Lecture 1

Test Driven Development in Ruby

Aleksander Smywiński-Pohl

Elektroniczne Przetwarzanie Informacji

21 marca 2013
Introduction to Testing

RSpec – unit tests

Test doubles
Why?

- defensive programming (safety net)
- test driven development (TDD)
- behavior driven development (BDD)
- constant refactoring (red-green-refactor cycle)
- instant deployment (deploy several times a day)
Types of tests

- unit tests
- integration tests
- acceptance tests
- performance tests
- regression tests
Unit tests

- one class (SUT – system under testing)
- isolation
- specification of behavior
- three types of conditions
  - typical: EPI student joins the course
  - unusual: Erasmus student joins the course
  - invalid: non-existent student joins the course
- different contexts
Agenda

Introduction to Testing

RSpec – unit tests

Test doubles
describe TodoList do
  context "without tasks" do
    subject { TodoList.new(tasks) }
    let(:tasks) { [] }

    it "is empty" do
      subject.should be_empty
    end
  end
end
RSpec keywords

- **describe** – SUT
- **context** – specific context of testing
- **subject** – SUT instance
- **let** – context-dependant variables
- **it**, **specify**, **example** – test definition
- **before**, **after**, **around** – helper code, that is run before, after and before and after (around) each test
RSpec – concise syntax

describe TodoList do
  context "without tasks" do
    subject { TodoList.new(tasks) }
    let(:tasks) { [] }

    it { should be_empty }
  end
end
RSpec – before, after

describe TodoList do
  context "with DB connection" do
    before do
      @db = Database.new(port: 7777, host: "db.example.com")
      @db.connect
    end

    after do
      @db.close
    end

    # test definitions...
  end
end
shared_examples "collections" do |collection_class|
  it "is empty when first created" do
    expect(collection_class.new).to be_empty
  end
end

describe Array do
  include_examples "collections", Array
end

describe Hash do
  include_examples "collections", Hash
end
RSpec expectations

# equivalence
expect(actual).to eq(expected)
actual.should == expected

# identity
expect(actual).to be(expected)
actual.should be(expected)

# comparison
expect(actual).to > expected
expect(actual).to >= expected
expect(actual).to <= expected
expect(actual).to < expected
expect(actual).to be_within(delta).of(expected)
actual.should > expected
RSpec expectations

# regular expressions
expect(actual).to match(/expression/)  

actual.should =~ /expression/

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

# yielding
expect { |b| 5.tap(&b) }.to yield_control  
expect { |b| 5.tap(&b) }.to yield_with_args(5)  
expect { |b| "a string".tap(&b) }.to yield_with_args(/str/)
RSpec expectations

# predicates
expect(actual).to be_xxx  # passes if actual.xxx?
expect(actual).to have_xxx(:arg)  # passes if actual.has_xxx?(:arg)

actual.should be_xxx

# collection membership
expect(actual).to include(expected)
expect(actual).to start_with(expected)
expect(actual).to end_with(expected)

actual.should include(expected)
RSpec custom matchers

RSpec::Matchers.define :have_many_tasks do
  match do |subject|
    subject.tasks.size.should > 10
  end
end

describe TodoList do
  subject { TodoList.new(tasks) }
  context "with many tasks" do
    let(:tasks) { 50.times.map{|i| "Task nr #{i}" } }

    it { should have_many_tasks }
  end
end
RSpec alternatives

- test/unit – built-in unit testing framework
- Shoulda – similar to RSpec, more Rails-specific matchers
- minitest – (presently) built-in unit testing/spec framework, fast and minimalistic
- Cucumber – (not for unit-testing) BDD framework designed to write human readable and machine runnable specification
Cucumber (acceptance tests)

**Feature:** advanced webapps course enrollment
The course should allow students to enroll in February

**Background:**
- **Given** the current month is February

**Scenario:** simple enrollment
- **Given** the student studies EPI
- **And** the student logs into the system
- **When** the student selects the course
- **And** the student clicks the 'enroll' button
- **Then** the student should be enrolled
Agenda

Introduction to Testing

RSpec – unit tests

Test doubles
Test doubles

- stub – test indirect inputs of the method
- spy – between stub and mock
- mock – test indirect outputs of the method
- fake object – replacement for real objects, e.g. call parameters

**Beware!** – RR allows for mixing and matching stubbed and mocked methods
### Stubs/stubbed methods

#### Indirect inputs for the test

```ruby
class Game
  attr_reader :mosters

  def initialize(configuration)
    @configuration = configuration
    @mosters = []
  end

  def generate_mosters(monster_factory)
    if @configuration[:mode] == :practice
      @mosters += 3.times.map{ monster_factory.new }
    else
      @mosters += 30.times.map{ monster_factory.new }
    end
  end
end
```
Stubs/stubbed methods

describe Game do
  subject(:game) { Game.new(configuration) }
  let(:configuration) { conf = stub![:mode] { mode }.subject
    stub(conf)[:speed] { 10 }
    conf
  }
  let(:monster_factory) { ... }

  context "in practice mode" do
    let(:mode) { :practice }
    it "should generate few monsters" do
      game.generate_mosters(monster_factory)
      game.monsters.size.should < 10
    end
  end

  context "in tournament mode" do
    let(:mode) { :tournament }
    it "should generate many monsters" do
      game.generate_mosters(monster_factory)
      game.monsters.size.should > 10
    end
  end
end
Stubs – RR syntax

- **stub!** – create a new stub
- **stub!....subject** – pull out the created object
- **stub(object).method** – add a stubbed method to the object
- **stub(object).method { value }** – define the value in a block
- **stub(object).method(params)** – define a specific parameter for the stubbed method
- **stub(object).method1.stub!.method2** – chain method calls
Mocks/mocked methods

Indirect outputs of the test

```ruby
class TodoList
  def initialize(social_network)
    @list = []
    @social_network = social_network
  end

  def <<(task)
    @list << task
    @social_network.spam(task + " added")
    self
  end
end
```
describe TodoList do
  subject(:list) { TodoList.new(network) }
  let(:network) { stub }

  it "spams the social network" do
    mock(network).spam("Buy toilet paper added")
    mock(network).spam("Clean the toilet added")
    mock(network).spam("Write RoR project added")

    list << "Buy toilet paper" << "Clean the toilet" << "Write RoR project"
  end
end
Mocks – calls number

- `mock(object).foo { 'bar' }` – expect one call to `foo`
- `mock(object).foo.times(2) { 'bar' }` – expect two calls to `foo`
- `mock(object).foo.at_least(2) { 'bar' }` – expect at least two calls to `foo`
- `mock(object).foo.never` – forbid calls to `foo`
- `dont_allow(object).foo` – same as above
Mocks – call parameters

- `mock(object).foo('param')` – expect call with 'param'
- `mock(object).foo.with_any_args` – whatever parameters
- `mock(object).foo.with_no_args` – without parameters
- `mock(object).foo(anything)` – with one parameter of any kind
- `mock(object).foo(is_a(Time))` – with param. that is an instance of a specific class
- `mock(object).foo(numeric)` – a numeric parameter
RR alternatives

- rspec/expectations
- Mocka
- Flexmock