Behavior-Driven (Software) Development

Scalable Software Engineering
WS 2021/22

Enterprise Platform and Integration Concepts

Image by Jay on flickr: https://www.flickr.com/photos/activars/6590201161/ (CC BY-NC 2.0)
Agenda

1. Behavior-driven Development (BDD)
   - Automated Testing
   - Writing Software that Matters
2. Basics of Tests and BDD
Automated Testing

Goals of automated testing

- Find errors faster
- Better code in the long run (correct, robust, maintainable)
- Less overhead when testing → tests run more frequently
- Easier to modify existing features, help with refactoring

But

- Tests might have bugs
- Test environment != production environment (what could help here?)
- Tests must be maintained (and refactored)
1. Behavior-driven Development (BDD)
   - Automated Testing
   - Writing Software that Matters
2. Basics of Tests and BDD
“BDD is about implementing an application by describing its behavior from the perspective of its stakeholders”

– Dan North (originator of BDD)

**Principles**

- **Unified language** between business and technology
- Systems should have identified, verifiable stakeholder value
- Up-front analysis, design and planning have diminishing returns

Related: YAGNI
Specify behavior in unified way, e.g.

- As a User Story (semi-formal language)
  - Narrative: As a <role>, I want <feature>, so that <value>
  - Acceptance criteria: Given <context>, When <event>, Then <outcome>
  - Can be mapped to code

- As an (automated) end-to-end test
  - Formalized language (code)
    - Go to URL, select ..., type ..., press 'OK'
    - RSpec & Capybara testing frameworks follow this approach

We use 'acceptance test' and 'end-to-end' test as synonyms here
BDD Cycle

1. Adapted from [Chelimsky et al.: The Rspec Book, 2010]
Definition of Done

How do I know when to stop? E.g.
- Acceptance criteria fulfilled
- All tests are green

Additional possibilities
- Objective quality standards are met
- Second opinions / reviews
- Secure
- Documented

Definition of Done:
A team’s consensus of what it takes to complete a feature.
BDD Cycle

Hierarchy of tests

- Tests should be specified in terms of desired behavior
- **Focus on Big Picture** (vs. implementation detail)
  - Big picture: end-to-end test, *e.g.* visit website, modify data
  - Technical implementation: unit test, *e.g.* this method returns only ints
- BDD can be considered "outside-in"
  - From coarse to fine
  - Acceptance/end-to-end tests in outer ring
  - Unit/other tests inside

Of course, technical implementation is still required
## Summary

### Behavior-driven Development
- Motivation
- Underlying ideas
- Automated testing vs manual

### BDD Cycle
- Inner ring & outer ring
- Definition of Done
- Pros & Cons
Introduction to Testing in Ruby on Rails

Scalable Software Engineering
WS 2021/22
Agenda

1. Behavior-driven Design (BDD)
2. Basics of Tests and BDD
   - Model Tests
   - View Tests
   - Integration & Acceptance Tests

In the intro exercise we hope you experience BDD
Test::Unit vs. RSpec

- Test::Unit comes with Ruby

```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
```
RSpec: Rails Testing Framework

- **RSpec offers syntactical sugar** over Rails default (Test::Unit)
- Many built-in modules
- `rspec` command with tools to constrain what examples are run

```ruby
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
```

[All following code refers to RSpec 3.2](http://blog.thefirehoseproject.com/posts/test-driven-development-rspec-vs-test-unit/)
RSpec Structure

Using `describe` and `it` like in a conversation

- "Describe an order!" "It sums prices of items."
- `describe` creates a test group
- `it` declares tests within group
- `context` for nested groups / structuring

```ruby
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
```


describe and context; it and example are aliases (do the same thing)
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)

[Specialized matchers, e.g.:]
expect(actual).to respond_to(expected)

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

1. Behavior-driven Design (BDD)
2. Basics of Tests and BDD
   - Model Tests
   - View Tests
   - Integration & Acceptance Tests
Model Tests

A Rails model
- Accesses data through an **Object-relational mapping** (ORM) tool
  - Object-oriented programming languages deal with "objects"
  - Relational databases deal with scalar values (*int, string*) in tables
  - ORM translates between these worlds
- Implements **business logic**
- Is "weighty", i.e. contains most code and application logic

**Model tests in Rails**
- Easiest tests to write
- Test most of application logic
Model Tests

Model test hints
■ Should cover almost all 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 (test-first approach)

Minimal model test set
■ One test for the “happy-path case” (the usual, normal way)
■ One test for each code branch
■ Corner cases (nil, wrong values, …), if appropriate

■ Keep each test small! (why?)
Model Tests: Example

app/models/contact.rb

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

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

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. Behavior-driven Design (BDD)
2. Basics of Tests and BDD
   - Model Tests
   - View Tests
   - Integration & Acceptance Tests
View Tests

A Ruby on Rails view
- Has only minimal logic
- Should not call the database (*why?*)
- Renders data passed by the controller as HTML

Challenges of view tests (*ideas?*)
- Time-intensive to write, HTML structure complex
- How to test look & feel?
- Brittle regarding interface redesigns

Different terminologies: view (RoR) ~ template (Django) controller (RoR) ~ view (Django)
View Tests

Verify the **logical and semantic structure** of rendered content

**Goals**
- Validate that view layer runs without error
- Render view templates in isolation
- Check that passed data is presented as expected
- Validate conditional display of information, e.g. based on user's role

**Possible anti-patterns**
- Validating HTML markup
- Checking the "design"
- Testing text that's likely to change instead of core structure elements
describe "users/index", type: :view 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

user.save! (with !) raises an ActiveRecord::RecordInvalid error, when user.save returns false
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

- [https://github.com/jnicklas/capybara](https://github.com/jnicklas/capybara)
- [rubydoc.info/github/jnicklas/capybara/master/Capybara/Node/Matchers](https://rubydoc.info/github/jnicklas/capybara/master/Capybara/Node/Matchers)

Capybara offers many helpful "matchers", including has_button, has_table & has_unchecked_field
Agenda

1. Behavior-driven Design (BDD)
2. Basics of Tests and BDD
   - Model Tests
   - View Tests
   - Integration & Acceptance Tests
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

**Acceptance Tests**
- Check if functionality satisfies the specification from a user perspective
- Possibly accessible for stakeholders

http://www.testfeed.co.uk/integration-vs-acceptance-tests/
Behavior-driven development (BDD)

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

Implementations on different abstraction levels:

- **Domain-specific languages** (e.g. Cucumber)
  - Pro: Readable by non-technicians, reads like formal English
  - Cons: Extra layer of abstraction, translation to executable Ruby code

- **Executable Code** (e.g. using testing frameworks, RSpec & Capybara)
  - Pro: No translation overhead
  - Con: Harder to read for non-developers
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 web pages”
  - e.g. `visit`, `fill_in`, `click_button`
- Integrates with RSpec
- Supports different “drivers”, some support JavaScript evaluation
  - Webkit browser engine
  - Selenium
    - Opens an actual browser window and performs actions within it

Acceptance Tests (Capybara)

```ruby
require 'capybara/rspec'

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

  it "signs me in" do
    visit new_session_path
    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
```

- What are some issues with this test?
- What is good?

Capybara includes aliases for RSpec syntax:
- `feature` is `describe ..., :type => :feature`
- `scenario` is `it`, `background` is `before`

[https://github.com/jnicklas/capybara](https://github.com/jnicklas/capybara)
Summary

Rails Testing Frameworks
- Rspec Structure
- Capybara Acceptance Tests

Testing
- Model Tests
- View Tests
- Integration & Acceptance Tests