TEST DRIVEN DEVELOPMENT

Marco Console
console@dis.uniroma1.it

NOTA PRELIMINARE

- Il seguente materiale didattico è basato su un insieme di lucidi preparato dal **Prof. Leonardo Querzoni.**

- Ringrazio il **Prof. Leonardo Querzoni** per avermi concesso di usare il materiale da lui prodotto.

TESTING

QUALITY

- **La nozione di qualità è fondamentale in tutti i processi produttivi.**

- Quality is fitness for use. **(Juran and Gryna 1998)**
  - Qualità significa idoneità all’uso.

- La qualità è un prodotto della Quality Assurance.
  - Processi e standard per assicurare la qualità del prodotto.

- Quando un software può dirsi di qualità?

QUALITÀ DEL SOFTWARE

- **Qualità del software è spesso sinonimo di:**
  1. Soddisfare i bisogni del committente.
  2. Essere facile da manutenere e aggiornare.

- Quality Assurance nel software:
  1. Assicurarsi di produrre software di qualità.
  2. Stabilire quali processi assicurano un software di qualità.

- Spesso un QA team è delegato ai controlli di qualità.
  - Agile sovverte questo modello, come vedremo.

QUALITÀ DEL SOFTWARE

- La qualità del software passa per due concetti fondamentali.

- **Verification**: did you build the thing right?
  - Lo hai costruito bene?
  - Il software rispetta tutte le specifiche?
  - Ci sono bug?

- **Validation**: did you build the right thing?
  - Hai costruito quello giusto?
  - Il software fa quello che vuole il committente?
  - La specifica era corretta?

APPROCCI ALLA QUALITÀ

- Gli approcci a Verification e Validation sono essenzialmente due (specialmente in Agile)

- **Prototyping.**
  - Costruire diverse versioni del software da mostrare al committente.
  - Produrre prototipi poco raffinati è meno costoso di produrre versioni imperfette dell'applicazione finale.

- **Testing.**
  - Collaudare le varie parti del software per verificarne le funzionalità.
  - Prima i bug vengono scovati meno impatteranno sul costo finale.

TESTING -- PROFONDITÀ

- **In ordine inverso di profondità.**
  1. **Unit test (Test Unitario)**
     - Collaudo delle single procedure software nel codice.
  2. **Functional (module) Test**
     - Collaudo dei singoli moduli software.
  3. **Integration Test**
     - Collaudo dell'integrazione fra più moduli.
  4. **System Testing** (anche acceptance).
     - Test delle funzionalità dell'intera applicazione.

- **Ogni livello delega al precedente i dettagli.**

DOMANDA…

- **Quali sono i test collegati alla Verification?**

- **Quali test sono collegati alla Validation ?**

DOMANDA… SOLUZIONE

- **Quali sono i test collegati alla Verification?**
  - **Unit, Module, Integration, System.**

- **Quali test sono collegati alla Validation ?**
  - **Integration, System.**

TESTING IN AGILE

“Our highest priority is to satisfy the customer through early and continuous delivery of valuable software”

http://agilemanifesto.org/principles.html

“La nostra priorità è quella di soddisfare il committente attraverso veloci e continui rilasci di software di qualità”

http://agilemanifesto.org/principles.html

TESTING CLASSICO

- Metodologie «**Plan-and-Document**».
  - Waterfall, Spiral, or Rational Unified Process.

- Un team di Quality Assurance è delegato al testing.
  - Gli sviluppatori non sono direttamente coinvolti.
  - Gli sviluppatori non sono responsabili della qualità
  - Rilascio solo dopo l’ok del QA team o nuova iterazione di sviluppo.

TESTING CLASSICO -- CRITICITÀ

- Gli sviluppatori non sono direttamente coinvolti.
  - I tester non sono a conoscenza dei dettagli delle implementazioni.
  - Quali test effettuare? Bassa «copertura»
  - Come risolvere i problemi? Il QA team non conosce gli internals.
- Gli sviluppatori non sono responsabili della qualità
  - Spesso gli sviluppatori non si preoccupano della qualità.
- Rilascio dopo l’ok del QA team.
  - I rilasci potrebbero essere rallentati da successive iterazioni.

TESTING IN AGILE

- In agile il QA team non esiste separato dagli sviluppatori.
  - Ogni sviluppatore è incaricato di testare il proprio codice.

- La definizione dei test è parte della progettazione.
  - Durante la fase di analisi dei requisiti si definiscono anche i test.
  - Prima dello sviluppo!

- Approccio Test-Driven Development (TDD)
  - Sviluppo guidato dalle verifiche.

- In Agile la qualità del software non è il risultato di una serie di test ma un processo produttivo.

TEST-DRIVEN DEVELOPMENT

- A partire dalle User Stories (BDD)
  1. Produrre Acceptance Tests e Integration tests.
  2. Scrivere Unit Test sul codice che intendiamo scrivere.
  3. Scrivere abbastanza codice perché i test passino.

- Isolare le funzionalità testate dal resto del codice
  - Seams (cuciture) pezzi di codice che rimpiazzano quelli reali.
  - Per isolare i bug è meglio testare il codice in isolamento.

- Analizzare la copertura.
  - Quanto codice è coperto dai test?

TDD WORKFLOW

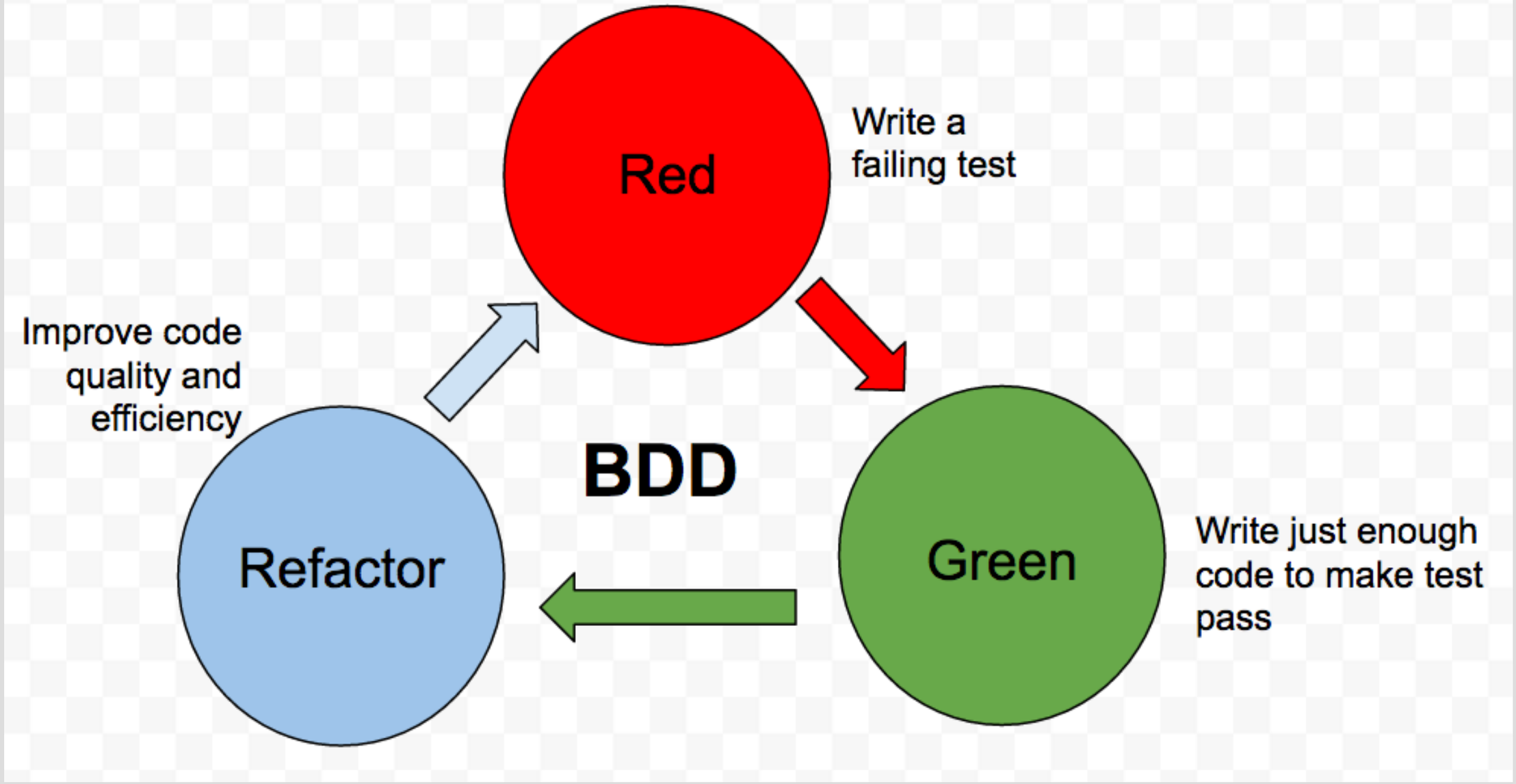
- TDD non è una metodologia di test ma di sviluppo
  - Red-Green-Refactor

- Prima di scrivere il codice, scrivere i test.

1. **Red:** i test falliranno.

2. **Green:** scrivere il più semplice pezzo di codice che fa si che i test passino.

3. **Refactor:** migliorare la qualità del codice e dei test durante l’esecuzione.


Red
Write a failing test
Green
Write just enough code to make test pass
Refactor
Improve code quality and efficiency
BDD

CARATTERISTICHE DEI TEST

- Test FIRST:
  - **Fast**: veloci da eseguire.
    - I test potrebbero essere lanciati tantissime volte.
  - **Independent**: i test sono indipendenti tra di loro.
    - Potremmo voler lanciare solo un sottoinsieme dei test alla volta.
  - **Repeatable**: il risultato dei test dipende solo dal codice esercitato.
    - Per isolare più chiaramente i bug.
  - **Self-Checking**: il codice dei test può valutarne il risultato in autonomia.
    - La presenza di uno sviluppatore rallenterebbe il processo.
  - **Timely**: i test vanno creati prima del codice.
    - Il codice scritto in questa maniera tende ad essere più chiaro.

ACCEPTANCE TEST

E

INTEGRATION TEST

USER STORIES

- Una User Story descrive il comportamento dell’applicazione in diversi scenari.
  - La versione Agile deli **use-case** in plan-and-document

- Le User Story sono SMART
  - **Specific.** Descrivono il requisito senza ambiguità.
  - **Measurable.** Possiamo misurare se il requisito è stato implementato.
  - **Achievable.** In una iterazione di Agile.
  - **Relevant.** Il requisito aggiunge valore all’applicazione.
  - **Timeboxed.** A ogni User Story è allocato un «time budget» oltre il quale la sua implementazione viene abbandonata o la U.S. Viene spezzata in U.S. più semplici

USER STORIES IN CUCUMBER

- User stories in Cucumber sono composte da:

- **Feature.**
  - La funzionalità che la U.S. Rappresenta.

- **Scenario.**
  - Uno specifico caso d’uso.
  - Più di uno Scenario per Feature è ammesso.

TESTS E USER STORIES

- I requisiti raccolti dai committenti sono di due tipi.
  - **Requisiti Espliciti.** Quelli dettati dal committente.
    - Acceptance test.
  - **Requisiti Impliciti.** Conseguenze di quelli impliciti.
    - Integration test.

- Le user stories possono catturare entrambi i requisiti
  - Le U.S. dei requisiti impliciti vanno definiti dal progettista.
  - Dalle U.S. possiamo definire Acceptance test e Integration test.

- **Come?**

USER STORIES COME TEST

- Come definire user story che possono diventare test?

- Ogni U.S. deve riferirsi a una singola funzionalità
  - Altrimenti i test non sarebbero specifici.

- Ogni scenario descrive un singolo esempio di utilizzo della funzionalità in oggetto.
  - Tutti i passi necessari devono essere descritti
  - Il risultato deve essere chiaramente definito.

- Gli esempi descritti negli Scenario verranno testati sull’applicazione.

ESEMPIO – U.S. COME TEST

- **Requisito**: Voglio aggiungere un film al catalogo.

- **Feature** : User can manually add movie
- **Scenario** : Add a movie

1. **Given** I am on the RottenPotatoes home page
2. **When** I follow " Add new movie "
3. **Then** I should be on the Create New Movie page
4. **When** I fill in " Title " with " Men In Black "
5. **And** I select "PG -13 " from " Rating "
6. **And** I press " Save Changes "
7. **Then** I should be on the RottenPotatoes home page
8. **And** I should see " Men In Black"

ESEMPIO – U.S. COME TEST

- **Requisito**: I film devono essere visualizzati in ordine alfabetico anche se sono stati inserito in un altro ordine.

- **Feature** : Movies should appear in alphabetical order, not added order
- **Scenario** : View Movie List after adding movies

1. **Given** I have added “Zorro” with rating “PG-13”.
2. **And** I have added “Apocalypse Now” with rating “R”.
3. **Then** I should see “Apocalypse Now” before “Zorro” on the RottenPotoes Homepage.

DOMANDE…

- Che tipi di requisiti erano quelli visti nell’esempio?

- Quanto codice dell’applicazione dovremmo avere per lanciare questi test?

- Le funzionalità esercitate dovrebbero essere isolate dalle loro componenti?

DOMANDE...SOLUZIONE

- Che tipi di requisiti erano quelli visti nell'esempio?
  - Voglio aggiungere un film al catalogo. Esplicito.
  - Ordine alfabetico. Implicito.
- Quanto codice dell'applicazione dovremmo avere per lanciare questi test?
  - Secondo la metodologia TDD, dovremmo definire prima i test poi scrive il codice.
- Le funzionalità esercitate dovrebbero essere isolate dalle loro componenti?
  - No. Test di accettazione e integrazione dovrebbero esercitare le parti dell'applicazione in modo sinergico.
  - Funzionalità esterne all'applicazione (come l'interazione con internet) potrebbero essere rimpiazzate con dei seam.

FUNCTIONAL TEST

E

UNIT TEST

STRUTTURA DI UN TEST

- Concettualmente, i test dovrebbe consistere in tre fasi separate.

- **Arrange**: creare le precondizioni per la funzionalità da esercitare.
  - creare oggetti di supporto?
  - Inizializzare componenti esterni?

- **Act**: esercitare la funzionalità.
  - chiamare un metodo?
  - richiedere una pagina con un metodo GET/POST?

- **Assert**: controllare il risultato.

LEAF E NON-LEAF METHODS

- Unit test e Function Test si concentrano sull'esercizio di specifici metodi nell'applicazione.

- **Leaf Method**: il risulato del metodo dipende solo dal suo codice.

- **Non-Leaf Method**: il risulato del metodo **non** dipende solo dal suo codice.

LEAF METHOD

- I Leaf method non sono necessariamente metodi semplici.
  - Possono avere comportamenti differenti dipendentemente dall’input.

- I Leaf method possono essere oggetto di vari test.
  - Uno per ogni comportamento.
  - Separare gli input in classi di «comportamento atteso»

LEAF METHOD -- ESEMPIO

```
1   def pn (x)
2     if x < 0
3       Do Something
4     elsif x > 0
5       Do Another Something
6     else
7       Conclude
8     end
9
10  end
11
```

- Tre «classi» di input.
  - >0
  - <0
  - 0

NON-LEAF METHOD

- Quando un metodo è non-leaf?
  - Il metodo invoca altri metodi.
  - Il metodo ha side-effect (cambia lo stato dell’applicazione)
  - Il risultato del metodo dipende da fattori impliciti (data, random...)

- In Ruby+Rails non-leaf methods sono spesso relativi ai controller.

- Come esercitiamo un metodo non-leaf?
  - Cerchiamo di isolare il suo comportamento

NON-LEAF METHOD -- ESEMPIO

```ruby
def pn ()
  y = MyClass.new
  x = a_method() + y.another_method()
  if x < 0
    puts "negative"
  elsif x > 0
    puts "positive"

  else
  puts "Zero!"
  end

end
```

- Come esercitiamo i tre comportamenti?

SEAMS

- Il risultato del test dovrebbe isolare il codice dalle sue dipendenze.

- Per isolare un non-leaf method usiamo **seams**
  - Pezzi di codice che rimpiazzano quello reale
  - Utile anche per TDD in fase di sviluppo. Perché?

- **Method stub**: un metodo fittizio che rimpiazza il comportamento di uno reale con un input controllabile.

- **Mock object**: un oggetto fittizio che rimpiazza il comportamento di uno reale ritornando valori fissi.

SEAMS -- ESEMPIO

```
1  def r()
2    return 0
3  end
4
5  class MyClass
6   def m()
7    return 1
8   end
9  end
```

- Possiamo controllare il comportanto dei seams

STRATEGIE DI TESTING

- **Se il metodo…**
- **È Leaf**
  - Testare le differenti classi di input.
- **Invoca metodi di supporto**.
  - Creare metodi stub per forzare il comportamento desiderato.
- **Comportamento non deterministico.**
  - I solare il comportamento non deterministico in un singolo metodo durante la fase di progettazione.
  - Definire uno stub per il metodo.

COVERAGE

QUANTO TEST?

- E’ complicato dire quanto test sia abbastanza.
- «Il più possibile prima della deadline»
  - Spesso è la risposta che si ottiene.

METRICHE PER UNIT TESTS

- **Code-to-test ratio** (rapporto codice-a-test)
  - $\frac{Righe\ di\ codice}{Righe\ di\ test}$
  - Appropriata?

- **Code Coverage** (Copertura del codice)
  - Metriche puntuali di copertura
  - Differenti livelli di copertura
  - Non è sempre facile raggiungerli.

CODE COVERAGE – LIVELLI

- **Method Coverage (S0):**
  - tutti i metodi sono eseguiti almeno una volta

- **Call Coverage (S1):**
  - tutti i metodi sono invocati da tutti i punti possibili.

- **Statement Coverage (C0):**
  - tutti gli statement sono eseguiti almeno una volta
  - Gli statement condizionali contano una sola volta

- **Branch Coverage (C1):**
  - tutti i branch vengono eseguiti.

CODE COVERAGE – ESEMPIO

```
1   class MyClass
2       def foo(x, y, z)
3           if x
4               if (y && z) then
5                   bar(0)
6               end
7           else
8               bar(1)
9           end
10      end
11      def bar(x) ; @w = x; end
```

CODE COVERAGE – LIVELLI

- **Method Coverage (S0):**
  - Il test invoca sia foo che bar

- **Call Coverage (S1):**
  - bar deve essere invocato da riga 5 e 8.

- **Statement Coverage (C0):**
  - foo(x, y, z) va chiamato con **`x = true`** `per eseguire lo statement condizionale a linea 4`

- **Branch Coverage (C1):**
  - **`x = true e x = false`**
  - **`(y && z) = true e (y && z) = false`**

DOMANDE...

- Alta Code Coverage significa che i test sono esaustivi?
- Qual è la differenza fra C0 e code coverage?

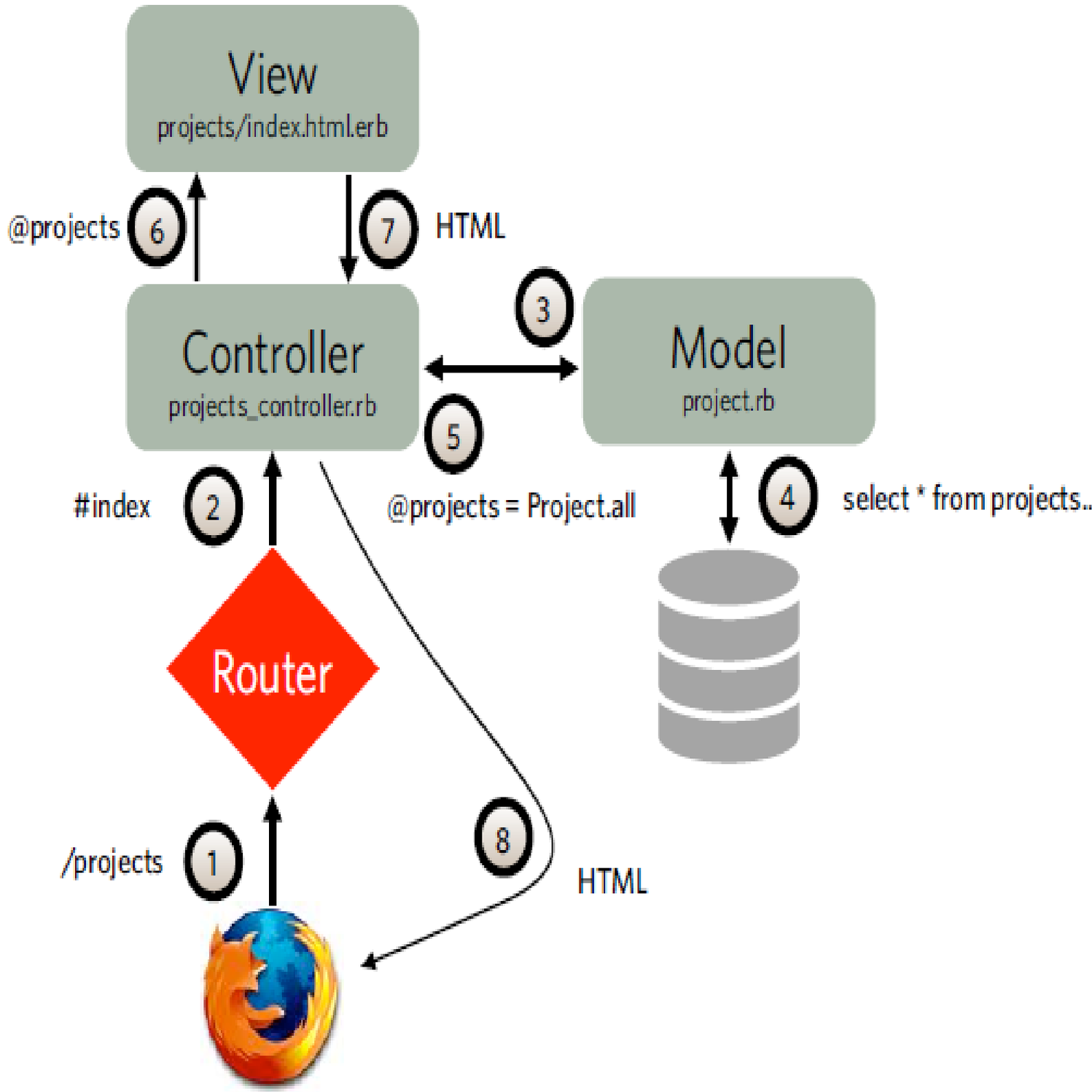
DOMANDE…SOLUZIONE

- Alta Code Coverage significa che i test sono esaustivi?
  - No. Code Coverage non riguarda il comportamento dell’applicazione
- Qual è la differenza fra C0 e code coverage?
  - C0 considera il numero di metodi invocati mentre code coverage è una misura di massima di quanto codice è stato scritto per i test.

TDD IN RUBY+RAILS

Request Handling
The Request-Response Pipeline

1. User requests /projects
2. Rails router forwards the request to projects_controller#index action
3. The index action creates the instance variable @projects by using the Project model all method
4. The all method is mapped by ActiveRecord to a select statement for your DB
5. @projects returns back with a collection of all Project objects
6. The index action renders the index.html.erb view
7. An HTML table of Projects is rendered using ERB (embedded Ruby) which has access to the @projects variable
8. The HTML response is returned to the User



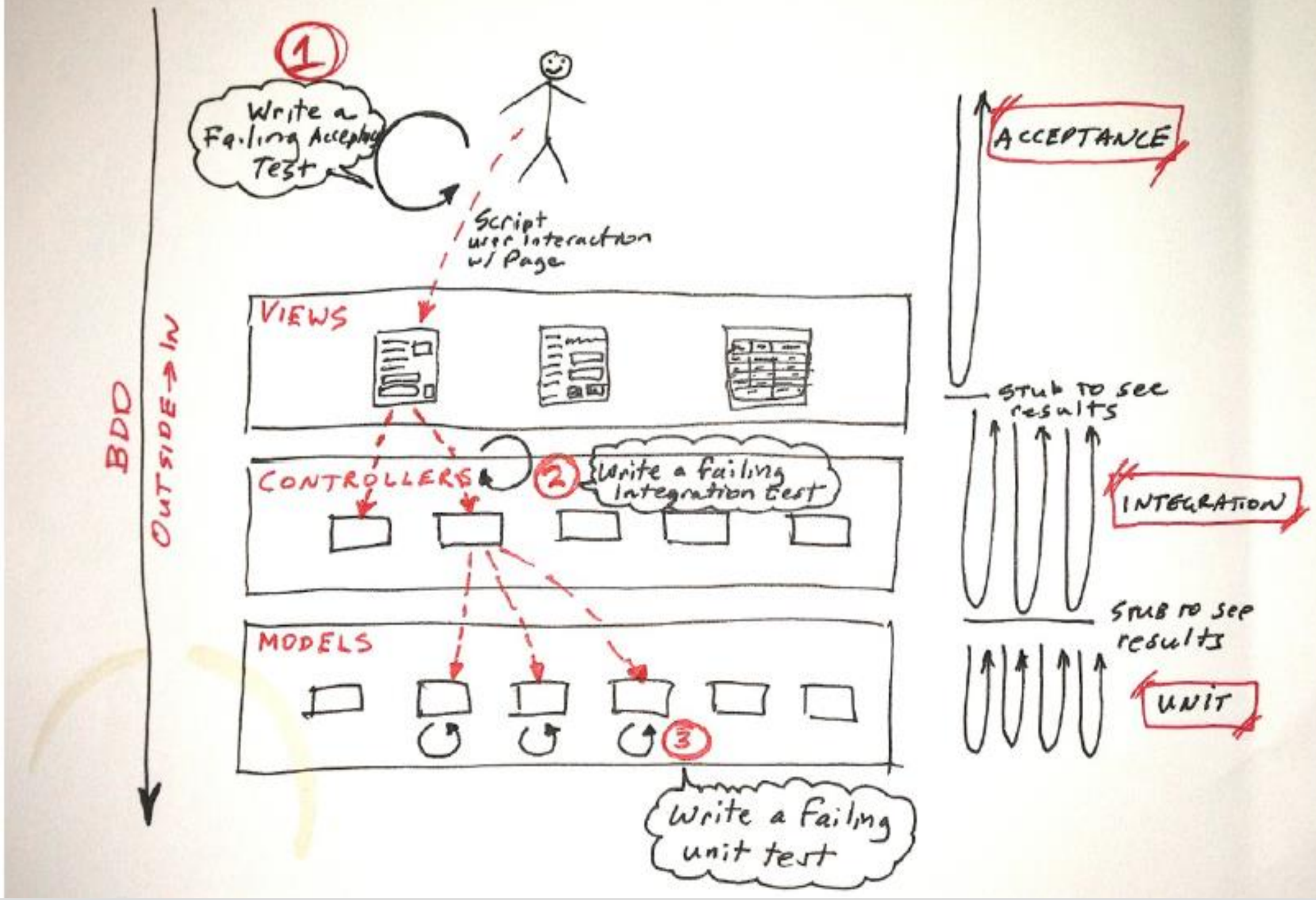
BDD IN RUBY+RAILS

- Per costruire un’applicazione come quella sopra procediamo come segue

1. Definiamo un insieme di user stories.

2. Per ogni user story implementiamo:
    1. Controllers
    2. Models
    3. Views

- Come e dove definire i test?


1
Write a Failing Acceptance Test
Script user interaction w/ Page
VIEWS
CONTROLLERS
2
Write a failing integration test
MODELS
3
Write a failing unit test
BDD
Outside→In
ACCEPTANCE
STUB TO SEE results
INTEGRATION
STUB TO SEE results
UNIT

BDD+TDD IN RUBY+RAILS

1. Definiamo un insieme di User Stories
   - Definiamo acceptance e integration test a partire dalle User Stories.
   - Per eseguire i test possiamo scrivere del codice fittizio che rimpiazza quello che ancora non abbiamo.

2. Per ogni user story
   1. Definiamo i test sul codice che vorremmo avere
   2. Scriviamo Model-View-Control

3. Analizzare la coverage dei test definiti.
   1. In caso, definire altri test.

RAILS E TDD

- Gli strumenti che RAILS ci offre per il testing sono:

- **Cucumber + Capybara** per acceptance e integration test.

- **RSpec** per Unit e Functional test.

- **SimpleCov** per la coverage

RSPEC

RSPEC – INTRO

- RSpec è un Domain Specific Langauge per la definizione di test.
  - Un piccolo linguaggio di programmazione per uno scopo specifico.
  - Esempi: regexes, SQL, …
- I test RSpec si chiamano **specs** o **examples**
- Per usare RSpec importiamo la gemma RSpec.
  - Nel Gemfile
- Per lanciare un test invochiamo rspec filename
  - Red failing, Green passing, Yellow pending

EXAMPLE GROUPS

- Una spec descrive uno o più esempi con il relativo comportamento atteso.

```
describe TemperatureControl, "check" do
        it "returns Gas if Temp is 100" do
            wtc = TemperatureControl.new("water")
            expect(wtc.check(100)).to eq "Gas"
        end
        it "returns Solid if Temp is 0" do
            wtc = TemperatureControl.new("water")
            expect(wtc.check(0)).to eq "Solid"
        end
end

```

CONTEXTS

- Le spec possono definire dei contesti d’uso

```
describe TemperatureControl, "check" do
        context 'We are testing water' do
            it "returns Gas if Temp is 100" do
                wtc = TemperatureControl.new("water")
                expect(wtc.check(100)).to eq "Gas"
            end
            it "returns Solid if Temp is 0" do
                wtc = TemperatureControl.new("water")
                expect(wtc.check(0)).to eq "Solid"
            end
        end
end

```

A TEST CLASS

```
1  class TemperatureControl
2      def initialize(material)
3          @material = material
4      end
5      def check(temp)
6          if temp > -1
7              return "Liquid"
8          else
9              return "Solid"
10         end
11     end
12 end
13
```

ESECUZIONE

```
Failures:

  1) TemperatureControl check We are testing water returns Solid if Temp is 0
     Failure/Error: expect(wtc.check(0)).to eq "Solid"

       expected: "Solid"
            got: "Liquid"
```

```
Finished in 0.00422 seconds (files took 0.14889 seconds to load)
2 examples, 0 failures
```

```
require 'ruby_intro.rb'

describe "BookInStock" do
  it "should be defined" do
    expect { BookInStock }.not_to raise_error
  end

  describe 'getters and setters' do
    before(:each)  { @book = BookInStock.new('isbn1', 33.8) }
    it 'sets ISBN' do
      expect(@book.isbn).to eq('isbn1')
    end
    it 'sets price' do
      expect(@book.price).to eq(33.8)
    end
    it 'can change ISBN' do
      @book.isbn = 'isbn2'
      expect(@book.isbn).to eq('isbn2')
    end
    it 'can change price' do
      @book.price = 300.0
      expect(@book.price).to eq(300.0)
    end
  end
```

```
#Without a valid movie
expect { m.save! }.
 to raise_error(ActiveRecord::RecordInvalid)

m = (create a valid movie)
expect(m).to be_valid
expect { m.save! }.
  to change { Movie.count }.by(1)
```

```
expect { lambda }.to(assertion)
expect(expression).to(assertion)
```

RSPEC+RAILS

- Rspec mette a disposizione funzionalità specifiche Rails.

- Metodi specifici per esercitare funzionalità Rails.
  - `get, post, put, ...` per esercitare l controller
  - Oggetti `response` per analizzare le risposte.

- Matchers specifici per Rails.

```
expect(response).to
   render_template("movies/index")
```

SEAMS IN RSPEC

- Rspec mette a disposizione una serie di seam.
  - Anche detti “test double”.
- **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.

SEAMS IN RSPEC

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
```

SEAMS IN RSPEC

- 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’isolamento dei test!!!

SEAMS IN RSPEC

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” per capire 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 isolato dagli altri moduli.

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
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
```

STUBS, MOCKS AND SPIES

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

  # Act
  subject.investigate

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

Usare un oggetto spy ci permette di semplificare ulteriormente

FIXTURES

- Rails mette a disposizione vari strumenti per il testing

- Le fixtures sono un metodo semplice per creare delle istanze dei modelli senza alterare la base di dati

- I file per le fixture vengono generati automaticamente per ogni modello quando eseguiamo **`rails g model`**
  - **`./test/fixtures`**

FIXTURES

- Una fixture è un modo per definire oggetti con valori fissi

```
marco:
  first_name: Marco
  last_name: Console
  phone: 555-123-6788
```

- Le fixtures sono definite in file yaml
  - test/fixtures/model_name.yaml

FIXTURES

```
describe User do
  fixtures :all

  describe "#full_name" do
    it "is composed of first and last name" do
      user = users(:marco)
      expect(user.full_name).to eql “Marco Console"
    end
  end
end
```