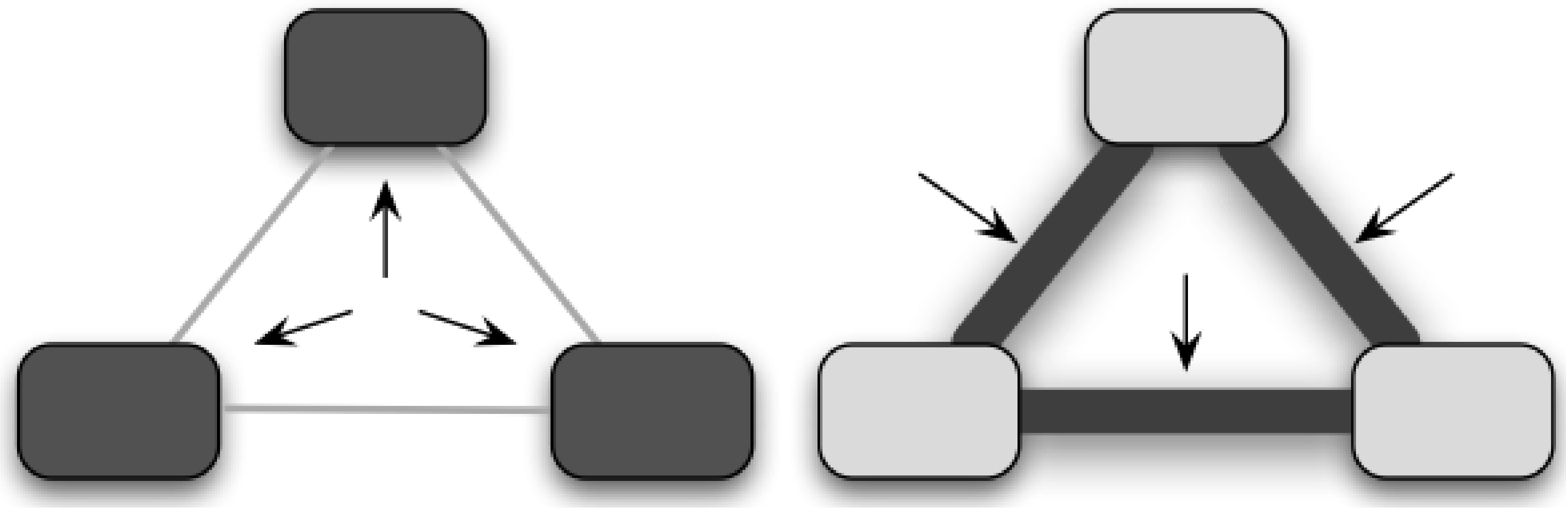
Used when

- 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

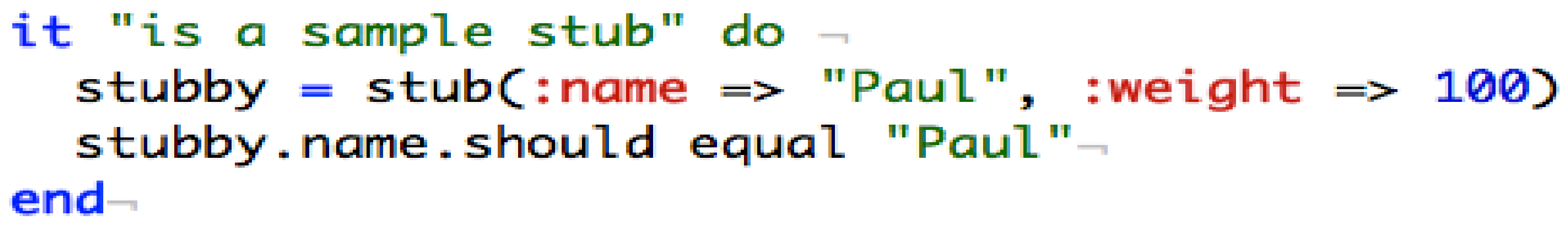
Agenda

Stubs

Replacement for one or many parts of an object

Normal method call is not happening

Returns a predefined value if called



```ruby
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

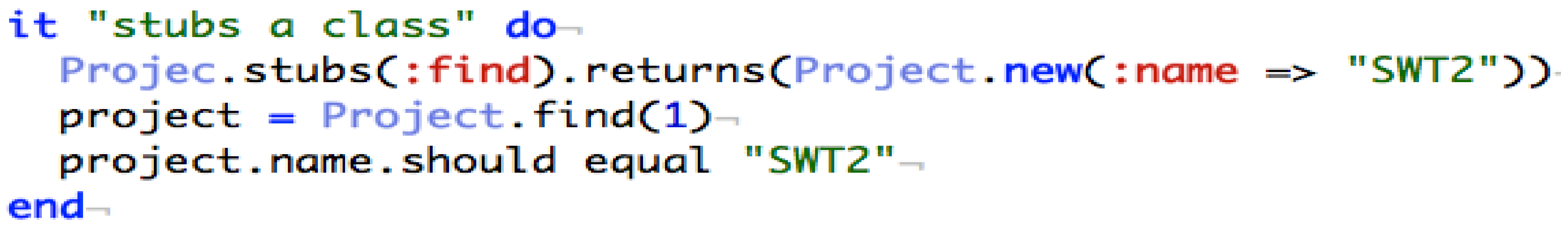
Else: error

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



```ruby
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

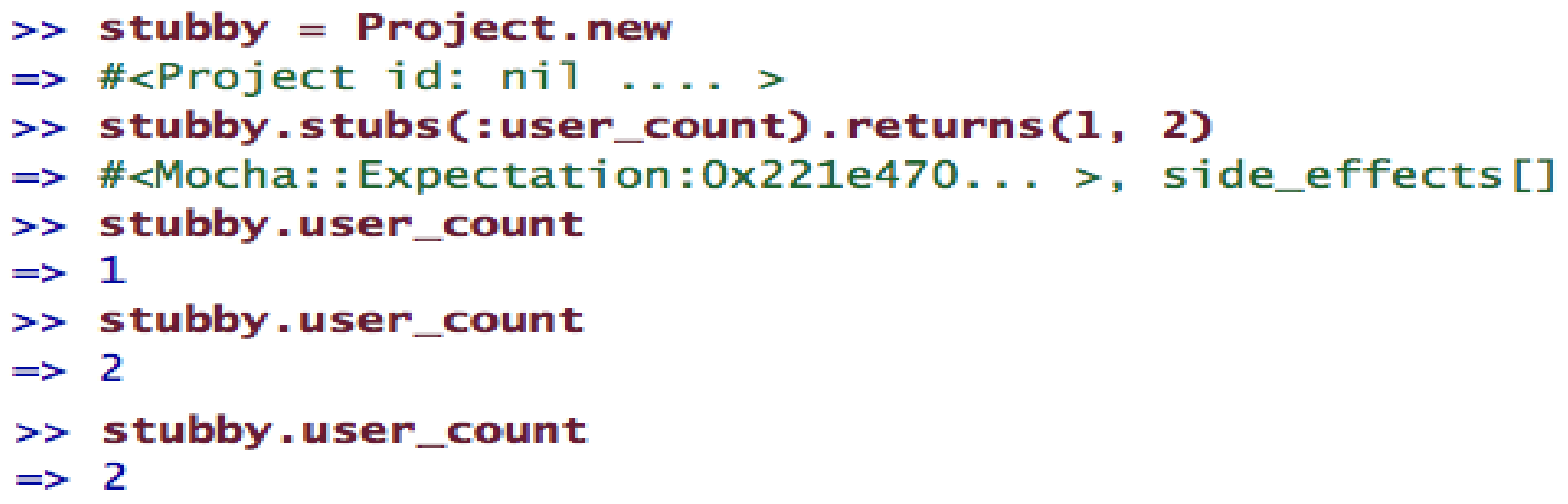
Database is not touched

“find” cannot be verified anymore BUT

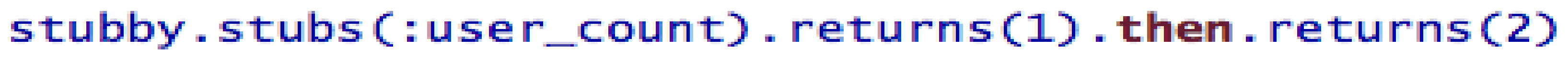
Tests based on “find” can be isolated

➔ 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
```

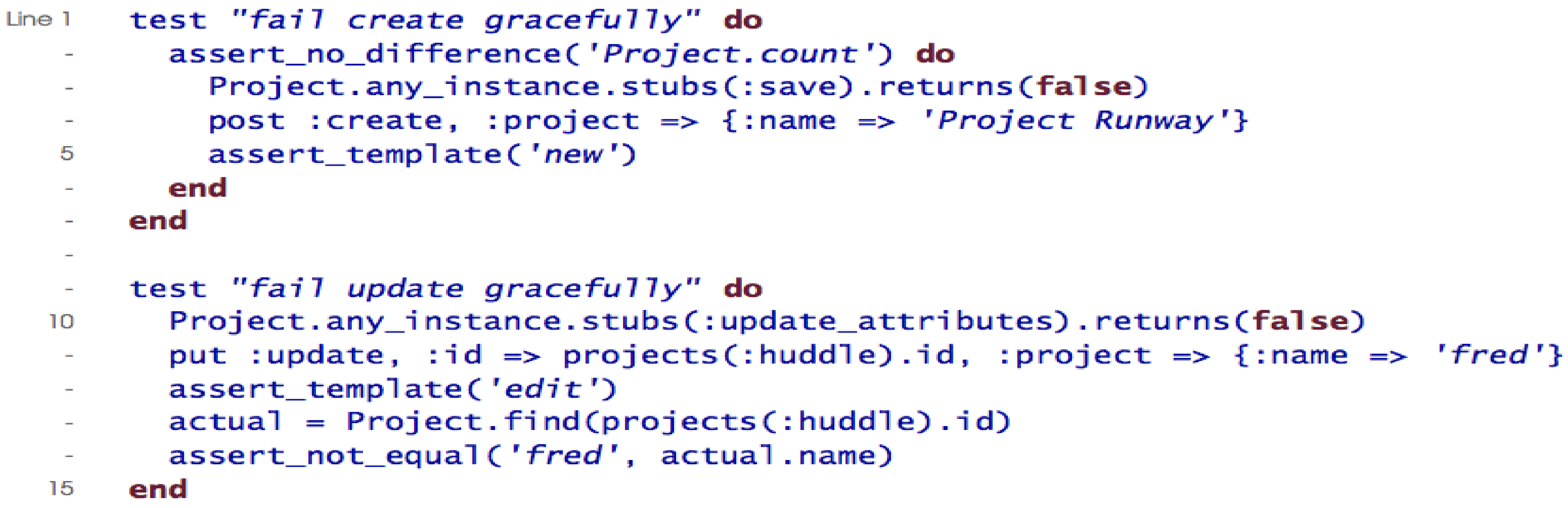


```
stubby.stubs(:user_count).returns(1).then.returns(2)
```

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 & Hints



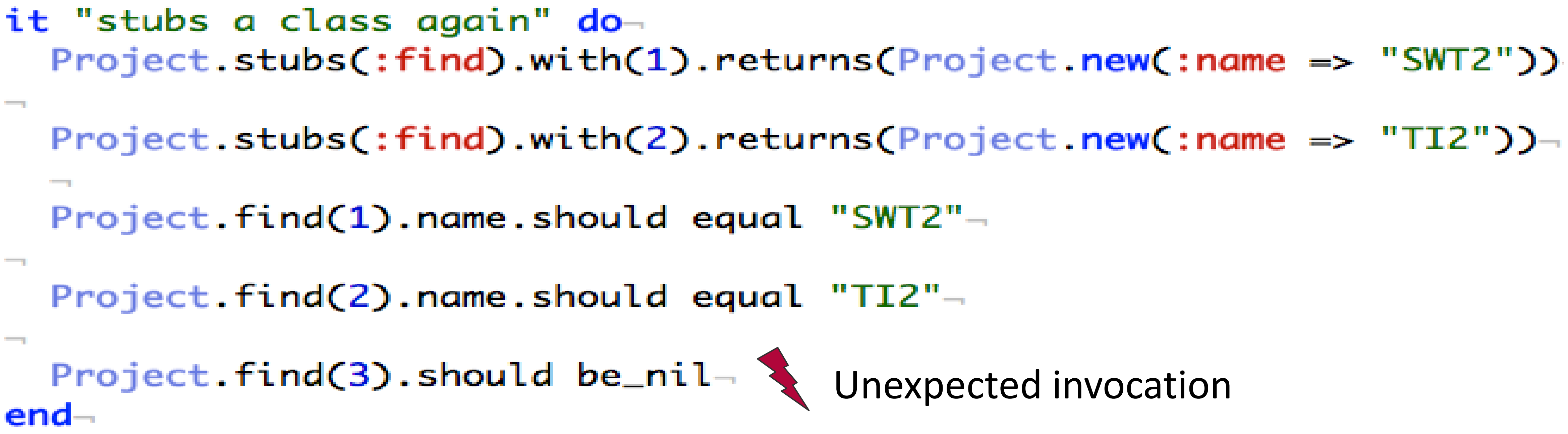
```
test "fail create gracefully" do
  assert_no_difference('Project.count') do
    Project.any_instance.stubs(:save).returns(false)
    post :create, :project => {:name => 'Project Runway'}
    assert_template('new')
  end
end

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
```

- 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)

Hints for any_instance

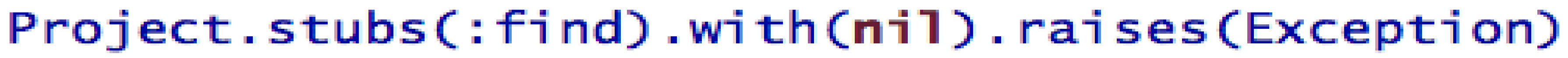
- No guarantee that find returns the exact object you expect
- any_instance is valid only for instances created after you declared the stub (not for fixture data)

Stubs with Parameters (with())

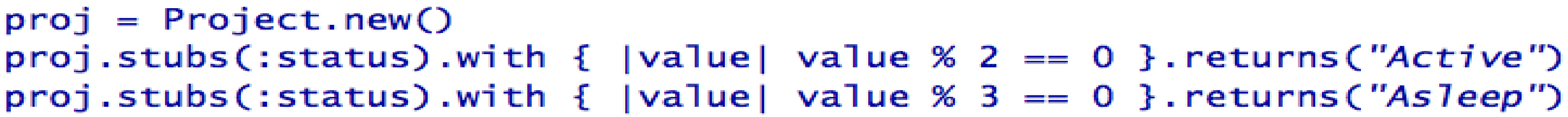


```
it "stubs a class again" do
  Project.stubs(:find).with(1).returns(Project.new(:name => "SWT2"))

  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
```

Unexpected invocation



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



```
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, any_of(), and regexp_matches()

```
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)
```

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

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

```
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

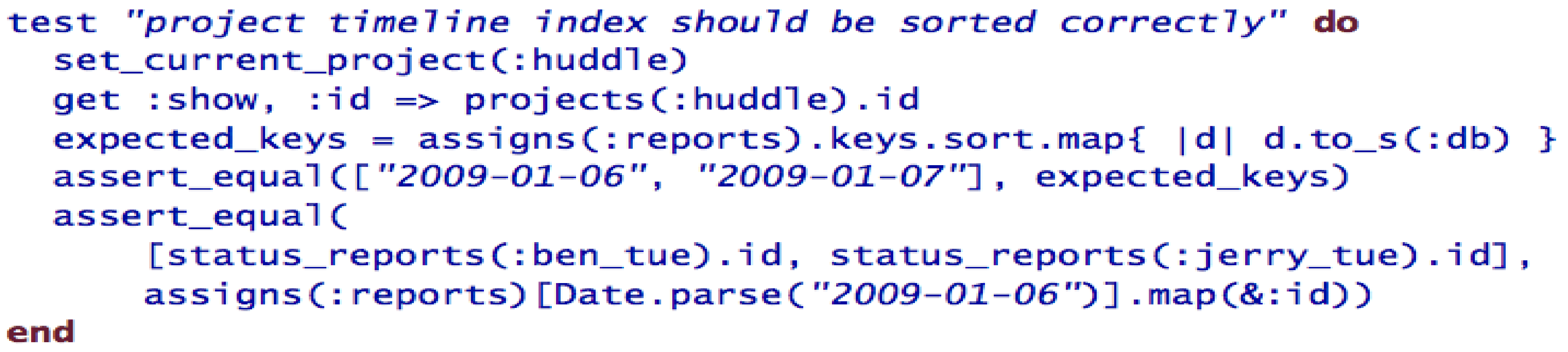
- Mock = Stub + attitude
- Demands that mock parameters are called (default: once)

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

-
  ```
  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
  ```

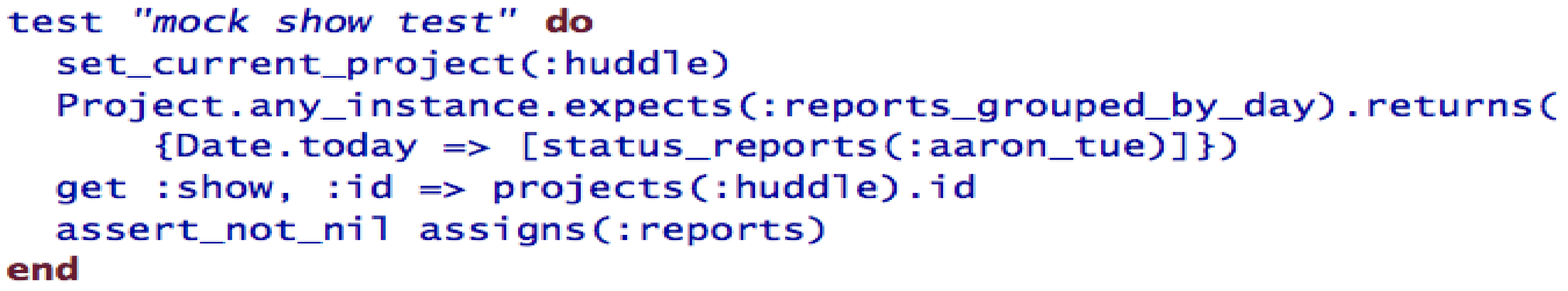
Mock Objects and Behavior-Driven Development

- 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⇧
  - ➔ test too large? Poor object-oriented design?

- Don't assert a value you set by a test double (false positives)

Agenda

Setup and Teardown
Rspec - before(:each)

```
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
```

- https://www.relishapp.com/rspec/rspec-core/v/2-0/docs/hooks/before-and-after-hooks

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 Framwork + test data + setup/teardown + test logic + additional assertions
  - Easiest tests to write

Hints for Model Tests

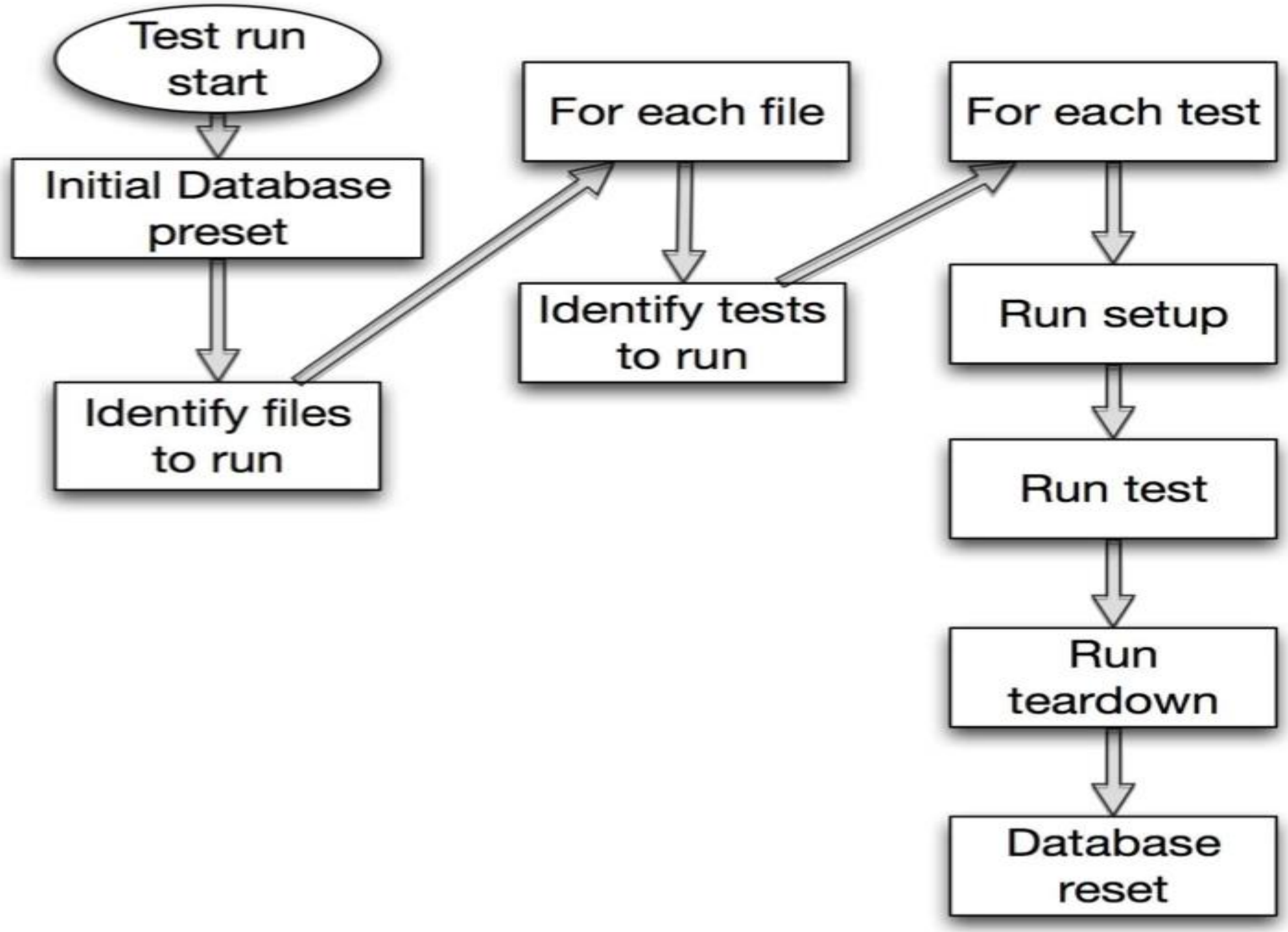
- Tests should cover ~100% of the model code
- Do not test framework functionality like “belongs_to”
- Test your validations
- How many tests? Let tests drive the code → perfect fit
- What comes out?
  - One test for the “happy-path case”
  - One test for each branch
  - Corner cases (nil, wrong values,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

Automate the process with Autotest

- Automate testing with Autotest (https://github.com/rspec/rspec/wiki/autotest)
- Run by using: autotest –rails
- Use FSEvent to determine file changes
- Automatically determines which tests to run again (remember: *Convention over Configuration*)

- Can be integrated with Growl on Macs ☺

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

View Tests

- Specify and verify logical and semantic structure

- Goals
  - Validate that view layer runs without error
  - Check that data gathered by the controller is presented as expected
    - message when passing empty collections
    - pagination upon more than x elements
    - …
  - Validate security-based output (e.g., for admins)

- Do not
  - Validate HTML markup
  - Evaluate look & feel
  - Test actual text

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]

- 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/ORM calls
    - that data is forwarded to view
  - Handling of invalid user requests
  - Handling of exceptions potentially raised by model calls
  - Verifying security roles / role-based access control

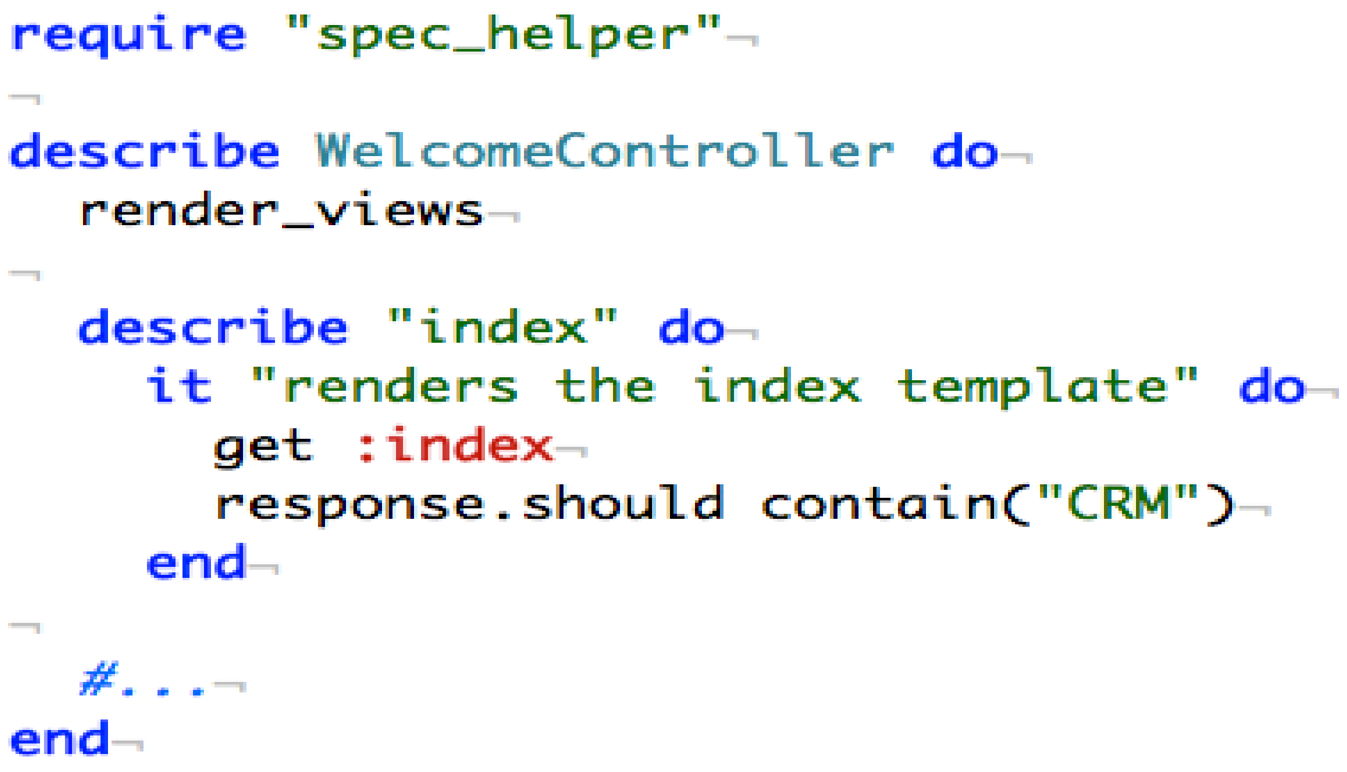
Background on Controller Tests

- 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

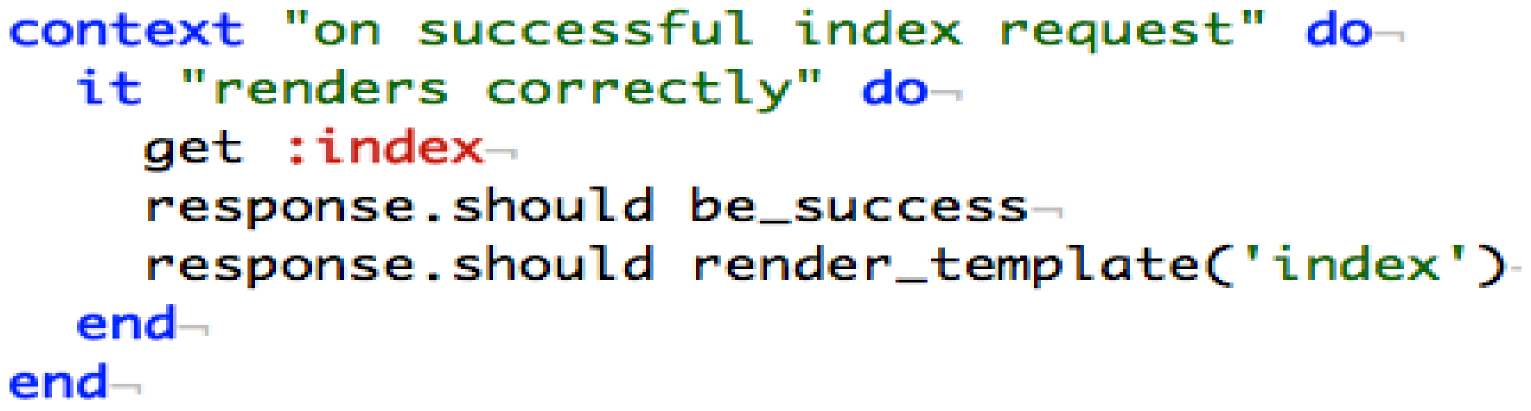
- By default, views are not rendered



```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

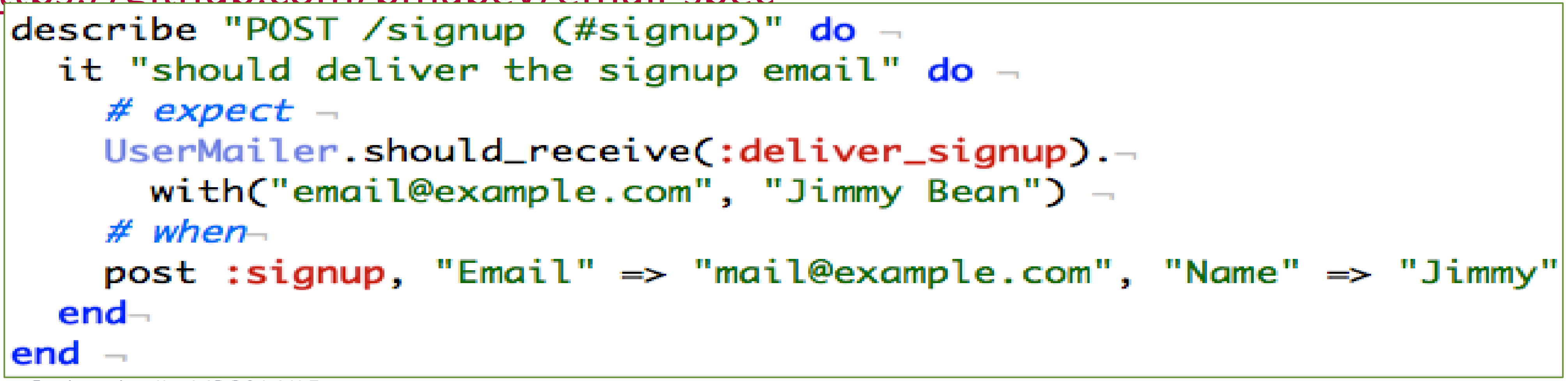
```
params_from(:get, "/hello/world").should ==
  {:controller => "hello", :action => "world"}
```

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

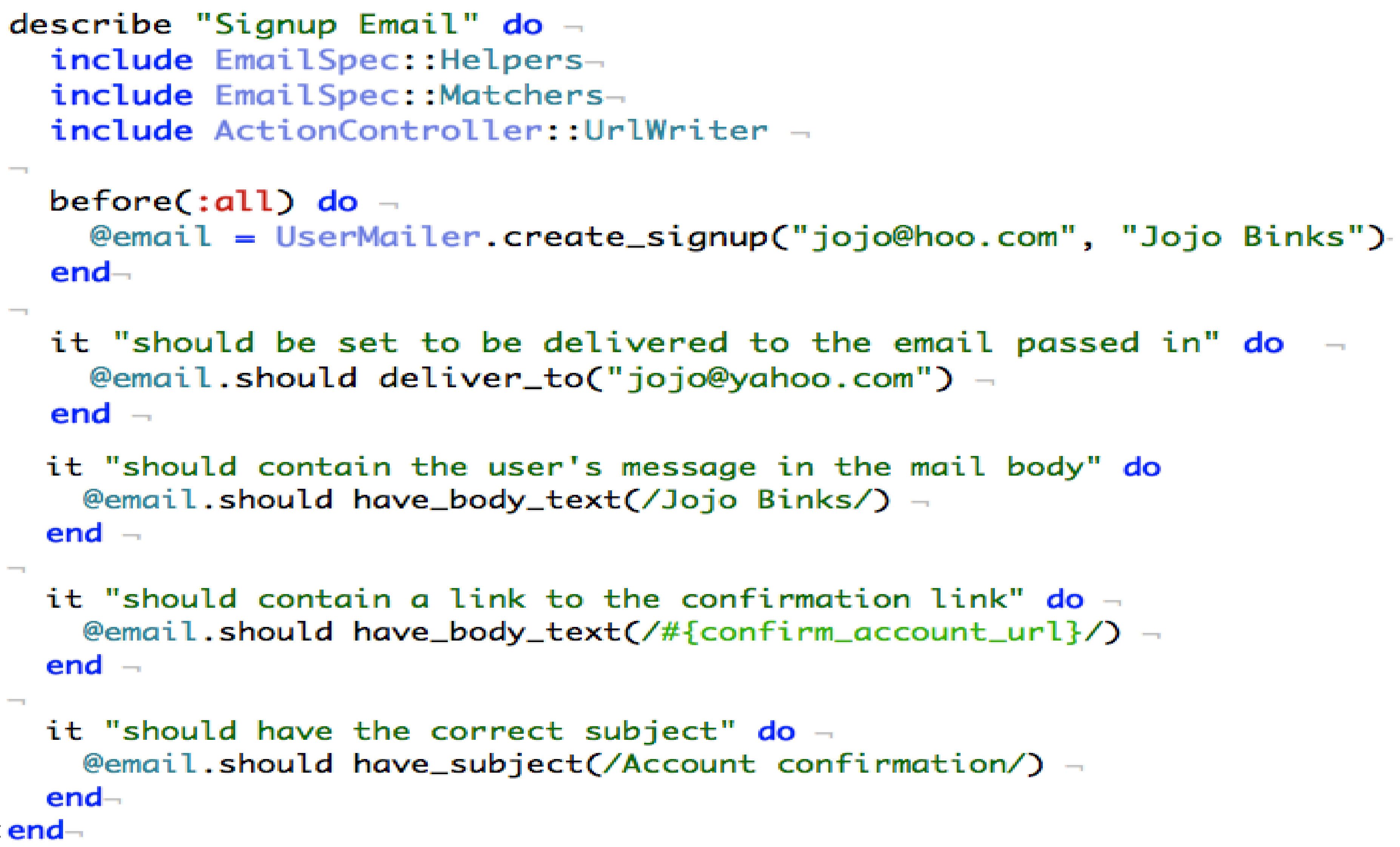
Outgoing Mail Tests

- What to validate?
  - Application sends mail when expected
  - Email content is what you expect
- Enable testing
  - Specs for content will be generated along with “rails g mailer”
  - For convenience matchers use email-spec gem https://github.com/bmabey/email-spec



```
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 Testing Mail Content and Metadata



```
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

  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

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

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

- Test communication of controllers via sessions/cookies
- Verify end-to-end behavior
- Make controller calls
- Verify that expected application states are created

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

Test::Unit

```ruby
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
```

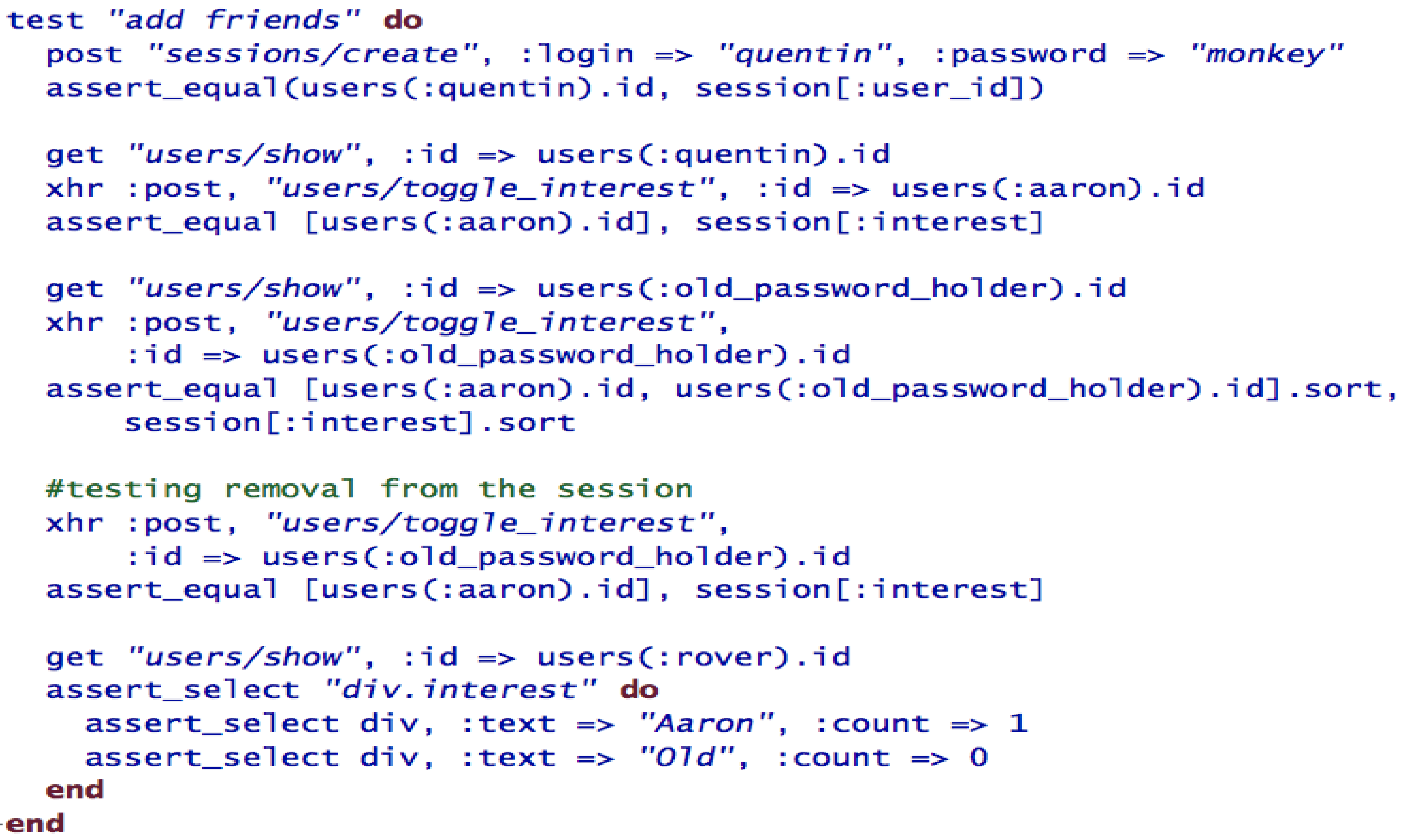
Multiple Session Example with Test::Unit

```
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

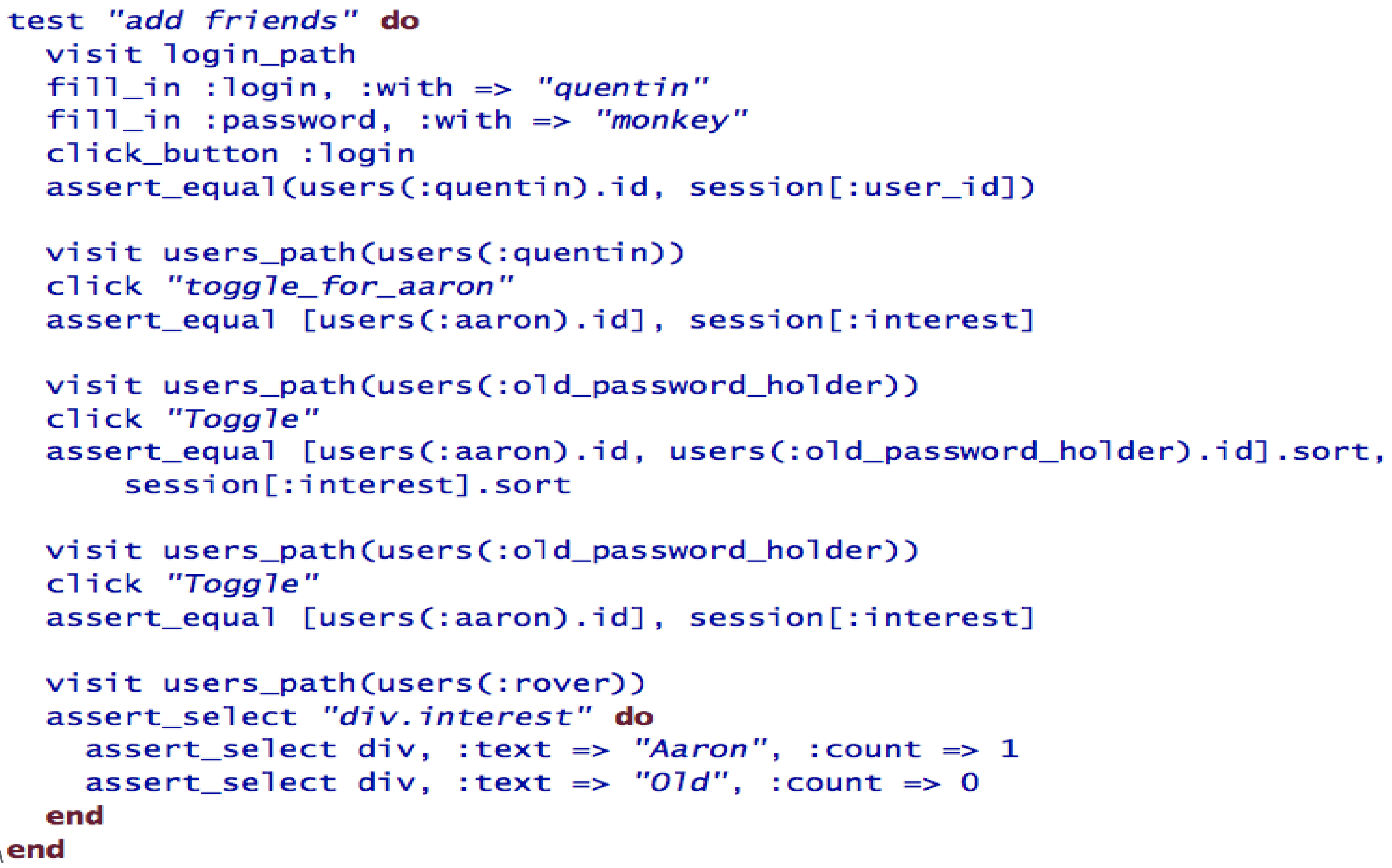
- DSLs for
  - "Browsing the Internet"
  - Acceptance testing
- 10 Useful Methods
  - attach_file(field_locator, path, content_type = nil)
  - check(field_locator)
  - choose(field_locator)
  - click_button(value)
  - click_link(text_or_title_or_id, options = {})
  - fill_in(field_locator, options = {})
  - save_and_open_page()
  - select(option_text, options = {})
  - uncheck(field_locator)
  - visit(url = nil, http_method = :get, data = {})

Capybara improves clarity (1/2)



```
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
```

Capybara improves clarity (2/2)



```
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
```

Capybara and Javascript (Rspec & Cucumber)

```ruby
describe "when current_user is the comment's author", js: true do
  it 'should edit the comment content' do
    visit post_path(commented_post)
    within ("#comment-#{commented_post.comments.first.id}") do
      click_on "edit"
    end
    fill_in 'comment_content', with: 'No, this is the best comment'
    click_on 'Edit Comment'
    expect(page).to have_content('No, this is the best comment')
  end
end
```

- Choses different capybara driver (e.g., selenium or phantomJS)
- Waiting period for Ajax Calls can be customised

```gherkin
@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

Test 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 line coverage indicates a potential problem
2. Existence of line coverage means very little
3. In combination with good testing practices, coverage might say something about test suite reach
4. ~100% test coverage is a by product of BDD

How to Measure Coverage?

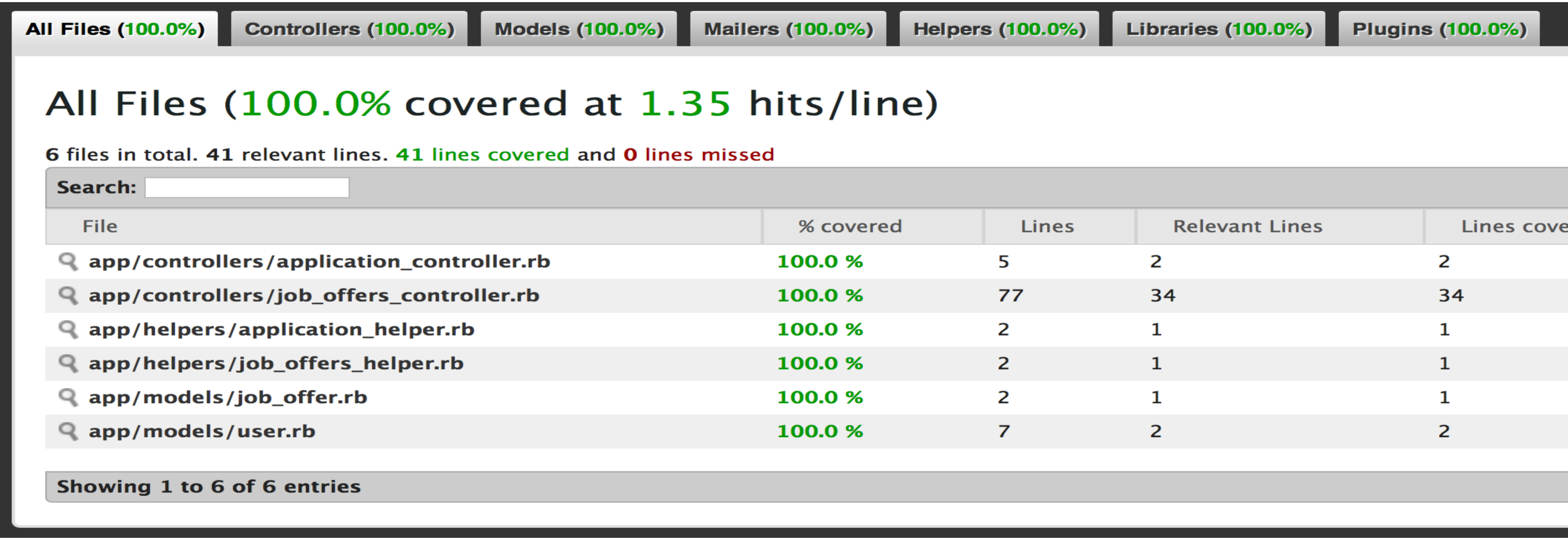
- Most useful approaches
  - Line coverage
  - Branch coverage

- 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



All Files (100.0% covered at 1.35 hits/line)

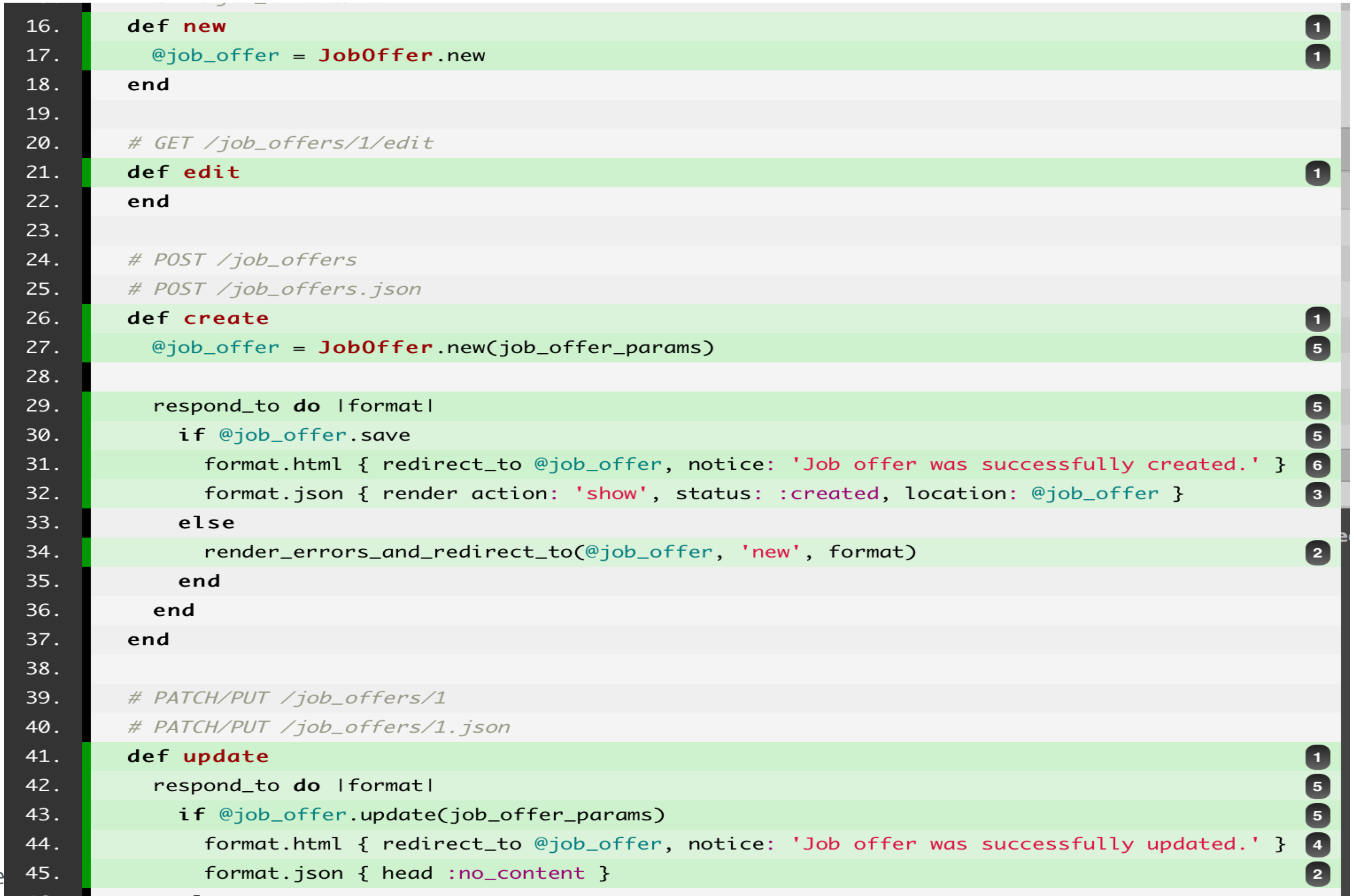
6 files in total. 41 relevant lines. 41 lines covered and 0 lines missed

Search:

File	% covered	Lines	Relevant Lines	Lines cove
app/controllers/application_controller.rb	100.0 %	5	2	2
app/controllers/job_offers_controller.rb	100.0 %	77	34	34
app/helpers/application_helper.rb	100.0 %	2	1	1
app/helpers/job_offers_helper.rb	100.0 %	2	1	1
app/models/job_offer.rb	100.0 %	2	1	1
app/models/user.rb	100.0 %	7	2	2

Showing 1 to 6 of 6 entries

Rcov / SimpleCov



```
  def new
    @job_offer = JobOffer.new
  end

  # GET /job_offers/1/edit
  def edit
  end

  # POST /job_offers
  # POST /job_offers.json
  def create
    @job_offer = JobOffer.new(job_offer_params)

    respond_to do |format|
      if @job_offer.save
        format.html { redirect_to @job_offer, notice: 'Job offer was successfully created.' }
        format.json { render action: 'show', status: :created, location: @job_offer }
      else
        render_errors_and_redirect_to(@job_offer, 'new', format)
      end
    end
  end

  # PATCH/PUT /job_offers/1
  # PATCH/PUT /job_offers/1.json
  def update
    respond_to do |format|
      if @job_offer.update(job_offer_params)
        format.html { redirect_to @job_offer, notice: 'Job offer was successfully updated.' }
        format.json { head :no_content }
```

5 Habits of Highly Successful Tests

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

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

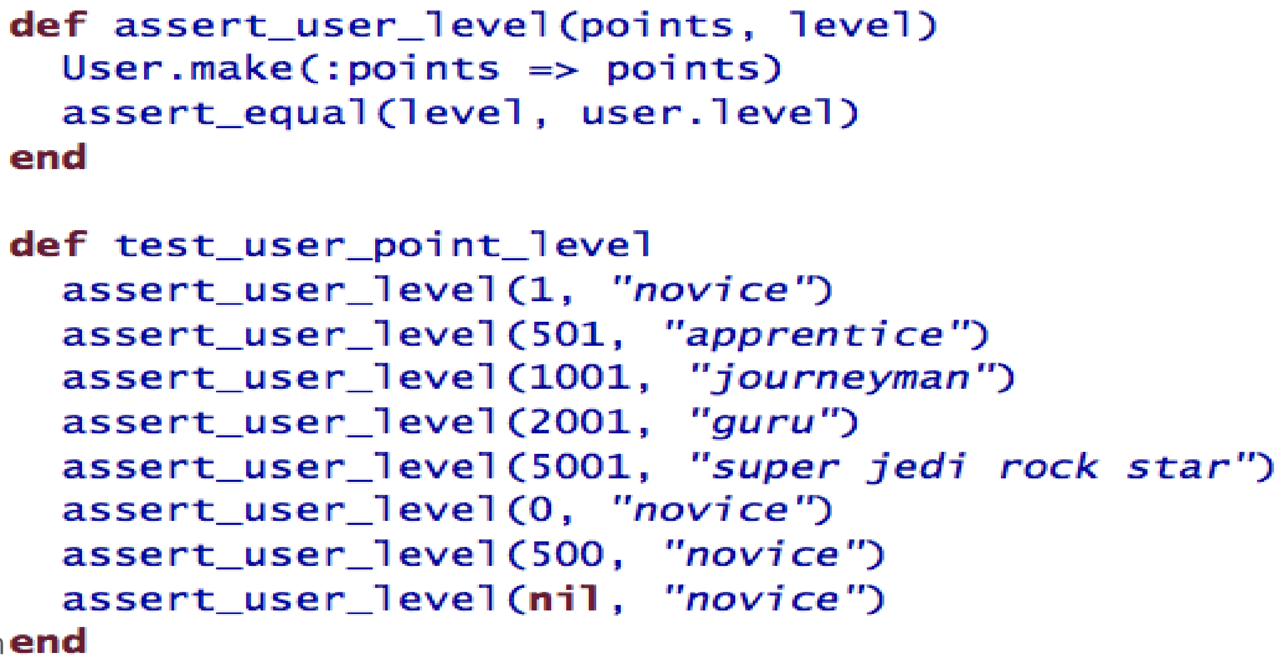
- Clarity
  - …
  - Better:

```
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!

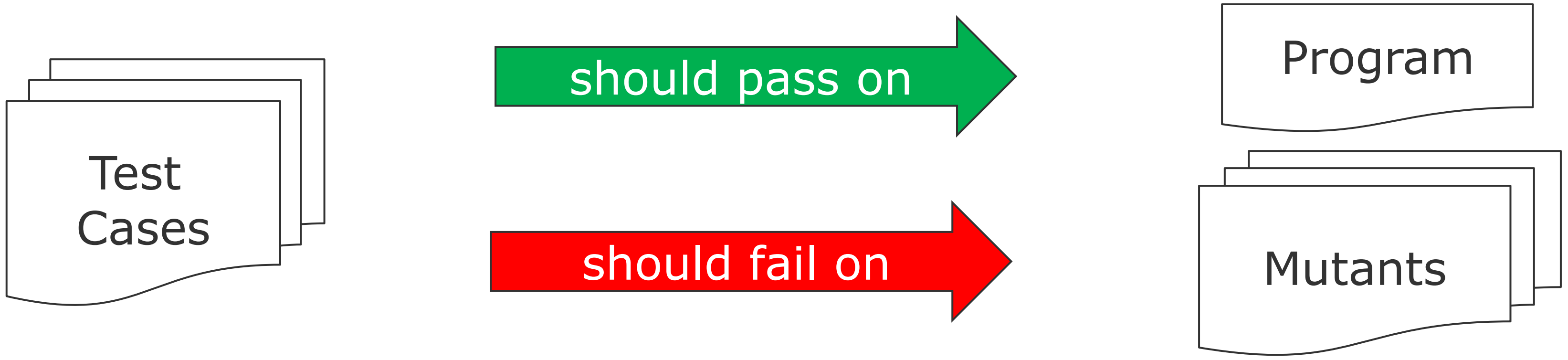
Mutation Testing

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
=	➔ <



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 (Nov 22, 1st slot)

- Retrospective Sprint #1
- Code Review Techniques
- Scrum Tips & Tricks