Test: vocabulaire

Objectif d’un test

vérifier le bon fonctionnement d’un programme par l’écriture d’exemples de fonctionnement

assertion/expectation

confronter une valeur calculée par le code à une valeur attendue

Test

programme exécutable, regroupant des expectations. Le résultat sépare les expectations vérifiées de celles non validées.

Rspec

gem install rspec

DSL pour écrire des exemples exécutables du comportement attendu d'un code.

describe/context : groupe d'exemples

it/specify : un exemple de comportement du code, avec une expectation pour spécifier le comportement attendu.

```
<pre 'code'>
```

```ruby
# something_spec.rb
describe "something" do # bloc describe : un groupe d'exemples
  it "does a thing" do # bloc it : un exemple
  end

  it "does this also" do
  end
end
```

```
rspec something_spec.rb --format documentation

something
  does a thing
  does this also

Finished in 0.47873 seconds
2 examples, 0 failures
```

Écriture d’un exemple

Expectations

```
object.should matcher(expected)
object.should_not matcher(expected)
```

Matchers

`be, be_true, be_false, be_nil, <, >, <=, >=`

- égalité

  be(val), eq(val), eql(val), == val

  Person.name.should == “Bob”

- regexp

  match /regexp/

Pour aller plus loin, voir la documentation

Exemples

<pre 'code'>

```ruby
describe "+" do
  it "should return 4 with 2 and 2" do
    (2+2).should == 4
  end
end

describe String do
  it 'should interpolate #{}' do
    val = 4
    s = "val: #{val}"
    s.should == "val: 4"
  end
end
```

- limitez dans la mesure du possible le nombre de matchers par exemple : 1 est idéal. Quand les spécifications changent, les tests seront plus faciles à modifier.
- forme d'un exemple : `it "should have a behaviour" do ...code... end`
- code d'un exemple :
  1. fixer le point de départ : définir des variables avec un contenu conu
  2. obtenir un résultat en appelant le code mettant en jeu le comportement testé
  3. comparer avec un matcher le résultat souhaité avec le résultat obtenu

Exemples

En reprenant les exemples données dans la présentation ruby, et en les exprimant sous forme de test avec rspec :

```
<pre 'code'>
```

```ruby
# fichier string_spec.rb
describe "String#split" do
  it "transforms a string into an array" do
    s = "a:b:c"
    s.split(':').should be_an Array
  end
  it "splits the string at each occurence of the given argument" do
    s = "a:b:c"
    s.split(':').should == ["a", "b", "c"]
  end
end
```

À vous : écrivez un test spécifiant le comportement de `Array#join` dans le fichier `array_spec.rb`. Déroulez ensuite le test :

```
rspec array_spec.rb
```

Hook

Pour factoriser du code qui revient dans plusieurs exemples, on utilise :

`before(:each)` # code exécuté avant chaque exemple du groupe

`before(:all)` # code exécuté une fois avant tous les exemples

`after(:each)`

`after(:all)`

Pour réutiliser une variable définie dans un `before` dans l'écriture d'un exemple (bloc `it`), il faut utiliser une variable d'instance (`@nom_variable`).

À vous : refactorisez les 2 tests précédents (sur `String#split` et `Array#join`) pour enlever les redites

```
<pre 'code'>
```

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

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

  describe "initialized in before(:each)" do
    it "has 0 widgets" do
      @thing.should have(0).widgets
    end

    it "can accept new widgets" do
      @thing.widgets << Object.new
      @thing.widgets.size.should == 1
    end

    it "does not share state across examples" do
      @thing.should have(0).widgets
    end
  end
end
```

Sujet d’un exemple

Au lieu d’initialiser soi-même la variable à tester dans un exemple, on peut utiliser `subject` comme objet sur lequel vont porter les expectations.

Implicitement, c’est une instance de la classe décrite.

```
<pre 'code'>
```

```
require 'person'
describe Person do
  context "when just created" do
    it "should have a blank name" do
      subject.name.should == ""
      # subject == Person.new
    end
  end
end
```

Quand `should()` ou `should_not()` sont appelés sans objet, l’objet par défaut est `subject` :

```
<pre 'code'>
```

```
describe Array do
  describe "when first created" do
    it { should be_empty }
  end
end
```

Sujet d’un exemple

On peut initialiser avec un bloc `subject`.

```
<pre 'code'>
```

```
describe Array, "with some elements" do
  subject { [1,2,3] }
  it "should have the prescribed elements" do
    subject.should == [1,2,3]
  end
end
```

Fixer le sujet à un attribut

```
<pre 'code'>
```

```
describe Person do
  context "when just created" do
      its(:name) { should == "" }
  end
end
```

À vous

Écrire la spécification d'une personne possédant ces caractéristiques:

- un nom
- un prénom
- un identifiant
- une méthode de validation devant vérifier que le nom, le prénom, et l'identifiant ne sont pas vides
- nom, prénom et identifiant sont vides initialement, mais peuvent être renseignés en utilisant un accesseur.

Remarques :

- code dans `person.rb`, test dans `person_spec.rb`
- le test d'abord, puis le code
- juste ce qui est demandé

mock/stub

Pour tester une seule chose à la fois, il faut que chaque test soit isolé. Si le fonctionnement de A dépend de B, et que je veux tester le fonctionnement de A, je dois être indépendant de B.

double/mock

crée le double d'un objet pour le remplacer dans un test.

```
double_person = double(Person) # ou mock
```

stub

remplace une méthode, renvoie des valeurs préenregistrées

```
object.stub(:method_name) { return_value }
double_person.stub(:name) { "Bob" }
```

message expectation

assertion sur l'appel et la valeur de retour d'une méthode

```
object.should_receive(:method_name).with(argument).and_return(value)
double_person.should_receive(:name).and_return("Bob")
```

On définit l'expectation sur l'appel de la méthode AVANT d'appeler la méthode.

Exemple

<pre 'code'>

```
describe Greeter do
  context "when created with a reference to a person" do
    before(:each) do
      @person = double("person")
      @person.stub(:last_name) { "Eponge"}
      @person.stub(:first_name) {"Bob"}
      @greeter = Greeter.new(@person)
    end
    it "should say hello to a person with her full name" do
      @greeter.hello.should == "Hello Bob Eponge"
    end
    it "should use the person first_name" do
      @person.should_receive(:first_name).and_return(@person.first_name)
      @greeter.hello
    end
  end

  context "when created with no person" do
    subject { Greeter.new }
    it "should say Hello Stanger" do
      subject.hello.should == "Hello Stranger"
    end
  end
end
```

<pre 'code'>

```
class Greeter
  def initialize(person = nil)
    @person = person
  end

  def hello
    if @person.nil?
      "Hello Stranger"
    else
      "Hello #{@person.first_name} #{@person.last_name}"
    end
  end
end
```

À vous

Rajoutez la gestion d'un mot de passe à la classe `Person`. Le mot de passe ne doit pas être stocké en clair, mais sous la forme d'une signature SHA1.

- difficulté technique : renseignez-vous sur l'utilisation de SHA1 avec ruby
- spécifiez avec des tests le comportement de l'accesseur `Person#password=`. En particulier, testez l'appel au code réalisant le chiffrage du mot de passe.
- écrivez le code permettant de passer les tests

Développement dirigé par les tests

TDD/BDD : Test/Behaviour Driven Development

1. Red : écriture du test, on exécute, le test échoue
2. Green : on écrit le code nécessaire pour que le test passe
3. Refactor : on améliore l'écriture, du test d'abord, du code ensuite.

Nous avons suivi les 2 premiers principes pour l'écriture de la classe `Person`.

Refactorisation

Essayez de suivre ces principes dans vos codes :

- ne vous répétez pas : que ce soit dans les tests ou dans le code, tout répétition de code doit disparaître.
- les objets d'une classe doivent avoir le moins de responsabilités possible. Par exemple : une classe représentant une personne ne devrait rien savoir des techniques de codage d'un mot de passe ...
- utilisez les modules pour encapsuler un comportement
- refactoriser les tests avant le code
- ne mettez pas plus dans votre code que ce qui est nécessaire pour passer les tests.

Avantages :

- le code est couvert par les tests
- la spécification de votre code est écrite : ce sont les tests
- les exemples d'utilisation de votre code sont documéntés : ce sont les tests !
- votre code est facilement modifiable : si une nouvelle fonctionnalité introsuite casse l'existant, un test vous dit immédiatement laquelle et à quel endroit
- vous avez confiance dans votre code

Inconvénient :

- l'écriture des tests a un coût immédiat : 30% environ de temps supplémentaire. Mais vous testiez déjà votre code non ? :-)

À vous

- Séparez la gestion des mots de passes de la classe `Person` dans une autre classe.
- Spécifiez cette autre classe, et refactorisez les tests et le code de la classe `Person` pour spécifier et coder l'utilisation de cette nouvelle classe.
- Permettez en particulier l'utilisation de plusieurs encodages différents pour le mot de passe.

Pour démarrer, repartez d'un nouveau projet en clonant un dépôt contenant déjà la classe personne première version :

```
git clone git://github.com/PierreGambarotto/tp_rspec.git
cd tp_rspec
```

Créer une nouvelle branche pour y développer la fonctionnalité :

```
git checkout -b "gestion du mot de passe"
```

Travaillez dans cette branche jusqu'à la fin de l'implémentation, puis une fois que tout marche, fusionnez cette branche dans la branche principale.

Ruby : require et $:

Inclusion de fichier

Pour inclure un fichier contenant du code ruby dans un autre, il faut utiliser `require`

```
<pre 'code'>

# fichier a.rb
require 'b' # inutile de mettre b.rb
```

Répertoires de recherche des fichiers à inclure

Si vous ne spécifiez pas un chemin d'accès, le fichier est cherché dans les répertoires dont le nom figure dans la variable globale `$:`, qui est un tableau.

```
<pre 'code'>

puts $:.inspect # inspect renvoie une chaine de caracteres representant l'objet

$:.unshift "/nouveau/chemin/de/recherche" # ajout en tete du tableau
$: << "autre/chemin"
```

File

```
puts __FILE__ # chemin complet du fichier courant
File.dirname(__FILE__) # repertoire contenant le fichier courant
```