Leonardo Querzoni
querzoni@dis.uniroma1.it

SAPIENZA
UNIVERSITÀ DI ROMA

CIS SAPIENZA
CYBER INTELLIGENCE AND INFORMATION SECURITY

DIPARTIMENTO DI INGEGNERIA
INFORMATICA AUTOMATICA E
GESTIONALE ANTONIO RUBERTI

RSPEC - ESEMPIO

Vogliamo progettare una piccola calcolatrice che svolge solo somme.

Creiamo una cartella per il progetto Ruby e all'interno definiamo un gemfile in cui inseriamo le dipendenze:

```
# Gemfile
source “https://rubygems.org"

gem “rspec"
```

Quindi risolviamo le dipendenze installando le gemme necessarie

```
$ bundle install
```

RSPEC - ESEMPIO

Creiamo una cartella per i nostri test

```
$ mkdir spec
```

E definiamo un primo test vuoto

```
# spec/string_calculator_spec.rb
describe StringCalculator do
end
```

Questo test definisce solo la necessità di avere un oggetto di nome StringCalculator

RSPEC - ESEMPIO

Infatti eseguendo il test…

```
$ bundle exec rspec

An error occurred while loading ./spec/string_calculator_spec.rb.
Failure/Error:
  describe StringCalculator do
  end

NameError:
  uninitialized constant StringCalculator
  Did you mean?  StringScanner
# ./spec/string_calculator_spec.rb:2:in `<top (required)>'
No examples found.
Finished in 0.00032 seconds (files took 0.20865 seconds to load)
0 examples, 0 failures, 1 error occurred outside of examples
end
```

RSPEC - ESEMPIO

Creiamo una cartella per il nostro codice

```
$ mkdir lib
```

E definiamo una classe StringCalculator

```
# lib/string_calculator.rb
class StringCalculator
end
```

Aggiungiamo anche una dipendenza in cima al file di test

```
# spec/string_calculator_spec.rb
require "string_calculator"
[…]
```

RSPEC - ESEMPIO

Eseguendo nuovamente il test…

```
$ bundle exec rspec
No examples found.


Finished in 0.00034 seconds (files took 0.16141 seconds to load)
0 examples, 0 failures
```

Rspec trova correttamente l'oggetto da testare, ma nessun test è definito.

RSPEC - ESEMPIO

La cosa più semplice che possiamo chiedere alla nostra calcolatrice è di accettare una string vuota in input.

```
# spec/string_calculator_spec.rb
describe StringCalculator do

  describe ".add" do

    context "given an empty string" do

      it "returns zero" do
        expect(StringCalculator.add("")).to eql(0)
      end

    end

  end

end
```

RSPEC - ESEMPIO

Eseguendo il test…

```
$ bundle exec rspec
F

Failures:
  1) StringCalculator.add given an empty string returns zero
     Failure/Error: expect(StringCalculator.add("")).to eql(0)

     NoMethodError:
       undefined method `add' for StringCalculator:Class
     # ./spec/string_calculator_spec.rb:9:in `block (4 levels) in <top (required)>'

Finished in 0.00368 seconds (files took 0.15808 seconds to load)
1 example, 1 failure

Failed examples:
rspec ./spec/string_calculator_spec.rb:8 # StringCalculator.add given an empty string
returns zero
```

RSPEC - ESEMPIO

Aggiungiamo il codice MINIMO necessario per soddisfare il test

```
# lib/string_calculator.rb
class StringCalculator

  def self.add(input)
    0
  end

end
```

Con questo codice il test esegue correttamente, ma la nostra calcolatrice non fa nulla di utile…

RSPEC - ESEMPIO

Definiamo degli esempi di funzionamento semplici…

```
# spec/string_calculator_spec.rb
describe StringCalculator do

  describe ".add" do
    […]
    context "given '4'" do
      it "returns 4" do
        expect(StringCalculator.add("4")).to eql(4)
      end
    end

    context "given '10'" do
      it "returns 10" do
        expect(StringCalculator.add("10")).to eql(10)
      end
    end
  end
end
```

RSPEC - ESEMPIO

E aggiungiamo il relativo codice MINIMO

```
# lib/string_calculator.rb
class StringCalculator

  def self.add(input)
    if input.empty?
      0
    else
      input.to_i
    end
  end

end
```

Con queste iterazioni possiamo andare ad aggiungere funzionalità definendole come esempi nei test!

RSPEC - ESEMPIO

Definiamo altri esempi di funzionamento più complessi

```
# spec/string_calculator_spec.rb
describe StringCalculator do

  describe ".add" do
    […]
      context "given '2,4'" do
        it "returns 6" do
          expect(StringCalculator.add("2,4")).to eql(6)
        end
      end

      context "given '17,100'" do
        it "returns 117" do
          expect(StringCalculator.add("17,100")).to eql(117)
        end
      end
  end
end
```

RSPEC - ESEMPIO

Modifichiamo la classe in accordo con i test

```ruby
# lib/string_calculator.rb
class StringCalculator

  def self.add(input)
    if input.empty?
      0
    else
      numbers = input.split(",").map { |num| num.to_i }
      numbers.inject(0) { |sum, number| sum + number }
    end
  end
end
```

STUBS, MOCKS AND SPIES

Concettualmente testare il software si riduce alle seguenti attività:

- ARRANGE: configurare correttamente l'ambiente per eseguire il test
  - creare oggetti di supporto?
  - Inizializzare componenti esterni?
- ACT: fornire uno "stimolo" al modulo in fase di test
  - chiamare un metodo?
  - richiedere una pagina con un metodo GET/POST?
- ASSERT: verificare che le conseguenze dello stimolo siano coerenti con quanto atteso

La fase ARRANGE è tipicamente la più complessa perché lascia molte scelte allo sviluppatore.

STUBS, MOCKS AND SPIES

Supponiamo di aver definito la seguente classe

```
class Detective
  def investigate
    "Nothing to investigate :'("
  end
end
```

Ed il relativo test

```
it "doesn't find much" do
  subject = Detective.new

  result = subject.investigate

  expect(result).to eq "Nothing to investigate :'("
end
```

STUBS, MOCKS AND SPIES

Aggiungiamo qualche funzionalità basilare

```
class Detective
  def initialize(thingie)
    @thingie = thingie
  end

  def investigate
    "It went '#{@thingie.prod}'"
  end
end
```

Testare questo codice richiede di avere un componente “thinghie” già testato e pronto all’uso…

Ma questo viola l’”indipendenza” dei test!!!

STUBS, MOCKS AND SPIES

Aggiungiamo qualche funzionalità basilare

```
class Detective
  def initialize(thingie)
    @thingie = thingie
  end

  def investigate
    "It went '#{@thingie.prod}'"
  end
end
```

Testare questo codice richiede di avere un componente “thinghie” già testato e pronto all’uso…

STUBS, MOCKS AND SPIES

Supponiamo di avere a disposizione questo oggetto

```
class Thingie
  def prod
    [ "erp!", "blop!", "ping!", "ribbit!" ].sample
  end
end
```

Potremmo usarlo nel nostro codice di test…

```
it "says what noise the thingie makes" do
  thingie = Thingie.new
  subject = Detective.new(thingie)

  result = subject.investigate

  expect(result).to match(/It went '(erp|blop|ping|ribbit)!'/)
end
```

STUBS, MOCKS AND SPIES

Problemi:

- Il test è difficile da comprendere:
  - Quale output viene da “Detective” e quale da “Thinghie”?
  - Dobbiamo conoscere a fondo “Thinghie” se il test funziona o no
- Il test è “fragile”
  - Che succede se Thinghie viene modificato aggiungendo un nuovo possibile output?
  - Cosa non funziona? Detective? La nuova versione di Thinghie? Il test?

Il problema deriva dal fatto che il nostro test non è indipendente da altri moduli

STUBS, MOCKS AND SPIES

Questo tipo di problemi può essere risolto con i “test double”.

Si tratta di oggetti “finti” a cui può essere assegnato un comportamento per i soli scopi del test.

Ne esistono di diversi tipi:

- **Dummy** – Un puro segnaposto che non fa nulla.
- **Fake** – Un oggetto di rimpiazzo che ha il comportamento di quello originale.
- **Stub** – Un oggetto che fornisce risposte prefissate a stimoli noti.
- **Mock** – Un oggetto a cui viene fornita una specifica del tipo di stimoli che riceverà e del tipo di risposte che ci si attende da lui durante il test.
- **Spy** – Un oggetto che tiene traccia di tutti gli stimoli ricevuti.

STUBS, MOCKS AND SPIES

Proviamo a migliorare il nostro test con uno stub

```
it "says what noise the thingie makes" do
  thingie = double(:thingie, prod: "oi")
  subject = Detective.new(thingie)

  result = subject.investigate

  expect(result).to eq "It went 'oi'"
end
```

Supponiamo che Detective debba eseguire .prod solo una volta.
Come possiamo testarlo?

STUBS, MOCKS AND SPIES

Rendiamo lo stub più complesso

```
it "prods the thingie at most once" do
  prod_count = 0
  thingie = double(:thingie)
  allow(thingie).to receive(:prod) { prod_count += 1 }
  subject = Detective.new(thingie)

  subject.investigate
  subject.investigate

  expect(prod_count).to eq 1
end
```

Ora il codice è difficile da interpretare…

STUBS, MOCKS AND SPIES

Usando un mock il test può essere semplificato

```
it "prods the thingie at most once" do
  thingie = double(:thingie)
  expect(thingie).to receive(:prod).once
  subject = Detective.new

  subject.investigate
  subject.investigate
end
```

Decisamente meglio ma l'organizzazione ARRANGE-ACT-ASSERT è persa…

STUBS, MOCKS AND SPIES

Usando un mock il test può essere semplificato

```
it "prods the thingie at most once" do
  # Arrange
  thingie = double(:thingie)
  # Assert
  expect(thingie).to receive(:prod).once
  # Arrange
  subject = Detective.new(thingie)

  # Act
  subject.investigate
  subject.investigate
end
```

Usare un oggetto spy ci permette di semplificare ulteriormente

STUBS, MOCKS AND SPIES

```
it "prods the thingie at most once" do
  # Arrange
  thingie = double(:thingie, prod: "")
  subject = Detective.new(thingie)

  # Act
  subject.investigate
  subject.investigate

  # Assert
  expect(thingie).to have_received(:prod).once
end
```