Behavior-driven Development and Testing in Ruby on Rails

Arian Treffer
arian.treffer@hpi.de
Prof. Plattner, Dr. Uflacker
Enterprise Platform and Integration Concepts group

Software Engineering II
WS 2016/17

Agenda

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

Agenda

1. Why Behavior-driven Development (BDD)?
   - Goals of Automated Testing
   - 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 05.00: working test Minute 10.00: 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 14: tested (3 times)	Minute 12.30: test ready 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 16.00: tested (3 times)
Minute 17: refactoring ready	*Minute 19.00: refactoring ready*
Minute 19: tested feature 1	Minute 19.10: tested both features
Minute 21: tested feature 2	Minute 20.10: committed
Minute 22: committed	

Goals of Automated Testing

- Find errors faster
- Better code (correct, robust, maintainable)
- Less overhead when testing à tests 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

è We'll cover a bit of this in this lecture

Agenda

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

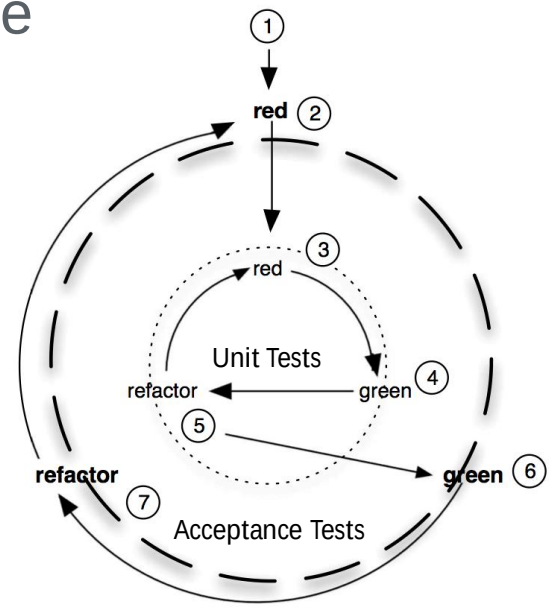
Writing Software that Matters

> *"BDD is about implementing an application by describing its behavior from the perspective of its stakeholders"*
>
> – Dan North

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]

Definition of Done

How do I know when to stop?

- Acceptance criteria fulfilled
- All tests are green
- Code looks good
- Objective quality goals
- Second opinion
- Internationalization
- Security
- Documentation

The Definition of Done is the team's consensus of what it takes to complete a feature.

Maximum BDD Pyramid

Vision

All Stakeholders, one statement

- *Example:* Improve Supply Chain; Understand Customers Better

Core stakeholders have to define the vision

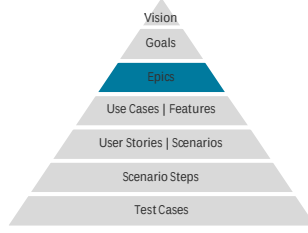
- Incidental stakeholders help understand
  - What is possible
  - At what cost
  - With what likelihood

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 Cases / 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
  - à Not too much detail
- Features deliver value to stakeholders

User Stories

- Stories are demonstrable functionality
- 1 Feature à 1..n User Stories
- Stories should be vertical (e.g. no database-only stories)
- User stories are a token for conversations
- Attributes (INVEST)
  - Independent
  - Negotiable
  - Valuable (from a business Point of View)
  - Estimable
  - Small enough to be implemented in one iteration
  - Testable

User Stories

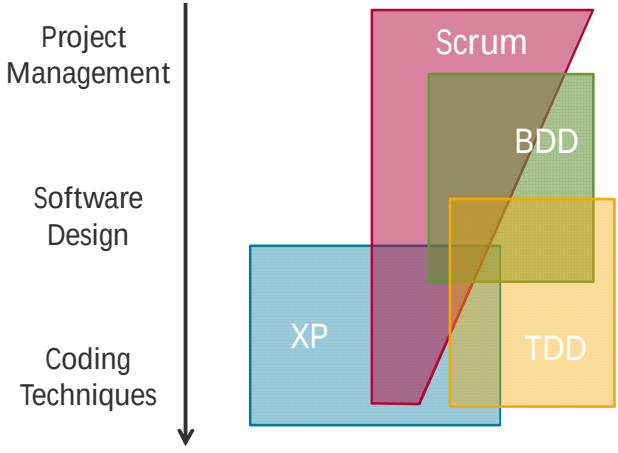
- Story content
  - Title
  - Narrative
    - Description, reason, benefit
    - “As a <stakeholder>, I want <feature> so that <benefit>”
    - “In order to <benefit>, a <stakeholder> wants to <feature>”
  - Acceptance criteria

Scenarios, Scenario Steps, Test Cases

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

- 1 scenario à m scenario steps + step implementation
- 1 scenario step à 0..i tests
- Describe low-level behavior

Agile Methodologies



Behavior-driven Development

Principles

- Story-based definition of application behavior
- Definition of features
- Driven by business value

For the developer

- BDD Cycle
- Coding with TDD
- Automated Testing

Agenda

1. Why Behavior-driven Design (BDD)?
2. Building Blocks of Tests and BDD
   - Model Tests
   - View Tests
   - Controller Tests
   - Setup and Teardown
   - Test Data
   - Test Doubles
   - Integration & Acceptance Tests
   - Specialized Tests
3. Testing Tests & Hints for Successful Test Design
4. Outlook

Test::Unit vs. RSpec

- Test::Unit comes with Ruby

```
class UserTest < Test::Unit::TestCase

  def test_first_name
    user = User.new
    assert_nil user.name, "User's name was not nil."
    user.name = "Chuck Norris"
    assert_equal user.first_name, "Chuck", "user.first_name did not return 'Chuck'."
  end

end
```

Test::Unit vs. RSpec

- RSpec offers syntactical sugar, different structure
- Many “built-in” modules (e.g. mocking)
- “`rspec`” command with tools to constrain what examples are run

```
describe User do

  it "should determine first name from name" do
    user = User.new
    expect(user.name).to be_nil
    user.name = "Chuck Norris"
    expect(user.first_name).to eq "Chuck"
  end

end
```

è We’ll use RSpec

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

RSpec Basic structure

- Using "*describe*" and "*it*" like in a conversation
  - *"Describe an order!" "It sums prices of items."*

- *describe* creates a test / example group
- *it* declares examples within group
- *context* for nested groups / structuring

- Aliases
  - Declare example groups using *describe* or *context*
  - Declare examples using *it*, *specify*, or *example*

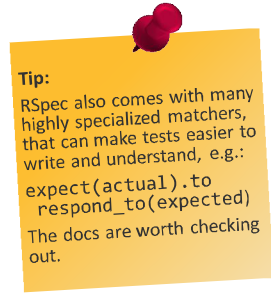
```
describe Order do
  context "with one item" do
    it "sums prices of items" do
      # ...
    end
  end

 context "with no items" do
    it "shows a warning" do
      # ...
    end
  end
end
```

- https://github.com/rspec/rspec-core/blob/master/README.md

RSpec Matchers

- General structure of RSpec expectation (assertion):
  - `expect(…).to <matcher>, expect(…).not_to <matcher>`

```
# Object identity
expect(actual).to be(expected) # passes if actual.equal?(expected)
# Object equivalence
expect(actual).to eq(expected) # passes if actual == expected
# Comparisons
expect(actual).to be >= expected
expect(actual).to be_between(minimum, maximum).inclusive
expect(actual).to match(/expression/) # regular expression
expect(actual).to start_with expected
# Collections
expect([]).to be_empty
expect(actual).to include(expected)
```



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

Agenda

1. Why Behavior-driven Design (BDD)?
2. Building Blocks of Tests and BDD
   - Model Tests
   - View Tests
   - Controller Tests
   - Setup and Teardown
   - Test Data
   - Test Doubles
   - Integration & Acceptance Tests
   - Specialized Tests
3. Testing Tests & Hints for Successful Test Design
4. Outlook

Model Tests

- A Rails model
  - Accesses data through an ORM
  - Implements business logic
  - Is “fat”
- Model tests in Rails
  - Easiest tests to write
  - Test most of application logic

Hints for Model Tests

- Tests should cover circa 100% of the model code
- Do not test framework functionality like "*belongs_to*"
- Test your validations
- How many tests? Let tests drive the code à perfect fit

- Minimal test set:
  - One test for the "happy-path case"
  - One test for each branch
  - Corner cases (nil, wrong values, …), if appropriate
- Keep each test small!

Model Test Example

app/models/contact.rb

```ruby
class Contact < ActiveRecord::Base
  validates :name, presence: true

  def self.by_letter(letter)
    where("name LIKE ?", "#{letter}%").order(:name)
  end
end
```

spec/models/contact_spec.rb

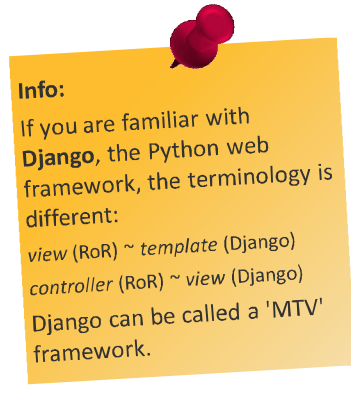
```ruby
require 'rails_helper'

describe Contact, :type => :model do

  before :each do #do this before each test
    @john= Contact.create(name: 'John')
    @tim = Contact.create(name: 'Tim')
    @jerry = Contact.create(name: 'Jerry')
  end

  #the actual test cases
  context "with matching letters" do
    it "returns a sorted array of results that match" do
      expect(Contact.by_letter("J")).to eq [@john, @jerry]
    end

    it "omits results that do not match" do
      expect(Contact.by_letter("J")).not_to include @tim
    end
  end

end
```

Agenda

1. Why Behavior-driven Design (BDD)?
2. Building Blocks of Tests and BDD
   - Model Tests
   - View Tests
   - Controller Tests
   - Setup and Teardown
   - Test Data
   - Test Doubles
   - Integration & Acceptance Tests
   - Specialized Tests
3. Testing Tests & Hints for Successful Test Design
4. Outlook

View Tests

- A Rails view
  - Has only minimal logic
  - Never calls the database!
  - Presents the data passed by the controller
- Challenges for view tests
  - Time-intensive
  - How to test look & feel?
  - Brittle with regard to interface redesigns



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
    - Messages on passing empty collections to the view
    - Pagination on more than n elements
  - Validate security-based output, e.g. for admins

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

View Tests in RSpec

```
describe "users/index" do
  it "displays user name" do
    assign(:user,
        User.create! :name => "Bob"
    )

    # path could be inferred from test file
    render :template => "users/index.html.erb"

    expect(rendered).to match /Hello Bob/
  end
end
```

- https://www.relishapp.com/rspec/rspec-rails/v/3-2/docs/view-specs/view-spec

View Tests in RSpec (with Capybara)

```
require 'capybara/rspec'

Rspec.describe "users/index" do
  it "displays user name" do
    assign(:user,
        User.create! :name => "Bob"
    )

    # path could be inferred from test file
    render :template => "users/index.html.erb"

    # same as before
    expect(rendered).to have_content('Hello Bob')
    # a better idea
    expect(rendered).to have_css('a#welcome')
    expect(rendered).to have_xpath('//table/tr')
  end
end
```

Tip:

For exploring in *irb*, using Capybara matchers on strings, use:

`Capybara.string`

robots.thoughtbot.com/use-capybara-on-any-html-fragment-or-page

Another Tip:

Capybara features a whole range of helpful "matchers", including `has_button`, `has_table`, `has_unchecked_field`.

rubydoc.info/github/jnicklas/capybara/master/Capybara/Node/Matchers

- https://github.com/jnicklas/capybara

Agenda

1. Why Behavior-driven Design (BDD)?
2. Building Blocks of Tests and BDD
   - Model Tests
   - View Tests
   - Controller Tests
   - Setup and Teardown
   - Test Data
   - Test Doubles
   - Integration & Acceptance Tests
   - Specialized Tests
3. Testing Tests & Hints for Successful Test Design
4. Outlook

Controller Tests

- A Rails controller
  - Is “skinny”
  - Calls the model
  - Passes data to the view
  - Responds with a rendered view

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

What to Test in Controller Tests?

- Verify that user requests trigger
  - Model / ORM calls
  - That data is correctly forwarded to view
- Verify handling of invalid user requests, e.g. redirects
- Verify handling of exceptions raised by model calls
- Verify security roles / role-based access control

Remember: Model functionality is tested in model tests!

Inside Controller Tests

Rails provides helpers to implement controller tests

- 3 important variables are automatically imported
  - `controller`
  - `request`
  - `response`
- Variable getter and setter for
  - session – `session[:key]`
  - controller variables – `assigns[:key]`
  - flash – `flash[:key]`
- Methods to simulate a single HTTP request
  - *get*, *post*, *put*, *delete*

Testing the Controller Response


```ruby
require "rails_helper"

describe TeamsController, :type => :controller do
  describe "GET index" do
    it "assigns @teams in the controller" do
      team = Team.create
      get :index
      expect(assigns(:teams)).to eq([team])
    end

    it "renders the index template" do
      get :index
      expect(response).to render_template("index")
    end
  end
end
```

- http://www.relishapp.com/rspec/rspec-rails/v/3-2/docs/controller-specs

Background on Controller Tests

- By default, views are not rendered

```
require "rails_helper"

describe WidgetsController, :type => :controller do
  render_views # explicitly render the view

  describe "GET index" do
    it "says 'Listing widgets'" do
      get :index
      expect(response.body).to match /Listing widgets/im
    end
  end
end
```

- http://www.relishapp.com/rspec/rspec-rails/v/3-2/docs/controller-specs/render-views

Agenda

1. Why Behavior-driven Design (BDD)?
2. Building Blocks of Tests and BDD
   - Model Tests
   - View Tests
   - Controller Tests
   - Setup and Teardown
   - Test Data
   - Test Doubles
   - Integration & Acceptance Tests
   - Specialized Tests
3. Testing Tests & Hints for Successful Test Design
4. Outlook

Setup and Teardown – RSpec

As a developer using RSpec
I want to execute arbitrary code before and after examples
So that I can control the environment in which tests are run

```
before(:example) # run before each example
before(:context) # run one time only, before all of the examples in a group

after(:example) # run after each example
after(:context) # run one time only, after all of the examples in a group
```

Setup RSpec – before(:example)

```ruby
require "rspec/expectations"

class Thing
  def widgets
    @widgets ||= []
  end
end

describe Thing do
  before(:example) do
    @thing = Thing.new
  end

  describe "initialized in before(:example)" do
    it "has 0 widgets" do
      expect(@thing.widgets.count).to eq(0)
    end
  end
end
```

- `before(:example)` blocks are run before each example
- `:example` scope is also available as `:each`

- https://www.relishapp.com/rspec/rspec-core/v/3-2/docs/hooks/before-and-after-hooks

Setup RSpec – before(:context)


```
require "rspec/expectations"
class Thing
  ... #as before

describe Thing do
  before(:context) do
    @thing = Thing.new
  end

  context "initialized in before(:context)" do
    it "can accept new widgets" do
      @thing.widgets << Object.new
    end

    it "shares state across examples" do
      expect(@thing.widgets.count).to eq(1)
    end
  end
end
```

- `before(:context)` blocks are run before all examples in a group
- `:context` scope is also available as `:all`
- Warning: Mocks are only supported in `before(:example)`

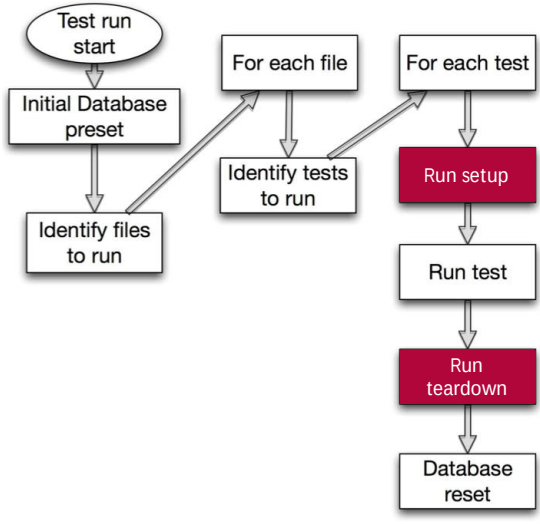
- https://www.relishapp.com/rspec/rspec-core/v/3-2/docs/hooks/before-and-after-hooks

Teardown RSpec


```ruby
describe "Test the website with a browser" do
  before(:context) do
    @browser = Watir::Browser.new
  end

  it "should visit a page" do
    ...
  end

  after(:context) do
    @browser.close
  end
end
```

- `after(:context)` blocks are run after all examples in a group
- For example to clean up

Test Run



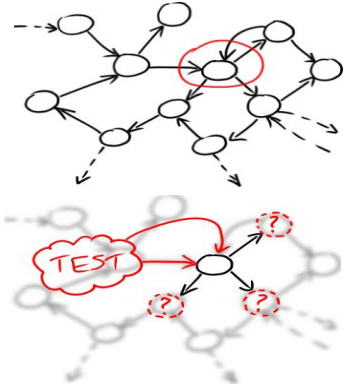
- Rails Test Prescriptions. Noel Rappin. 2010. p. 37. http://zepho.com/rails/books/rails-test-prescriptions.pdf

Agenda

1. Why Behavior-driven Design (BDD)?
2. Building Blocks of Tests and BDD
   - Model Tests
   - View Tests
   - Controller Tests
   - Setup and Teardown
   - Test Data
   - Test Doubles
   - Integration & Acceptance Tests
   - Specialized Tests
3. Testing Tests & Hints for Successful Test Design
4. Outlook

Isolation of Test Cases

- Tests should be independent
- If a bug in a model is introduced
  - 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 Data Overview

Two main ways to provide data to test cases:

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

Fixture Overview

- Fixtures represent sample data
- Populate testing database with predefined data before tests run
- Stored in database independent YAML files (.yml)
- One file per model, location: `test/fixtures/<name>.yml`

```
# test/fixtures/users.yml
david: # Each fixture has a name
  name: David Heinemeier Hansson
  birthday: 1979-10-15
  profession: Systems development

steve:
  name: Steve Ross Kellock
  birthday: 1974-09-27
  profession: guy with keyboard
```

Info:

By default, `test_helper.rb` (`require 'test_helper'`) will load all fixtures into the database.

To ensure consistent data, fixtures are deleted before loading.

Another Info:

Fixture data can be accessed by using a special dynamic method, with the same name as the model:

```
users(:steve).name
# => Steve Ross Kellock
```

- http://api.rubyonrails.org/classes/ActiveRecord/FixtureSet.html
- http://guides.rubyonrails.org/testing.html

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 à data for one test is spread out over many files
  - Tracing relationships is a pain
- Fixtures are distant
  - Fixture data is not immediately available in the test
  - `expect(users(:ernie).age + users(:bert).age).to eq(20)`
- Fixtures are brittle
  - Tests rely on fixture data, they break when data is changed
  - Data requirements of tests 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 specify; even for complex data sets
- Robust
  - Independent from other tests

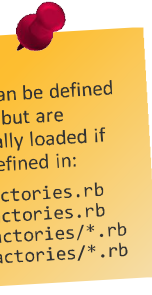
è Our choice to achieve this: Data factories

Data Factories

- Blueprint for sample instances
- Rails tool support
  - Factory Girl (our choice)
  - Machinist
  - Fabrication
  - FixtureBuilder
  - Cf. 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
```



- http://www.rubydoc.info/gems/factory_girl/file/GETTING_STARTED.md

Using Factories

- Build strategies: build, create (standard), attributes_for, build_stubbed

```
# Returns a User instance that's _not_ saved
user = build(:user)

# Returns a _saved_ User instance
user = create(:user)

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

# Passing a block to any of the methods above will yield the return object
create(:user) do |user|
  user.posts.create(attributes_for(:post))
end
```

- http://www.rubydoc.info/gems/factory_girl/file/GETTING_STARTED.md

Attributes


```ruby
# Lazy attributes
factory :user do
  activation_code { User.generate_activation_code }
  date_of_birth { 21.years.ago }
end

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

# override the defined attributes by passing a hash
create(:user, last_name: "Doe").email
# => "joe.doe@example.com"
```

- http://www.rubydoc.info/gems/factory_girl/file/GETTING_STARTED.md

Associations


```
factory :post do
  # If factory name == association name, the factory name can be left out.
  author
End
```

```
factory :post do
  # specify a different factory or override attributes
  association :author, factory: :user, last_name: "Writely“
End
```

```
# Builds and saves a User and a Post
post = create(: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 = build(:post)
post.new_record?             # => true
post.author.new_record?      # => false
```

- http://www.rubydoc.info/gems/factory_girl/file/GETTING_STARTED.md

Inheritance


```ruby
# The title attribute is required for all posts
factory :post do
  title "A title"
End

# An approved post includes an extra field
factory :approved_post, parent: :post do
  approved true
end
```

- http://www.rubydoc.info/gems/factory_girl/file/GETTING_STARTED.md

Sequences for Unique Values


```ruby
# Defines a new sequence
FactoryGirl.define do
  sequence :email do |n|
    "person#{n}@example.com"
  end
end

generate :email # => "person1@example.com"
generate :email # => "person2@example.com"

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

```ruby
# in lazy attribute
factory :invite do
  invitee { generate(:email) }
end

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

- http://www.rubydoc.info/gems/factory_girl/file/GETTING_STARTED.md

Callbacks

factory_girl makes four callbacks available for injecting code:

- *after*(:build)- called after the object is built (via `FactoryGirl.build, FactoryGirl.create`)
- *before*(:create) - called before the object is saved (via `FactoryGirl.create`)
- *after*(:create) - called after the object is saved (via `FactoryGirl.create`)
- *after*(:stub) - called after the object is stubbed (via `FactoryGirl.build_stubbed`)

```
# Call customize() after the user is built
factory :user do
  after(:build) { |user| customize(user) }
end

# multiple types of callbacks on the same factory
factory :user do
  after(:build) { |user| customize(user) }
  after(:create) { |user| customize_further(user) }
end
```

- http://www.rubydoc.info/gems/factory_girl/file/GETTING_STARTED.md

Factory Girl – Further Reading

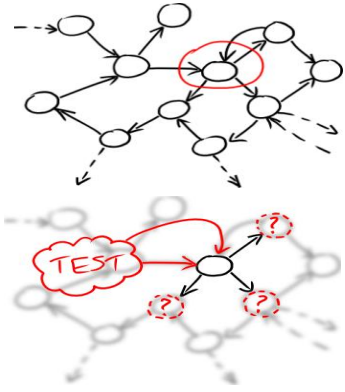
- Faster tests with `build_stubbed`
  - https://robots.thoughtbot.com/use-factory-girls-build-stubbed-for-a-faster-test
- Tips and tricks
  - http://arjanvandergaag.nl/blog/factory_girl_tips.html

Agenda

1. Why Behavior-driven Design (BDD)?
2. Building Blocks of Tests and BDD
   - Model Tests
   - View Tests
   - Controller Tests
   - Setup and Teardown
   - Test Data
   - Test Doubles
   - Integration & Acceptance Tests
   - Specialized Tests
3. Testing Tests & Hints for Successful Test Design
4. Outlook

Isolation of Test Cases

- Tests should be independent
- If a bug in a model is introduced
  - 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

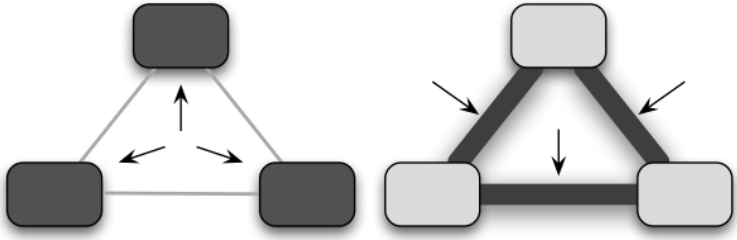
Test Doubles

- Generic term for object that stands in for a real object during a test
  - Think “stunt double”
- 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
  - Only the result of a call is tested, intermediate steps are not considered
- With test doubles: Test system behavior
  - E.g. How often a method is called, in which order, with which parameters



Ruby Test Double Frameworks

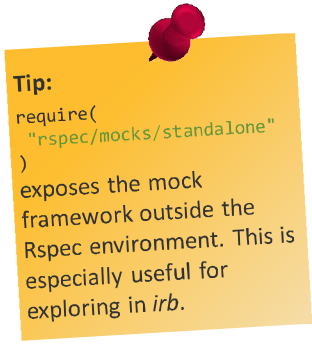
Many frameworks available:

- RSpec-mocks (http://github.com/rspec/rspec-mocks)
- Mocha (https://github.com/freerange/mocha)
- FlexMock (https://github.com/jimweirich/flexmock)

A collection of mocking frameworks (as well as many others):

- https://www.ruby-toolbox.com/categories/mocking

è We recommend RSpec-Mocks as it shares a common syntax with RSpec



Stubs

- Method call on the real object does not happen
- Returns a predefined value if called
- Strict by default (error when messages received that have not been allowed)

```
dbl = double(“user”)
allow(dbl).to receive_messages (:name => “Fred”, :age => 21 )
expect (dbl.name).to eq(“Fred”) #this is not really a good test :)
dbl.height #raises error (even if your original object had that property)
```

- Alternatively, if all method calls should succeed: Null object double

```
dbl = double(“user”).as_null_object
dbl.height # this is ok! Returns itself (dbl)
```

- http://www.relishapp.com/rspec/rspec-mocks/v/3-2/docs/basics/null-object-doubles

Spies

- Stubs with *Given-When-Then* structure
- Allows to expect that a message has been received after the message call

```
dbl = spy("user")
dbl.height
dbl.height
expect(dbl).to have_received(:height).at_least(2).times
```

- Alternatively, spy on specific messages of real objects

```
user = User.new
allow(user).to receive(:height)        # Given a user
user.measure_size                      # When I measure the size
expect(user).to have_received(:height) # Then height is called
```

- http://www.relishapp.com/rspec/rspec-mocks/v/3-2/docs/basics/spies

Mocks

- Mocks are Stubs with attitude
- Demands that mocked methods are called

```
book = double("book", :title => "The RSpec Book")
expect(book).to receive(:open).once # 'once' is default
book.open # this works
book.open # this fails
```

- Or as often as desired

```
user = double("user")
expect(user).to receive(:email).exactly(3).times
expect(user).to receive(:level_up).at_least(4).times
expect(user).to receive(:notify).at_most(3).times
```

- If test ends with expected calls missing, it fails!

- https://relishapp.com/rspec/rspec-mocks/v/3-2/docs/configuring-responses/returning-a-value

Stubs vs. Mocks

Stub (passive)

- Returns a predetermined value for a method call

```
dbl = double("a user")
allow(dbl).to receive (:name) => { "Fred" }
expect (dbl.name).to eq("Fred") #this is not really a good test :)
```

Mock (more aggressive)

- In addition to stubbing: set a “message expectation”
- If expectation is not met, i.e. the method is not called à test failure

```
dbl = double(“a user”)
expect(dbl).to receive(:name)
dbl.name     #without this call the test would fail
```

è Stubs don‘t fail your tests, mocks can!

Info:
In **RSpec** the *allow* keyword refers to a stub, *expect* to a mock.
This might vary by framework.

Partially Stubbing Instances

- Sometimes you want only part of your object to be stubbed
  - Many methods on object, only expensive ones need stubbing for a test
- Extension of a real object in a system that is instrumented with stub like behaviour
- “Partial test double” (in RSpec terminology)

```
s = "a user name" # s.length == 11
allow(s).to receive(:length).and_return(9001)
expect (s.length).to eq(9001) # the method was stubbed
s.capitalize! # this still works, only length was stubbed
```

- http://www.relishapp.com/rspec/rspec-mocks/v/3-2/docs/basics/partial-test-doubles

Class Methods

- Class methods can also be stubbed
- Example: Stubbing the User class
  - The database is not touched, a specific instance is returned
  - “find” cannot be verified anymore but tests based on “find” can be isolated
  - à just test the logic that is under test

```
u = double("a user")
allow(User).to receive(:find) {u} # “User” is a class
expect (User.find(1)).to eq(u) # the class method was stubbed
```

- http://www.relishapp.com/rspec/rspec-mocks/v/3-2/docs/basics/partial-test-doubles

Multiple Return Values

- A stub might have to be invoked more than once
- Return values for each call (in the given order)

```
die = double("a rigged die")
allow(die).to receive(:roll).and_return(4,5,6) # a better version

puts die.roll # => 4
puts die.roll # => 5
puts die.roll # => 6
puts die.roll # => 6
# last value is returned for any subsequent invocations
```

- https://relishapp.com/rspec/rspec-mocks/v/3-2/docs/configuring-responses/returning-a-value

Method Stubs with Parameters

- Allow failure when calling stub with wrong parameters
- Respond differently based on passed parameters

- A mock / expectation will only be satisfied when called with matching arguments

```
calc = double("calculator")
allow(calc).to receive(:double).with(4).and_return(8)
expect(calc.double(4)).to eq(8) # this works
```

- Calling mock with wrong parameters fails:

```
dbl = double("spiderman")
# anything matches any argument
expect(dbl).to receive(:injury).with(1, anything, /bar/)
dbl.injure(1, 'lightly', 'car') # this fails, "car" does not match /bar/
```

Info:

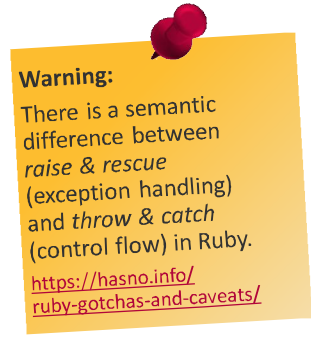
These are only a few of the matchers *RSpec-mocks* provides.

- https://relishapp.com/rspec/rspec-mocks/v/3-2/docs/setting-constraints/matching-arguments

Raising Errors

- A stub can raise an error when it receives a message
- Allows easier testing of exception handling

```
dbl = double()
allow(dbl).to receive(:foo).and_raise("boom")
dbl.foo # This will fail with:

#     Failure/Error: dbl.foo
#     RuntimeError:
#       boom
```



- https://relishapp.com/rspec/rspec-mocks/v/3-2/docs/configuring-responses/raising-an-error

Verifying Doubles

- Stricter alternative to normal doubles
- Check that methods being stubbed are actually present on the underlying object (if it is available)
- Verify that provided arguments are supported by actual method signature

```
class Post
  attr_accessor :title, :author, :body
end

post = instance_double("Post") # reference to the class Post
allow(post).to receive(:title)
allow(post).to receive(:message).with (‘a msg’) # this fails (not defined)
```

Tip:
`class_double()` & `object_double()` (create from existing “template” object) also exist.

- https://relishapp.com/rspec/rspec-mocks/v/3-2/docs/verifying-doubles

Why Use Mocks?

- Using mocks makes (some) tests more concise

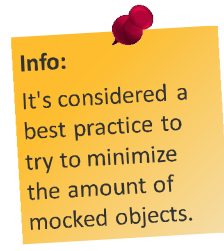
```
digger = Digger.new # a tracked vehicle
initial_left = digger.left_track.position
initial_right = digger.right_track.position
digger.turn_right # run method being tested

expect(digger.left_track.position - initial_left).to eq(+5)
expect(digger.right_track.position - initial_right).to eq(-5)
```

VS.

```
left_track = double('left_track')
right_track = double('right_track')
digger = Digger.new(left_track, right_track)
left_track.expects(:move).with(+5)
right_track.expects(:move).with(-5)

digger.turn_right # run method being tested
```

Test Doubles Pro and Contra

- Disadvantages
  - Mock objects have to accurately model the behaviour of the object they are mocking
  - Risk to test a value set by a test double (false positives)
  - Possibility to run out of sync with real implementation
    - à Brittle while refactoring

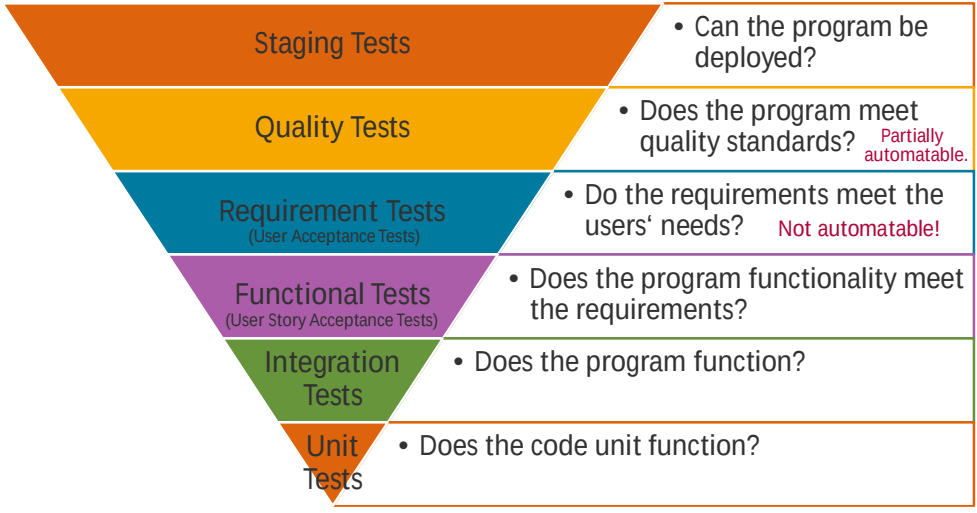


- Advantages
  - The test is focused on behavior
  - Speed (e.g. not having to use an expensive database query)
  - Isolation of tests (e.g. failure in model does not affect controller test)

Agenda

1. Why Behavior-driven Design (BDD)?
2. Building Blocks of Tests and BDD
   - Model Tests
   - View Tests
   - Controller Tests
   - Setup and Teardown
   - Test Data
   - Test Doubles
   - Integration & Acceptance Tests
   - Specialized Tests
3. Testing Tests & Hints for Successful Test Design
4. Outlook

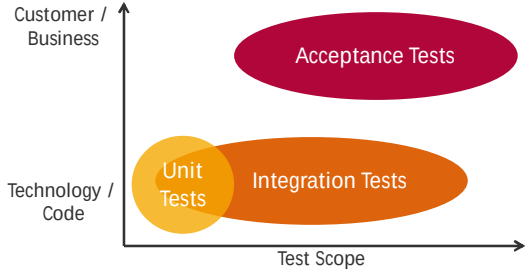
Levels of Testing



Integration & Acceptance Tests

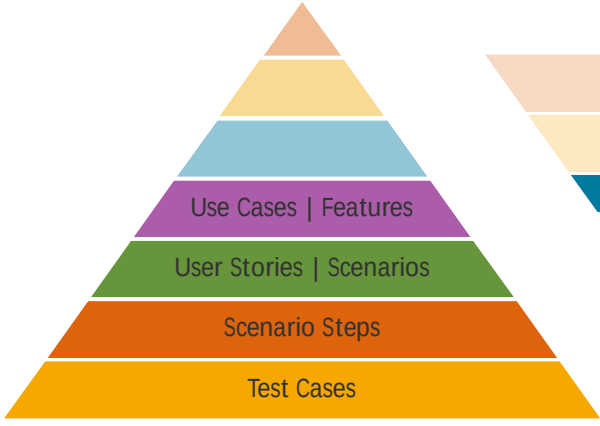
- Perform tests on the full system, across multiple components
- Test end-to-end functionality

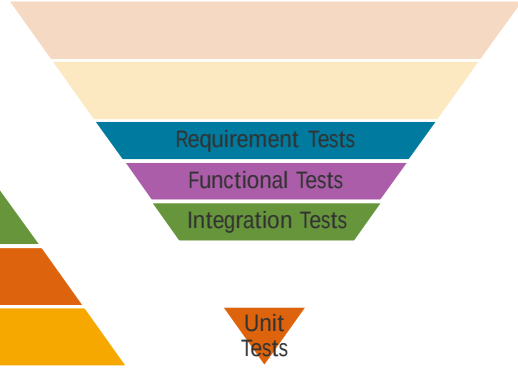
- Integration Tests
  - Build on unit tests, written for developers
  - Test component interactions
  - Consider environment changes (e.g. database instead of volatile memory)
- Acceptance Tests
  - Check if functionality satisfies the specification from a user perspective
  - Accessible for the stakeholders (e.g. using webpage via a browser)



- http://www.testfeed.co.uk/integration-vs-acceptance-tests/

BDD vs Test Levels





BDD Implementations

Behavior-driven development (BDD)

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

Implementations on different abstraction levels:

- Domain-specific languages (e.g. Cucumber)
  - Pro: Readable by non-technicians
  - Cons: Extra layer of abstraction, translation to Ruby
- Executable Code (e.g. using testing frameworks, RSpec, Mini::Test)
  - Pro: No translation overhead
  - Con: Barely readable by domain experts

Cucumber Test Framework

- Tool for running automated tests written in plain language
- Allows customers / non-technical stakeholders to read & write tests
- Provides runnable feature definitions
- Follows “Given-When-Then” pattern
- Features are located in `features/*.feature`
- Each line is a “step” that is implemented in Ruby
  - e.g. using RSpec and Capybara
  - Located in `features/step_definitions/*_steps.rb`
- Interpreted via regular expressions

- https://cukes.info/
- https://github.com/cucumber/cucumber/wiki

Cucumber Implementation Example

features/calculator_division.feature

```
# Cucumber feature
Feature: Division
  In order to allow users to
  calculate fractions, the calculator
  should perform correct divisions

Scenario: Floating point numbers
  Given the calculator is on
  When I press 3
  And I press /
  And I press 2
  And I press =
  Then I should see 1.5
```

features/step_definitions/division_steps.rb

```
# Steps for the Cucumber 'Division' feature
# implemented in Ruby
require 'calculator'

Before do
  @calc = Calculator.new
end
Given /the calulator is (on|off)/ do |state|
  @calc.power(state)
end
When /I press (.*)/ do |op|
  @calc.send(op)
end
Then /I should see (\d+)/ do |result|
  expect(@calc.result).to eq(result)
end
```

Cucumber vs. RSpec 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 DSL (no implementation)

```
describe "Author Management" do
  example "Add an author" do
    visit '/authors/'
    click_button 'Add author'
    fill_in 'Name', :with => 'Brecht'
    click_button 'Add'
    expect(page).to have_content 'Brecht'
  end
end
```

RSpec (with Capybara)

Discussion

- Which one is easier to understand ?
  - By programmers
  - By business stakeholders
- Which is easier to implement?
- 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

Capybara Test Framework

- Simulate how a real user would interact with a web application
- Well suited for writing acceptance & integration tests for web applications
- Provides DSL for “surfing the web”
  - e.g. `visit, fill_in, click_button`
- Integrates with RSpec
- Supports different “drivers”, some support Javascript evaluation
  - Webkit browser engine (used in Safari)
  - Selenium
    - Opens an actual browser window and performs actions within it

- https://github.com/jnicklas/capybara#using-capybara-with-rspec

Integration & Acceptance Tests (with Capybara)

```ruby
require 'capybara/rspec'

describe "the signin process", :type => :feature do
  before :each do
    User.make(:email => 'user@example.com', :password => 'password')
  end

  it "signs me in" do
    visit '/sessions/new'
    within("#session") do
      fill_in 'Email', :with => 'user@example.com'
      fill_in 'Password', :with => 'password'
    end
    click_button 'Sign in'
    expect(page).to have_css('div#success')
  end
end
```

Tip:

Capybara includes aliases for RSpec syntax:

`feature` instead of `describe …, :type => :feature,`

`scenario` instead of `it`,

`background` instead of `before`,

`given/given!` instead of `let/let!`

- https://github.com/jnicklas/capybara

Agenda

1. Why Behavior-driven Design (BDD)?
2. Building Blocks of Tests and BDD
   - Model Tests
   - View Tests
   - Controller Tests
   - Setup and Teardown
   - Test Data
   - Test Doubles
   - Integration & Acceptance Tests
   - Specialized Tests
3. Testing Tests & Hints for Successful Test Design
4. Outlook

Demo

https://github.com/hpi-swt2/Ruby-on-Rails-TDD-example

Route Tests

- route_to

```
require "rails_helper"

describe "routes for Widgets", :type => :routing do
  it "routes /widgets to the widgets controller" do
    expect(get("/widgets")).to route_to("widgets#index")
  end
end
```

- be_routable

```
require "rails_helper"

describe "routes for Widgets", :type => :routing do
  it "does not route to widgets/foo/bar" do
    expect(:get => "/widgets/foo/bar").not_to be_routable
  end
end
```

- http://www.relishapp.com/rspec/rspec-rails/v/3-2/docs/routing-specs/route-to-matcher
- http://www.relishapp.com/rspec/rspec-rails/v/3-2/docs/routing-specs/be-routable-matcher

Outgoing Mail Tests

- Test E-Mail generation (mock delivery)
  - Validate that application sends mail when expected
  - Validate that email content is what you expect
- 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).to receive(:deliver_signup).with("email@example.com", "Jim")
    post :signup, "Email" => "email@example.com", "Name" => "Jim"
  end
end
```

RSpec Testing Mail Content and Metadata


```ruby
describe "Signup Email" do, :type => :model do
  include EmailSpec::Helpers
  include EmailSpec::Matchers
  include Rails.application.routes.url_helpers

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

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

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

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

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

Testing Helper Modules

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

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

# Helper test
it "displays a complete user name" do
   @user = User.new(:first_name => "Garry", :last_name -> "Meyer")
   expect(display_name(@user)).to eq “Garry Meyer”
end
```

Optimizing the Testing Process

- Automate testing with Guard (https://github.com/guard/guard-rspec)
  - Automatically launch tests when files are modified
  - Run only the tests related to the change
- Parallelize tests with Spork (https://github.com/sporkrb/spork-rails)
  - Especially relevant with many time-consuming acceptance tests

Agenda

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

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

  - Absence of line coverage indicates a potential problem
  - Existence of line coverage means very little
  - In combination with good testing practices, coverage might say something about test suite reach
  - Circa 100% test coverage is a by-product of BDD

How to Measure Coverage?

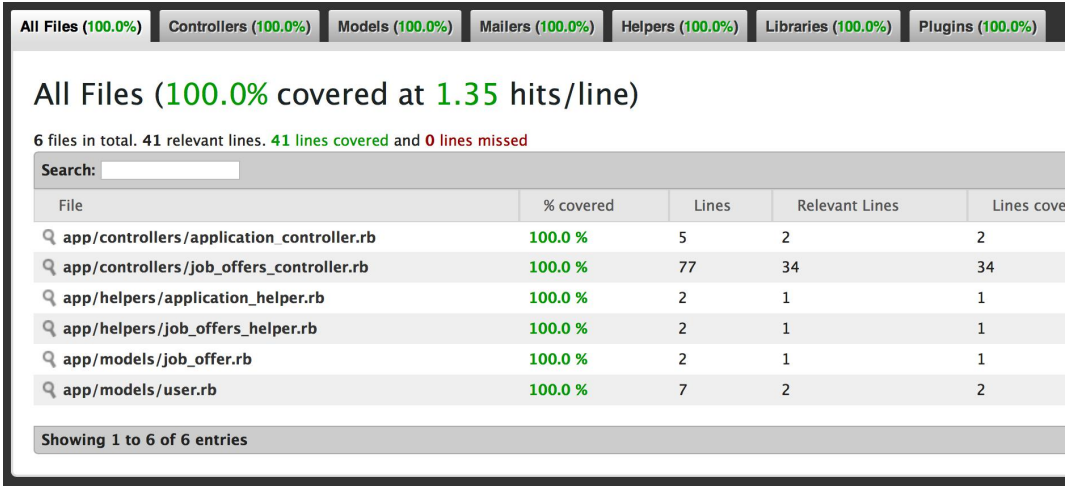
- Most useful approaches
  - Line coverage
  - Branch coverage

- Tool
  - SimpleCov (https://github.com/colszowka/simplecov)
  - Uses line coverage

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

  è 100% line coverage although 1 branch wasn't executed

SimpleCov


All Files (100.0%)
Controllers (100.0%)
Models (100.0%)
Mailers (100.0%)
Helpers (100.0%)
Libraries (100.0%)
Plugins (100.0%)
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

SimpleCov

```
16.  def new                                                    1
17.    @job_offer = JobOffer.new                                1
18.  end
19.
20.  # GET /job_offers/1/edit
21.  def edit                                                   1
22.  end
23.
24.  # POST /job_offers
25.  # POST /job_offers.json
26.  def create                                                 1
27.    @job_offer = JobOffer.new(job_offer_params)              5
28.
29.    respond_to do |format|                                   5
30.      if @job_offer.save                                     5
31.        format.html { redirect_to @job_offer, notice: 'Job offer was successfully created.' } 6
32.        format.json { render action: 'show', status: :created, location: @job_offer }       3
33.      else
34.        render_errors_and_redirect_to(@job_offer, 'new', format)                            2
35.      end
36.    end
37.  end
38.
39.  # PATCH/PUT /job_offers/1
40.  # PATCH/PUT /job_offers/1.json
41.  def update                                                 1
42.    respond_to do |format|                                   5
43.      if @job_offer.update(job_offer_params)                 5
44.        format.html { redirect_to @job_offer, notice: 'Job offer was successfully updated.' } 4
45.        format.json { head :no_content }                     2
```

- Standalone alternative to CodeClimate
- Methods related to failed tests are marked

```
39.  unless Devise.rack_session?                                1
40.    # We cannot use Rails Indifferent Hash because it messes up the flash object.
41.    class Devise::IndifferentHash < Hash
42.      alias_method :regular_writer, :[]= unless method_defined?(:regular_writer)
43.      alias_method :regular_update, :update unless method_defined?(:regular_update)
44.
45.      def [](key)
46.        super(convert_key(key))
47.      end
```

https://github.com/colszowka/simplecov

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, e.g. Timecop (https://github.com/travisjeffery/timecop)
    - Random numbers (try to avoid them or stub the generation)

5 Habits of Highly Successful Tests

- Clarity
  - Test purpose should be immediately understandable
  - Tests should be simple, readable
  - Make it clear how the test fits into the larger test suite
  - Worst case:

```
it "sums to 37" do
  expect(37).to eq(User.all_total_points)
end
```

  - Better:

```
it "rounds total points to nearest integer" do
  User.add_points(32.1)
  User.add_points(5.3)
  expect(37).to eq(User.all_total_points)
end
```

5 Habits of Highly Successful Tests

- Conciseness
  - Use the minimum amount of code and objects
  - Clear beats concise
  - Writing the minimum required amount of tests for a feature

  à Test suite will be faster

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

it 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
```

- Rails Test Prescriptions. Noel Rappin. 2010. p. 277. http://zepho.com/rails/books/rails-test-prescriptions.pdf

Conciseness: How many Assertions per Test?

- If a single call to a model results in many model changes:
  - High number of assertions à High clarity and cohesion
  - High number of assertions à Low test independence

  è Use context & describe and have 1 assertion per test

5 Habits of Highly Successful Tests

- Robustness
  - Underlying code is correct à test passes
  - Underlying code is wrong à test fails
  - *Example:* view testing

```
describe "the signin process", :type => :feature do
  it "signs me in (text version)" do
    visit '/dashboard'
    expect(page).to have_content “My Projects”
  end
  # version below is more robust against text changes
  it "signs me in (css selector version)" do
    visit '/dashboard'
    expect(page).to have_css "h2#projects"
  end
end
```

5 Habits of Highly Successful Tests

- Robustness
  - Reusable constants instead of magic numbers

```ruby
def assert_user_level(points, level)
  user = User.make(:points => points)
  expect(level).to eq(user.level)
end

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

  - But be aware of tests that always pass regardless of underlying logic

- Rails Test Prescriptions. Noel Rappin. 2010. p. 278. http://zepho.com/rails/books/rails-test-prescriptions.pdf

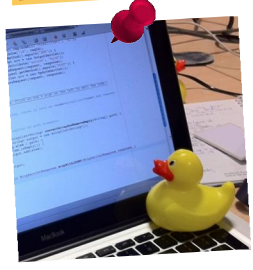
Troubleshooting

- Reproduce the error
  - Write a test
- What has changed?
  - Isolate commit/change that causes failure
- Isolate the failure
  - `thing.inspect`
  - Add assertions/prints to your test
  - `Rails.logger.error`
  - `save_and_open_page`
    (Capybara method to take a snapshot of a page)
- Explain to someone else
  - Rubber duck debugging

Tip:

`git-bisect` is a powerful git tool that can help isolate the change that caused a bug by binary search through the commit history.

http://git-scm.com/docs/git-bisect



http://commons.wikimedia.org/wiki/File:Rubber_duck_assisting_with_debugging.jpg

Manual Fault Seeding

- Conscious introduction of faults into the program
- Run tests
- Minimum 1 test should fail

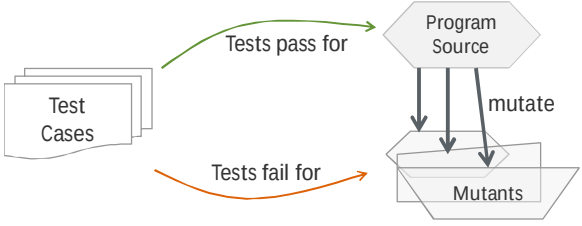
If no test fails, then a test is missing

- Possible even with 100% line coverage
- Asserts functionality coverage

Mutation Testing

Mutant: Modified version of the program with small change

- Tests correctly cover code à Test should notice change and fail




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

↓

```
if not month > 13 then
  year -= month / 12
  month = month % 12
end
```

- Mutation Coverage: How many mutants did not cause a test to fail? Asserts functionality & behavior coverage
  - For Ruby: *Mutant* (https://github.com/mbj/mutant)

Summary

BDD
- Motivation
- BDD Cycle

TDD
- Pros & Cons

Automated Testing
- Model/View/Controller
- Test Data
- Test Doubles

Testing Hierarchy
- Integration Tests
- Acceptance Tests

Test Quality
- Coverage
- Mutation Tests

Further Reading

http://betterspecs.org – Collaborative RSpec best practices documentation effort

Everyday Rails Testing with RSpec by Aaron Sumner, leanpub

The RSpec Book: Behaviour-Driven Development with RSpec, Cucumber, and Friends by David Chelimsky et al.

Rails 4 Test Prescriptions: Build a Healthy Codebase by Noel Rappin, Pragmatic Programmers 2014

Quizzes

http://www.codequizzes.com/rails-test-driven-development/controller-specs

http://www.codequizzes.com/rails-test-driven-development/model-specs

Outlook (Dec 4, 1st slot)

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