Testing como apoyo al desarrollo

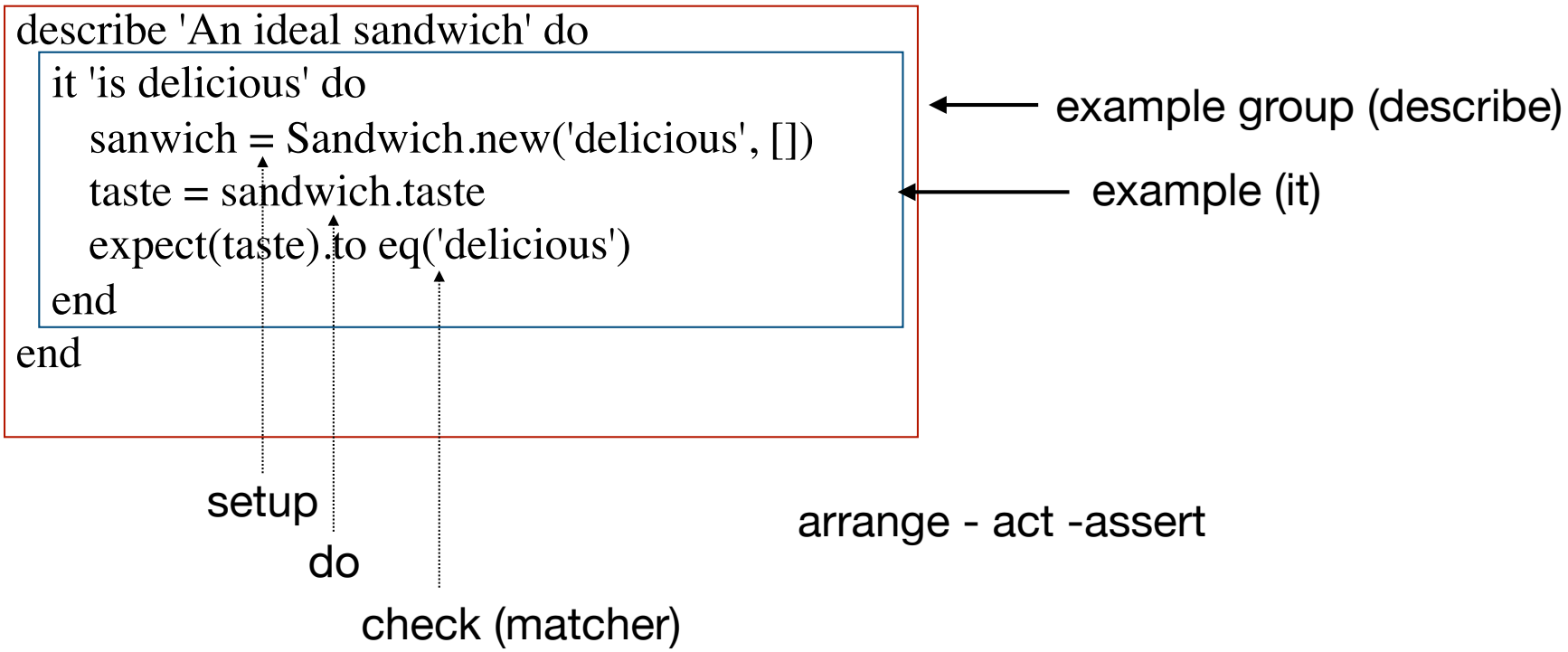
- Facilita la comprensión a findo de la funcionalidad deseada
- Facilita el construir rápidamente una solución e ir mejorándola en forma posterior (refactoring)
- Ciclo red/green/refactor
- La idea llevada al extremo: TDD (test driven development)
  - escribir los tests asociados a in trozo de código ANTES de escribir el código
  - probar el código (que no existe) - fails
  - escribir código para que pueda superar el test
  - refactor
  - escribir el siguiente test

Tests Automatizados

- dependiendo de la plataforma hay herramientas que facilitan el desarrollo de los tests
- muchas reproducen el modelo de JUnit (Java)
- La plataforma Ruby desde sus inicios ha incluido herramientas para hacer test unitarios
  - test::unit
  - minitest (reemplaza al anterior a partir de 1.9.3, default)
  - gema rspec (recomendable)

Introducción a RSpec

- Creada por Steven Baker en 2005
- El programador debe escribir "specs" que describen el comportamiento esperado
- Ejemplo (del texto Effective Testing with RSpec 3):



Terminología

- Un test valida que un trozo de código esté funcionando correctamente
- Un spec describe el comportamiento deseado de un trozo de código
- Un ejemplo muestra como se espera que una funcionalidad sea usada

Corriendo el test ...

```
Tatooine:02 jnavon$ rspec
F

Failures:

  1) An ideal sandwich is delicious
     Failure/Error: sandwich = Sandwich.new('delicious', [])

     NameError:
       uninitialized constant Sandwich
     # ./spec/sandwich_spec.rb:12:in `block (2 levels) in <top (required)>'

Finished in 0.00239 seconds (files took 0.10024 seconds to load)
1 example, 1 failure

Failed examples:

rspec ./spec/sandwich_spec.rb:11 # An ideal sandwich is delicious

Tatooine:02 jnavon$
```

passing specs - green
failing specs - red
example descriptions and structural text - black
extra details - blue
pending specs - yellow

Haciendo que el test pase ...

```
Sandwich = Struct.new(:taste, :toppings)

RSpec.describe 'An ideal sandwich' do
  it 'is delicious' do
    sandwich = Sandwich.new('delicious', [])

    taste = sandwich.taste

    expect(taste).to eq('delicious')
  end
end
```

y ahora ...

```
Tatooine:03 jnavon$ rspec
.

Finished in 0.00498 seconds (files took 0.15996 seconds to load)
1 example, 0 failures

Tatooine:03 jnavon$
```

- normalmente el código que no es del test va en otro directorio (aquí se hizo por simplicidad así)
- se utilizó la clase predefinida Struct de Ruby
  - es una manera conveniente de empaquetar un set de atributos sin tener que generar una clase
  - no es necesario definir attr_accesors

```
Customer = Struct.new(:name, :address) do
  def greeting
    "Hello #{name}!"
  end
end

dave = Customer.new("Dave", "123 Main")
dave.name     #=> "Dave"
dave.greeting #=> "Hello Dave!"
```

Un archivo spec mas real

```
require 'rspec'
  require_relative 'address_validator'
  describe 'AddressValidator' do                                    # example  group
    it 'returns false for incomplete address' do                    # example
      address = { street: "123 Any Street", city: "Anytown" }
      expect(AddressValidator.valid?(address)).to eq(false)         # matcher
    end
    it 'missing_parts returns an array of missing required parts' do  # example
      address = { street: "123 Any Street", city: "Anytown" }
      expect(AddressValidator.missing_parts(address)).to eq([:region, :postal_code, :country])  # matcher
    end
end
```

Algunos elementos adicionales

- Setup compartido entre los ejemplos
- Es muy común que los ejemplos de un grupo compartan el setup
- Puede simplificarse la spec significativamente

Vuelta al sandwich

```
Sandwich = Struct.new(:taste, :toppings)

RSpec.describe 'An ideal sandwich' do
  it 'is delicious' do
    sandwich = Sandwich.new('delicious', [])

    taste = sandwich.taste

    expect(taste).to eq('delicious')
  end

  it 'lets me add toppings' do
    sandwich = Sandwich.new('delicious', [])

    sandwich.toppings << 'cheese'
    toppings = sandwich.toppings

    expect(toppings).not_to be_empty
  end
end
```

```
Sandwich = Struct.new(:taste, :toppings)

RSpec.describe 'An ideal sandwich' do
  before { @sandwich = Sandwich.new('delicious', []) }

  it 'is delicious' do
    taste = @sandwich.taste

    expect(taste).to eq('delicious')
  end

  it 'lets me add toppings' do
    @sandwich.toppings << 'cheese'
    toppings = @sandwich.toppings

    expect(toppings).not_to be_empty
  end
end
```

- código asociado a before corre antes de cada ejemplo
- la instancia @sanwich no es compartida (cada ejemplo obtiene uno limpio)

Uso de let y let!

- bloque before corre siempre antes de cada ejemplo
- bloque let no es evaluado hasta que es necesario
- al usar let! se fuerza la invocación en cada ejemplo

```
$count = 0
describe "let" do
  let(:count) { $count += 1 }

  it "stores the value" do
    expect(count).to eq(1)
    expect(count).to eq(1)
  end

  it "is not cached across examples" do
    expect(count).to eq(2)
  end
end
```

```
$count = 0
describe "let!" do
  invocation_order = []

  let!(:count) do
    invocation_order << :let!
    $count += 1
  end

  it "calls the helper method in a before hook" do
    invocation_order << :example
    expect(invocation_order).to eq([:let!, :example])
    expect(count).to eq(1)
  end
end
```

Context

- A veces es conveniente separar grupos de ejemplos al interior del describe inicial
- En realidad context es un alias de describe
- Describe agrupa ejemplos asociados a una funcionalidad
- Context agrupa los subconjuntos de ejemplos que parten del mismo estado

Ejemplo

```
describe "launch the rocket" do
  before(:each) do
    #prepare a rocket for all of the tests
    @rocket = Rocket.new
  end

  context "all ready" do
    before(:each) do
      #under the state of ready
      @rocket.ready = true
    end

    it "launch the rocket" do
      @rocket.launch().should be_true
    end
  end

  context "not ready" do
    before(:each) do
      #under the state of NOT ready
      @rocket.ready = false
    end

    it "does not launch the rocket" do
      @rocket.launch().should be_false
    end
  end
end
```

Más sobre Matchers

- Rspec incluye una gran cantidad
- Es posible crear matchers propios (custom)
- Dada una expresión Ruby a, un matcher tiene la forma
  - expect(a).to matcher
  - expect(a).not_to matcher or expext(a).to_not matcher

Matcher	Passes if…	Available aliases
eq(x)	a == x	an_object_eq_to(x)
eql(x)	a.eql?(x)	an_object_eql_to(x)
equal(x)	a.equal?(x)	be(x) an_object_equal_to(x)

Matcher	Passes if…	Available aliases
be_truthy	a != nil && a != false	a_truthy_value
be true	a == true	
be_falsey	a == nil || a == false	be_falsy a_falsey_value a_falsy_value
be false	a == false	
be_nil	a.nil?	a_nil_value

Matcher	Passes if…	Available aliases
be_an_instance_of(klass)	a.class == klass	be_instance_of(klass) an_instance_of(klass)
be_a_kind_of(klass)	a.is_a?(klass)	be_a(klass) be_kind_of(klass) a_kind_of(klass)

Matcher	Passes if…	Available aliases
be == x	a == x	a_value == x
be < x	a < x	a_value < x
be > x	a > x	a_value > x
be <= x	a <= x	a_value <= x
be >= x	a >= x	a_value >= x
be =~ x	a =~ x	a_value =~ x
be === x	a === x	a_value === x

Delta/Range Comparisons

Matcher	Passes if…	Available aliases
be_between(1, 10).inclusive	a >= 1 && a <= 10	be_between(1, 10) a_value_between(1, 10).inclusive a_value_between(1, 10)
be_between(1, 10).exclusive	a > 1 && a < 10	a_value_between(1, 10).exclusive
be_within(0.1).of(x)	(a - x).abs <= 0.1	a_value_within(0.1).of(x)
be_within(5).percent_of(x)	(a - x).abs <= (0.05 * x)	a_value_within(5).percent_of(x)
cover(x, y)	a.cover?(x) && a.cover?(y)	a_range_covering(x, y)

Strings and Collections

Matcher	Passes if…	Available aliases
contain_exactly(2, 1, 3)	a.sort == [2, 1, 3].sort	match_array([2, 1, 3]) a_collection_containing_exactly(2, 1, 3)
start_with(x, y)	a[0] == x && a[1] == y	a_collection_starting_with(x, y) a_string_starting_with(x, y)
end_with(x, y)	a[-1] == x && a[-2] == y	a_collection_starting_with(x, y) a_string_starting_with(x, y)
include(x, y)	(a.include?(x) && a.include?(y)) || (a.key?(x) && a.key?(y))	a_collection_including(x, y) a_string_including(x, y) a_hash_including(x, y)

Duck Typing and Attributes

Matcher	Passes if…	Available aliases
have_attributes(w: x, y: z)	a.w == x && a.y == z	an_object_having_attributes(w: x, y: z)
respond_to(:x, :y)	a.respond_to?(:x) && a.respond_to?(:y)	an_object_responding_to(:x, :y)
respond_to(:x) .with(2).arguments	a.respond_to?(:x) && a.method(:x).arity == 2	an_object_responding_to(:x) .with(2).arguments

Dynamic Predicates

Matcher	Passes if…	Available aliases
be_xyz	a.xyz? || a.xyzs?	be_a_xyz be_an_xyz
be_foo(x, y, &b)	a.foo(x, y, &b)? || a.foos(x, y, &b)?	be_a_foo(x, y, &b) be_an_foo(x, y, &b)
have_xyz	a.has_xyz?	
have_foo(x, y, &b)	a.has_foo(x, y, &b)?	

Additional Matchers

Matcher	Passes if…	Available aliases
exist	a.exist? || a.exists?	an_object_existing
exist(x, y)	a.exist(x, y)? || a.exists(x, y)?	an_object_existing(x, y)
satisfy { |x| ... }	Provided block returns true	an_object_satisfying { |x| ... }
satisfy("criteria") { |x| ... }	Provided block returns true	an_object_satisfying("...") { |x| ... }

Ejemplo TDD : Gatherer

- app destinada a llevar un control de las tareas de un proyecto
- para cada tarea, status, desarrolladores asignados, etc
- interesa llevar registros de tiempos para completar las tareas

Primer test

- un proyecto sin tareas pendientes está terminado
- estado inicial al crear proyecto sin tareas debería ser "done"
- el modelo del project va en app/models/project.rb
- el spec (example) va en spec/models/project_spec.rb

spec/models/project_spec.rb

```
require 'rails_helper'

RSpec.describe Project do
  it 'considers a project with no tasks to be done' do
    project = Project.new
    expect(project.done?).to be_truthy
  end
end
```

Corriendo el test

```
$ rspec
gatherer/spec/models/project_spec.rb:3:in `<top (required)>':
    uninitialized constant Project
```

El test falla porque aún no hemos definido Project
La corrección mas simple sería agregar archivo app/models/project.rb con

```
class Project
end
```

Corriendo nuevamente el test

```
$ rspec
F
Failures:
  1) Project considers a project with no tasks to be done
     Failure/Error: expect(project.done?).to be_truthy
     NoMethodError:
       undefined method `done?' for #<Project:0x00000107ce67d0>
     # ./spec/models/project_spec.rb:6:in `block (2 levels) in <top (required)>'
Finished in 0.00104 seconds (files took 1.29 seconds to load)
1 example, 1 failure
Failed examples:
rspec ./spec/models/project_spec.rb:4 #
    Project considers a project with no tasks to be done
```

Refactor

```
class Project
  def done?
    true
  end
end
```

```
$ rspec
.
Finished in 0.00105 seconds (files took 1.2 seconds to load)
1 example, 0 failures
```

Agregamos un Segundo Ejemplo a project_spec.rb

```
require 'rails_helper'

RSpec.describe Project do
  it 'considers a project with no tasks to be done' do
    project = Project.new
    expect(project.done?).to be_truthy
  end

  it 'knows that a project with an incomplete task is not done' do
    project = Project.new
    task = Task.new
    project.tasks << task
    expect(project.done?).to be_falsy
  end
end
```

Debemos Modificar el modelo ...

```
class Task

end
```

← app/models/task.rb

```
class Project
  attr_accessor :tasks
  def initialize
    @tasks = []
  end
  def done?
    tasks.empty?
  end
end
```

← app/models/project

Mejoramos Spec con lets

```
require 'rails_helper'
RSpec.describe Project do
  context 'empty' do
    let(:project) { Project.new }
    let(:task) { Task.new }

    it "considers a project with no test to be done" do
      expect(project).to be_done
    end

    it 'knows that a project with an incomplete test is not done' do
      project.tasks << task
      expect(project).not_to be_done
    end
  end
end
```

Falta diferenciar entre tareas completas e incompletas

Introducimos un test en spec/models/task_spec-rb

```
require 'rails_helper'
RSpec.describe Task do
  it 'can distinguish a completed task' do
    task = Task.new
    expect(task).not_to be_complete
    task.mark_completed
    expect(task).to be_complete
  end
end
```

y el modelo de tarea ...

```
class Task
  def initialize
    @completed = false
  end
  def mark_completed
    @completed = true
  end
  def complete?
    @completed
  end
end
```

Proyecto completado cuando las tareas lo están ...

Agregamos un nuevo test a spec de project

```
it 'marks a project done if its tasks are done' do
  project.tasks << task
  task.mark_completed
  expect(project).to be_done
end
```

y en el modelo modificamos método done?

```
def done?
  tasks.reject(&:complete?).empty?
end
```

Para Profundizar

Marston M. & Dees, I "Effective Testing with RSpec 3", The Pragmatic Programmers 2017

Sumner A. "Everyday Rails Testing with RSpec", Leanpub.com 2018