

RSpec & TDD 測試 Tutorial

<http://ihower.tw>

2014/12

About me

- 張文鈿 a.k.a. ihower
 - <http://ihower.tw>
 - <http://twitter.com/ihower>
- Instructor at ALPHA Camp
 - <http://alphacamp.tw>
- Rails Developer since 2006
 - i.e. Rails 1.1.6 era



Agenda

- Part I: RSpec and TDD
 - 什麼是自動化測試和 RSpec
 - Continuous Testing and Guard
 - TDD and Code Kata
 - Mocks and Stubs
- Part2: RSpec on Rails
 - model, controller, view, helper, routing, mailer, request spec
 - feature spec and capybara
 - Tools: spring, simplecov, cucumber
 - CI (Continuous Integration)

什麼是測試？



Search



Home

Profile

Messages

Who To Follow



ihower ▾



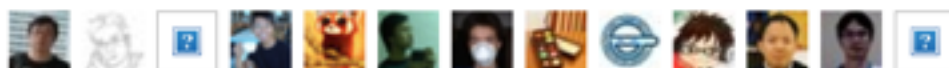
@ihower

Wen-Tien Chang

為什麼測試喜歡搞這麼多名目：單元測試、
驗收測試、需求測試、煙霧測試、回歸測
試、整合測試、系統測試、功能測試、確認
(validation)測試、白箱測試、黑箱測試、
灰箱測試、錯誤處理測試、壓力測試、效能
測試、安全性測試、相容性測試、使用性測
試、完整性測試、結構測試、安裝測試、

27 Sep via web ☆ Favorite ↻ Reply 🗑 Delete

Retweeted by sandzhang and 15 others



Sorry, I'm not QA guy :/

測試 Overview

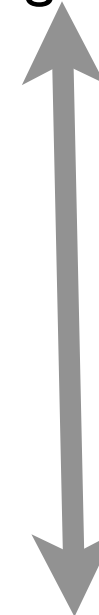
單元測試
Unit Test

整合測試
Integration Test

驗收測試
Acceptance Test

Verification

確認程式的執行結果有沒有對?
Are we writing the code right?



Validation

檢查這軟體有滿足用戶需求嗎?
Are we writing the right software?

單元測試
Unit Test

整合測試
Integration Test

驗收測試
Acceptance Test

測試個別的類別和函式結果正確

測試多個類別之間的互動正確

從用戶觀點測試整個軟體

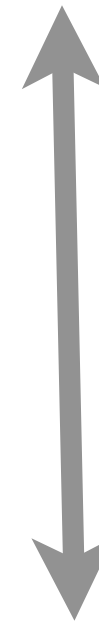


單元測試
Unit Test

整合測試
Integration Test

驗收測試
Acceptance Test

白箱測試



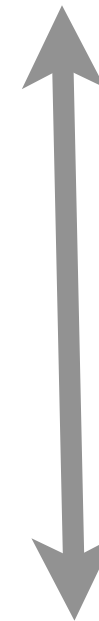
黑箱測試

單元測試
Unit Test

整合測試
Integration Test

驗收測試
Acceptance Test

Developer 開發者



QA 測試人員

Verification

確認程式的執行結果有沒有對?

Are we writing the code right?

我們天天都在檢查程式的結果有沒有對

- 程式寫完後，手動執行看看結果對不對、打開瀏覽器看看結果對不對？(如果你是web developer 的話)
- 聽說有人只檢查 `syntax` 語法沒錯就 `commit` 程式的？

手動測試很沒效率

特別是複雜或 dependency 較多的程式，你可能會偷懶沒測試所有的執行路徑

有三種路徑，你得執行(手動測試)三次才能完整檢查

```
def correct_year_month(year, month)
```

```
  if month > 12
```

```
    if month % 12 == 0
```

```
      year += (month - 12) / 12
```

```
      month = 12
```

```
    else
```

```
      year += month / 12
```

```
      month = month % 12
```

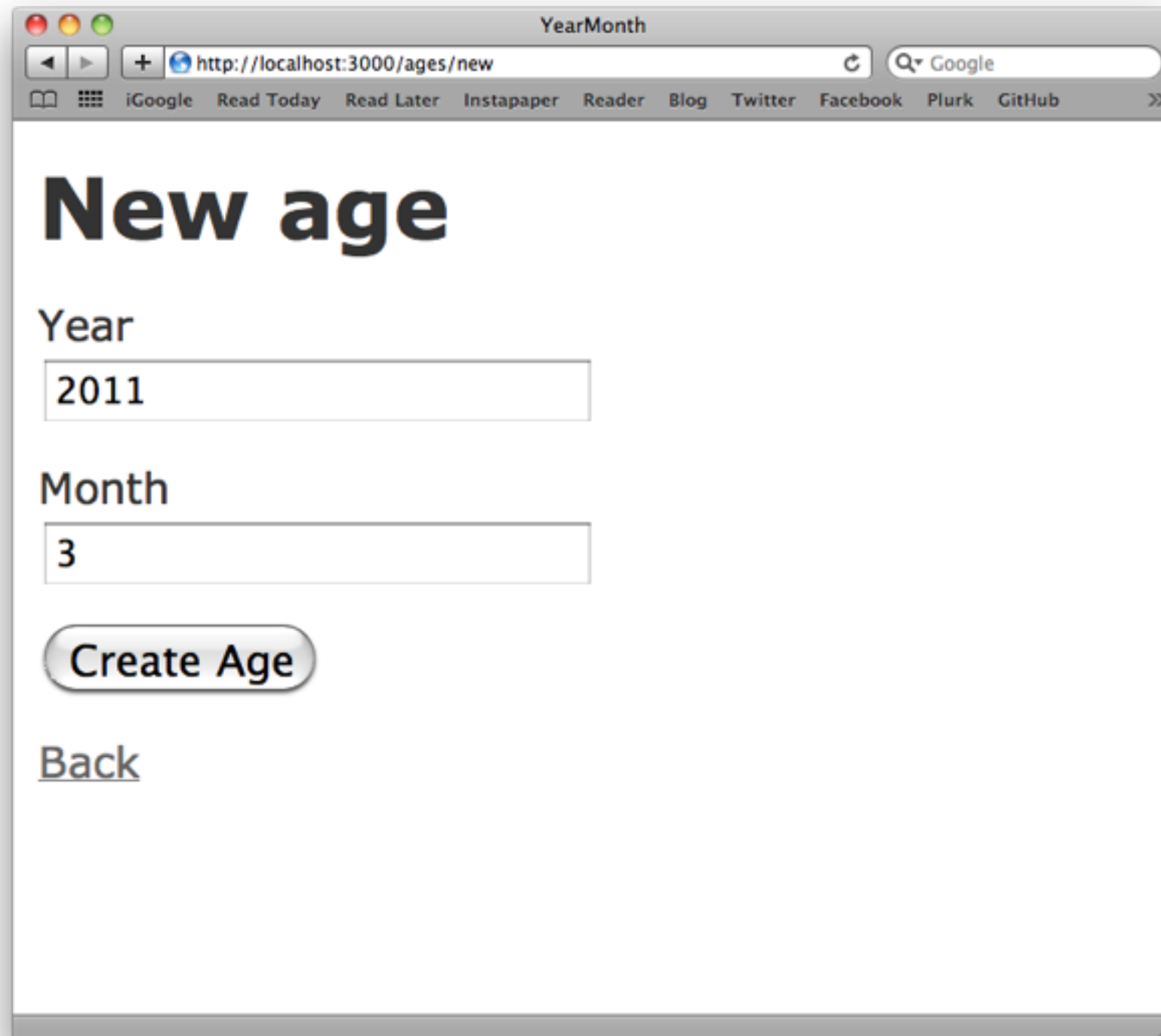
```
    end
```

```
  end
```

```
  return [year, month]
```

```
end
```

用介面手動測試?



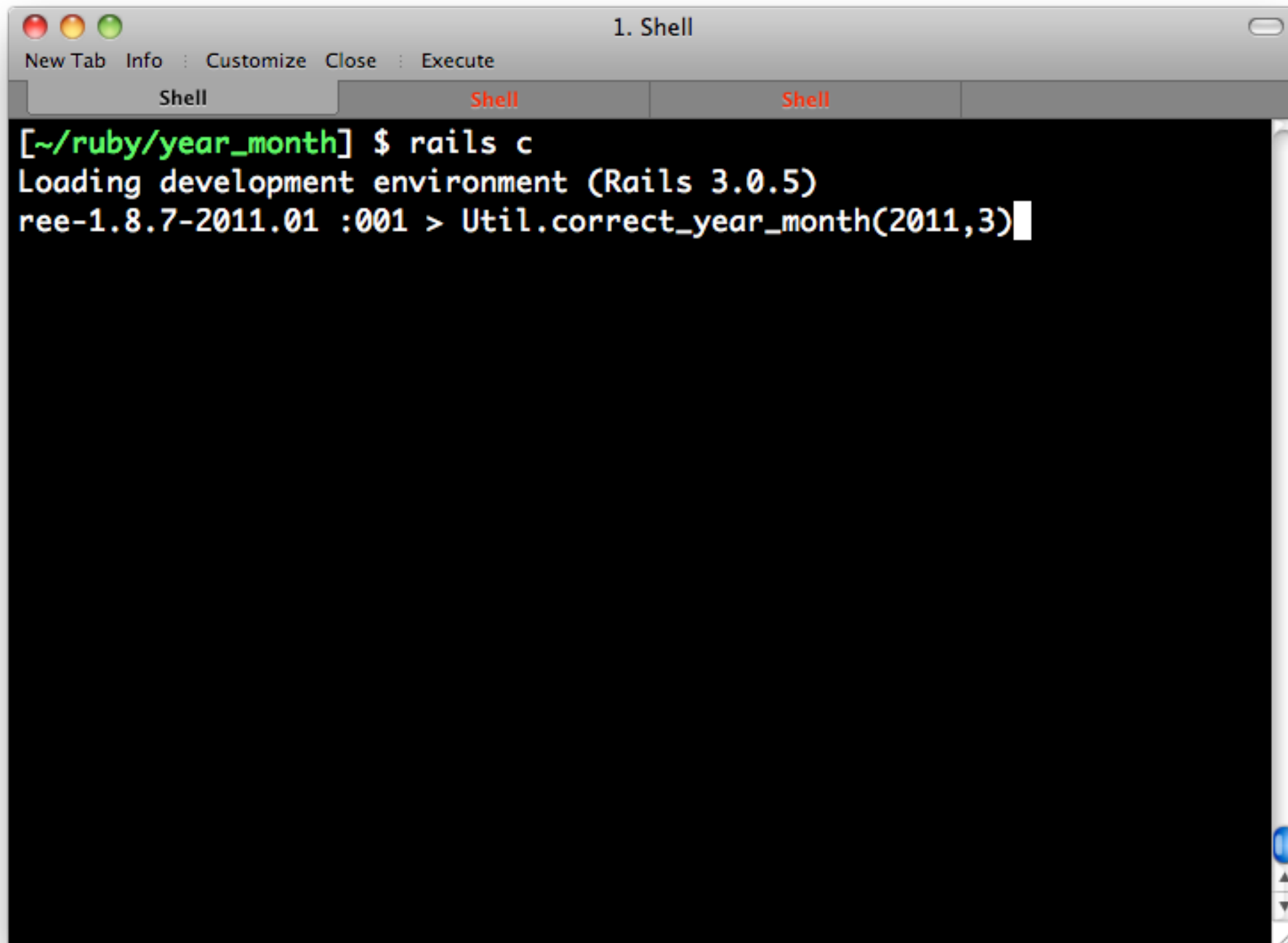
A screenshot of a web browser window titled "YearMonth". The address bar shows "http://localhost:3000/ages/new". The browser's toolbar includes a search bar with "Google" and a list of bookmarks: "iGoogle", "Read Today", "Read Later", "Instapaper", "Reader", "Blog", "Twitter", "Facebook", "Plurk", and "GitHub".

The main content area displays the heading "New age" in a large, bold font. Below this, there are two input fields:

- A "Year" label followed by a text input field containing the value "2011".
- A "Month" label followed by a text input field containing the value "3".

Below the input fields is a rounded button labeled "Create Age". At the bottom left of the form area is a link labeled "Back".

用 Console 介面手動測?

A screenshot of a web browser window displaying a Ruby on Rails console. The window has a title bar with three colored buttons (red, yellow, green) and the text "1. Shell". Below the title bar is a menu bar with "New Tab", "Info", "Customize", "Close", and "Execute". The main content area is a black console with white text. The text shows the user's prompt, the command entered, and the output of the command. The user's prompt is "[~/ruby/year_month] \$". The command entered is "rails c". The output is "Loading development environment (Rails 3.0.5)". The user's prompt is now "ree-1.8.7-2011.01 :001 >". The command entered is "Util.correct_year_month(2011,3)". The cursor is at the end of the command.

```
[~/ruby/year_month] $ rails c
Loading development environment (Rails 3.0.5)
ree-1.8.7-2011.01 :001 > Util.correct_year_month(2011,3)
```

自動化測試

讓程式去測試程式

I. Instant Feedback

寫好測試後，很快就可以知道程式有沒有寫對

```
it "should return correct year and month" do
```

```
  expect(correct_year_month(2011, 3)).to eq [2011, 3]
```

```
  expect(correct_year_month(2011, 14)).to eq [2012, 2]
```

```
  expect(correct_year_month(2011, 24)).to eq [2012, 12]
```

```
end
```

寫測試的時間
小於
debug 除錯的時間

2. 回歸測試及重構

- 隨著程式越寫越多，新加的程式或修改，會不會搞爛現有的功能？
- 重構的時候，會不會造成破壞？
- 如果有之前寫好的測試程式，那們就可以幫助我們檢查。

3. 幫助你設計 API

- 寫測試的最佳時機點：先寫測試再實作
- TDD (Test-driven development)
 - Red-Green-Refactor 流程
- 從呼叫者的角度去看待程式，關注介面，協助設計出好用的 API。

4. 一種程式文件

- 不知道程式的 API 怎麼呼叫使用?
- 查測試程式怎麼寫的，就知道了。

小結

- 你的程式不 Trivial -> 寫測試節省開發時間
- 你的程式不是用過即丟 -> 回歸測試
- TDD -> 設計出可測試，更好的 API 介面
- API 怎麼用 -> 測試也是一種文件

怎麼寫測試?

xUnit Framework

xUnit Framework

每個程式語言都有這樣的框架

- test case 依序執行各個 test :
 - Setup 建立初始狀態
 - Exercise 執行要測的程式
 - Verify (assertion) 驗證狀態如預期
 - Teardown 結束清理資料

Ruby 的 Test::Unit


```
class OrderTest < Test::Unit::TestCase
```

```
  def setup  
    @order = Order.new  
  end
```

```
  def test_order_status_when_initialized  
    → assert_equal @order.status, "New"  
  end
```

```
  def test_order_amount_when_initialized  
    → assert_equal @order.amount, 0  
  end
```

```
end
```



Method 的命名是
個麻煩

RSpec 寫法

```
describe Order do
  before do
    @order = Order.new
  end
```

```
  context "when initialized" do
    it "should have default status is New" do
      → expect(@order.status).to eq("New")
    end
```

```
    it "should have default amount is 0" do
      → expect(@order.amount).to eq(0)
    end
  end
end
```

RSpec 第一印象

- 語法 Syntax 不同
 - 程式更好讀
 - 更像一種 spec 文件

RSpec

- RSpec 是一種 Ruby 的測試 DSL
(Domain-specific language)
- Semantic Code：比 Test::Unit 更好讀，寫的人更容易描述測試目的
- Self-documenting：可執行的規格文件
- 非常多的 Ruby on Rails 專案採用 RSpec 作為測試框架

RSpec (cont.)

- 不是全新的測試方法論
- 是一種改良版的 xUnit Framework
- 演進自 TDD，改稱 BDD (Behavior-driven development)
 - 先寫測試，後寫實作

Learn RSpec

- syntax
- syntax
- syntax
- more syntax

describe 和 context
幫助你組織分類

要測的東西是什麼？

→ describe Order do
...
end

或是

describe "A Order" do
...
end

通常是一個
類別

可以 Nested 加入想要測試的方法是哪個

```
describe Order do
```

```
→ describe "#amount" do
  # ...
end
end
```

通常開頭用 # 表示
instance method
dot 開頭表示 class method

可以再 nested 加入不同情境

```
describe Order do
```

```
  describe "#amount" do
```

```
    → context "when user is vip" do
```

```
      # ...
```

```
    end
```

```
    → context "when user is not vip" do
```

```
      # ...
```

```
    end
```

```
  end
```

```
end
```

每個 it 就是一小段測試

Assertions 又叫作 Expectations

加入 it

```
describe Order do
```

```
  describe "#amount" do
```

```
    context "when user is vip" do
```

```
      → it "should discount five percent if total > 1000" do
        # ...
      end
```

```
      → it "should discount ten percent if total > 10000" do
        # ...
      end
    end
```

```
    context "when user is not vip" do
```

```
      → it "should discount three percent if total > 10000" do
        # ...
      end
    end
  end
```

```
end
```

expect(...).to, to_not

所有物件都有這個方法來定義你的期望

```
describe Order do
```

```
  describe "#amount" do
```

```
    context "when user is vip" do
```

```
      it "should discount five percent if total >= 1000" do
```

```
        user = User.new( :is_vip => true )
```

```
        order = Order.new( :user => user, :total => 2000 )
```

```
        → expect(order.amount).to eq(1900)
```

```
      end
```

```
      it "should discount ten percent if total >= 10000" { ... }
```

```
    end
```

```
  context "when user is vip" { ... }
```

```
end
```

```
end
```


輸出結果(red)

```
1. Shell
New Tab Info : Customize Close : Execute

[~/ruby] $ rspec order_spec.rb -fs

Order
  #amount
    when user is vip
      should discount five percent if total >= 1000 (FAILED - 1)
      should discount ten percent if total >= 10000 (FAILED - 2)
    when user is not vip
      should discount three percent if total > 10000 (FAILED - 3)

Failures:

1) Order#amount when user is vip should discount five percent if total >= 1000
   Failure/Error: order = Order.new( :total => 2000 )
   ArgumentError:
     wrong number of arguments (1 for 0)
   # ./order_spec.rb:15:in `initialize'
   # ./order_spec.rb:15:in `new'
   # ./order_spec.rb:15

2) Order#amount when user is vip should discount ten percent if total >= 10000
   Failure/Error: order = Order.new( :total => 10000 )
   ArgumentError:
     wrong number of arguments (1 for 0)
   # ./order_spec.rb:20:in `initialize'
   # ./order_spec.rb:20:in `new'
   # ./order_spec.rb:20
```

(狀態顯示為在寫 Order 主程式)

輸出結果(green)



```
1. Shell
New Tab Info : Customize Close : Execute
[~/ruby] rspec order_spec.rb -fs

Order
  #amount
    when user is vip
      should discount five percent if total >= 1000
      should discount ten percent if total >= 10000
    when user is not vip
      should discount three percent if total > 10000

Finished in 0.00102 seconds
3 examples, 0 failures
[~/ruby] $
```

漂亮的程式文件


before 和 after

- 如同 xUnit 框架的 setup 和 teardown
- before(:each) 每段 it 之前執行
- before(:all) 整段 describe 執行一次
- after(:each)
- afte(:all)

```
describe Order do
```

```
  describe "#amount" do
```

```
    context "when user is vip" do
```

```
       before(:each) do
        @user = User.new( :is_vip => true )
        @order = Order.new( :user => @user )
      end
```

```
      it "should discount five percent if total >= 1000" do
        @order.total = 2000
        @order.amount.should == 1900
      end
```

```
      it "should discount ten percent if total >= 10000" do
        @order.total = 10000
        @order.amount.should == 9000
      end
```

```
    end
```

```
    context "when user is vip" { ... }
  end
```

```
end
```

pending

可以先列出來打算要寫的測試

```
describe Order do  
  describe "#paid?" do  
    → it "should be false if status is new"  
    → xit "should be true if status is paid or shipping" do  
      end  
    end  
  end  
end
```

```
[~/ruby] $ rspec order_spec.rb -fs
```

Order

#amount

when user is vip

should discount five percent if total >= 1000

should discount ten percent if total >= 10000

when user is not vip

should discount three percent if total > 10000

#paid?

should be false if status is new (PENDING: Not Yet Implemented)

should be true if status is paid or shipping (PENDING: No reason given)

Pending:

Order#paid? should be false if status is new

Not Yet Implemented

./order_spec.rb:39

Order#paid? should be true if status is paid or shipping

No reason given

./order_spec.rb:41

Finished in 0.00137 seconds

5 examples, 0 failures, 2 pending

```
[~/ruby] $
```

let(:name) { exp }

- **Lazy**，有需要才會運算，並且是 **Memoized**。
- 相較於 **before(:each)** 增加執行速度
- 不需用 **instance variable** 放 **before**
- 增加可讀性
- **let!** 則是非 **lazy** 版本


```
describe Order do
```

```
  describe "#amount" do
```

```
    context "when user is vip" do
```

```
      ➡ let(:user) { User.new( :is_vip => true ) }
      ➡ let(:order) { Order.new( :user => user ) }
```

```
      it "should discount five percent if total >= 1000" do
        order.total = 2000
        order.amount.should == 1900
      end
```

```
      it "should discount ten percent if total >= 10000" do
        order.total = 10000
        order.amount.should == 9000
      end
```

```
    end
```

```
    context "when user is vip" { ... }
```

```
  end
```

```
end
```

一些別名方法

哪個念起來順就用哪個

describe Order **do** # describe 和 context 其實是 alias

it, specify, example 其實都一樣

it { ... }

specify { ... }

example { ... }

end

一些慣例

- 一個 rb 檔案配一個同名的 `_spec.rb` 檔案
 - 容易找
 - `guard` 等工具容易設定
 - `editor` 有支援快速鍵
- `describe “#title”` 是 instance method
- `describe “.title”` 是 class method

輸出格式

- `rspec filename.rb` 預設不產生文件
- `rspec filename.rb -fd`輸出 specdoc 文件
- `rspec filename.rb -fh` 輸出 html 文件

HTML format

鼓勵寫出高 spec coverage 程式，因為可以當做一種文件

The screenshot shows a web browser window titled "RSpec results" displaying an HTML report for "file:///Users/ihower/ruby/report.html". The browser's address bar and search bar are visible. Below the browser window, the RSpec report is displayed with a red header bar. The header bar contains the text "RSpec Code Examples" on the left, a summary "11 examples, 1 failure, 1 pending" and "Finished in 0.023062 seconds" on the right, and a status bar with three checkboxes: "Passed", "Failed", and "Pending". The report is organized into sections, each with a green header bar: "Order", "status", "#amount", "when user is vip", "when user is not vip", "#paid?", "#receiver_name", and "#ship!". Each section contains a list of examples, each with a green bar indicating it passed. The "with paid" section is highlighted with a red bar and contains a code snippet showing a mock expectation for the "gateway API" method.

RSpec results

file:///Users/ihower/ruby/report.html

Google

iGoogle Read Today Read Later Instapaper Reader Blog Twitter Facebook Plurk GitHub Tumblr Yahoo! WebTV Redmine(O) Redmine(T) 翻譯(G) 字典(Y)

RSpec Code Examples 11 examples, 1 failure, 1 pending
Finished in 0.023062 seconds

✓ Passed ✓ Failed ✓ Pending

Order

- should be valid

status

- should == "New"

#amount

- when user is vip**
 - should discount five percent if total >= 1000
 - should discount ten percent if total >= 10000
- when user is not vip**
 - should discount three percent if total > 10000

#paid?

- should be false if status is new
- should be true if status is paid or shipping

#receiver_name

- should be user name

#ship!

with paid

- should call gateway API

```
(Mock "ezcat").deliver([RSpec::Mocks::Mock:0x809e3830 @name="user"])  
  expected: 1 time  
  received: 0 times  
  
./order_spec.rb:84  
  
82 context "with paid" do  
83   it "should call gateway API" do
```

Continuous Testing

- 使用 `guard-rspec`
- 程式一修改完存檔，自動跑對應的測試
- 節省時間，立即回饋

Code Kata

- 一種練習方法，透過小型程式題目進行鍛鍊，就像學功夫套拳，重複練功一樣
- 除了自己練，也可以 pair-programming
- TDD 需要練習才能抓到訣竅，與其說它是一種測試方式，不如說它更像一種設計手法
- 抓到 TDD 測試精神和訣竅，比學再多漂亮的語法糖更重要

準備 Kata 環境

<https://github.com/ihower/ruby-kata>

- Gemfile 設定 rspec, guard
- 安裝 growl notification (optional)
- bundle

TDD 訣竅一：

Red-Green-Refactor
development cycle

FizzBuzz

- 從 1 數到 100
- 逢三整除，輸出 Fizz
- 逢五整除，輸出 Buzz
- 逢三五整除，輸出 FizzBuzz

Leap Years

- 判斷閏年
 - 西元年份除以400可整除，為閏年。
 - 西元年份除以4可整除但除以100不可整除，為閏年。
 - 其他則是平年

Roman Numerals

- 轉換羅馬數字
 - 1 -> I
 - 4 -> IV
 - 5 -> V
 - 9 -> IX
 - 10 -> X
 - 20 -> XX

What have we learned?

- 一個 it 裡面只有一種測試目的，最好就只有一個 expectation
- 要先從測試 failed 失敗案例開始
 - 確保每個測試都有效益，不會發生砍掉實作卻沒有造成任何測試失敗
- 一開始的實作不一定要先直攻一般解，可以一步一步在 cycle 中進行思考和重構
- 測試程式碼的可讀性比 DRY 更重要
- 安全重構：無論是改實作或是改測試碼，當時的狀態應該要維持 Green

TDD 訣竅二：

讓測試引導 API 設計

Bowling Game

- 計算保齡球分數
 - X (strike) 累加後兩球分數
 - / (spare) 累加後一球分數

BowlingGame

```
# roll
# roll_many(1,2,3,4)
# or roll_many("1234512345")
# finished?
# score
```

What have we learned?

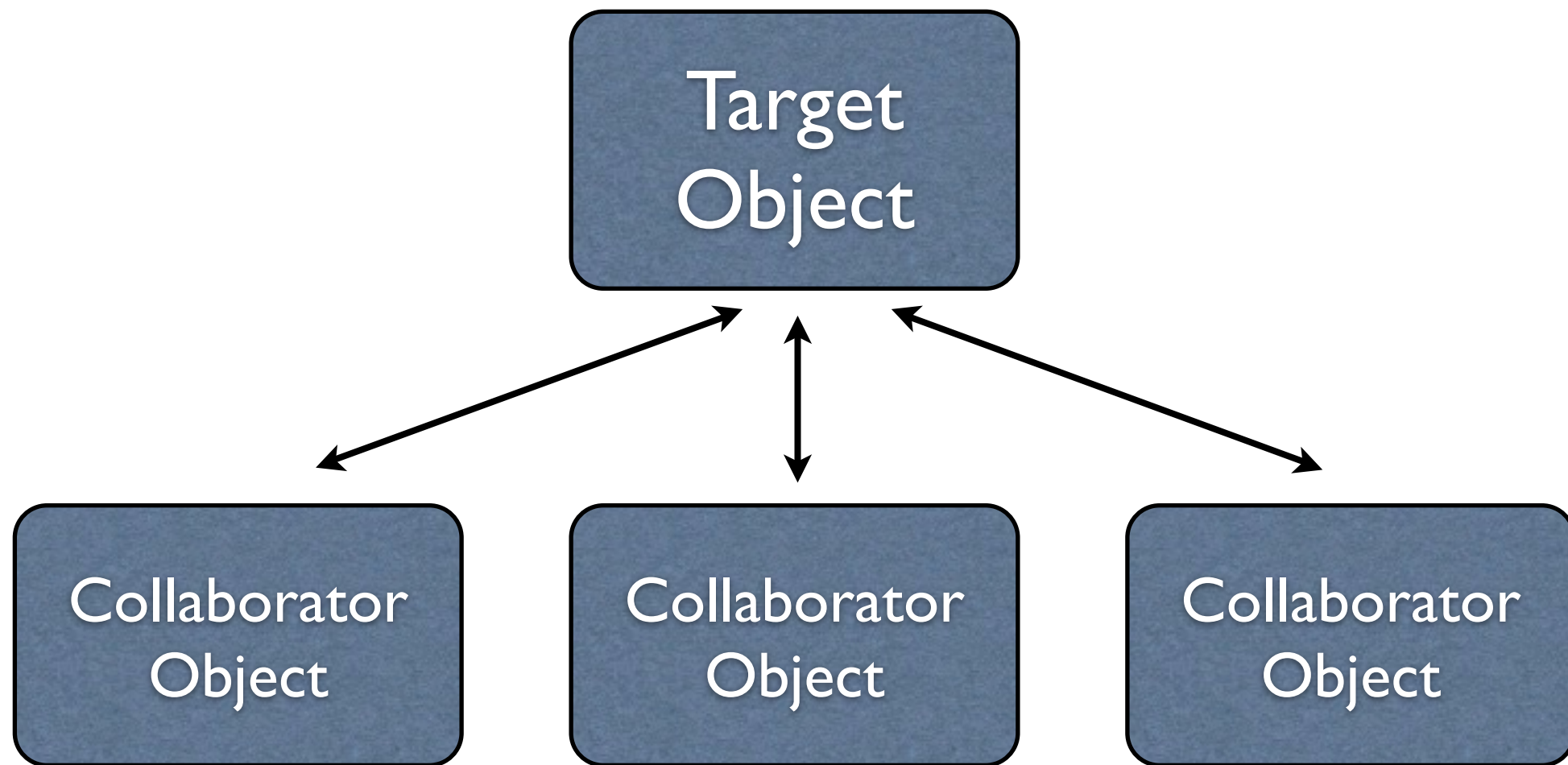
- 透過寫測試，定義出哪些是類別公開的介面(API)
- 測試碼方便呼叫的API，就是好API
- 不需要公開的，定義成 `private` 方法，讓實作有更好的物件導向封裝。
- 不需要針對 `private` 方法直接寫單元測試，而是透過 `public` 方法間接測試

RSpec Mocks

用假的物件替換真正的物件，作為測試之用

物件導向是

物件和物件之間的互動



當你在寫目標類別的測試和實作時，這些 Collaborator...

- 無法控制回傳值的外部系統 (例如第三方 web service)
- 建構正確的回傳值很麻煩 (例如得準備很多假資料)
- 可能很慢，拖慢測試速度 (例如耗時的運算)
- 有難以預測的回傳值 (例如亂數方法)
- 還沒開始實作 (特別是採用 TDD 流程)

使用假物件

- 可以隔離 Collaborator 的 Dependencies
- 讓你專心在目標類別上
- 只要 Collaborator 提供的介面不變，修改實作不會影響這邊的測試。

Stub

回傳設定好的回傳值

→ `@user = double("user", :name => "ihower")`
`@user.name # "ihower"`

→ `@customer = double("customer").as_null_object`
`@customer.foobar # nil`

用來測試最後的狀態

```
describe "#receiver_name" do
  it "should be user name" do
    user = double(:user, :name => "ihower")

    order = Order.new(:user => user)
    → expect(order.receiver_name).to eq("ihower")
  end
end
```

Mock

如果沒被呼叫到，就算測試失敗

```
@gateway = double("ezcat")
```

```
# Message Expectations
```

```
expect(@gateway).to receive(:deliver).with(@user).and_return(true)
```

```
@gateway.deliver
```



用來測試行為的發生

```
describe "#ship!" do
```

```
  before do
```

```
    @user = double("user").as_null_object
```

```
    @gateway = double("ezcat")
```

```
    @order = Order.new( :user => @user, :gateway => @gateway )
```

```
  end
```

```
  context "with paid" do
```

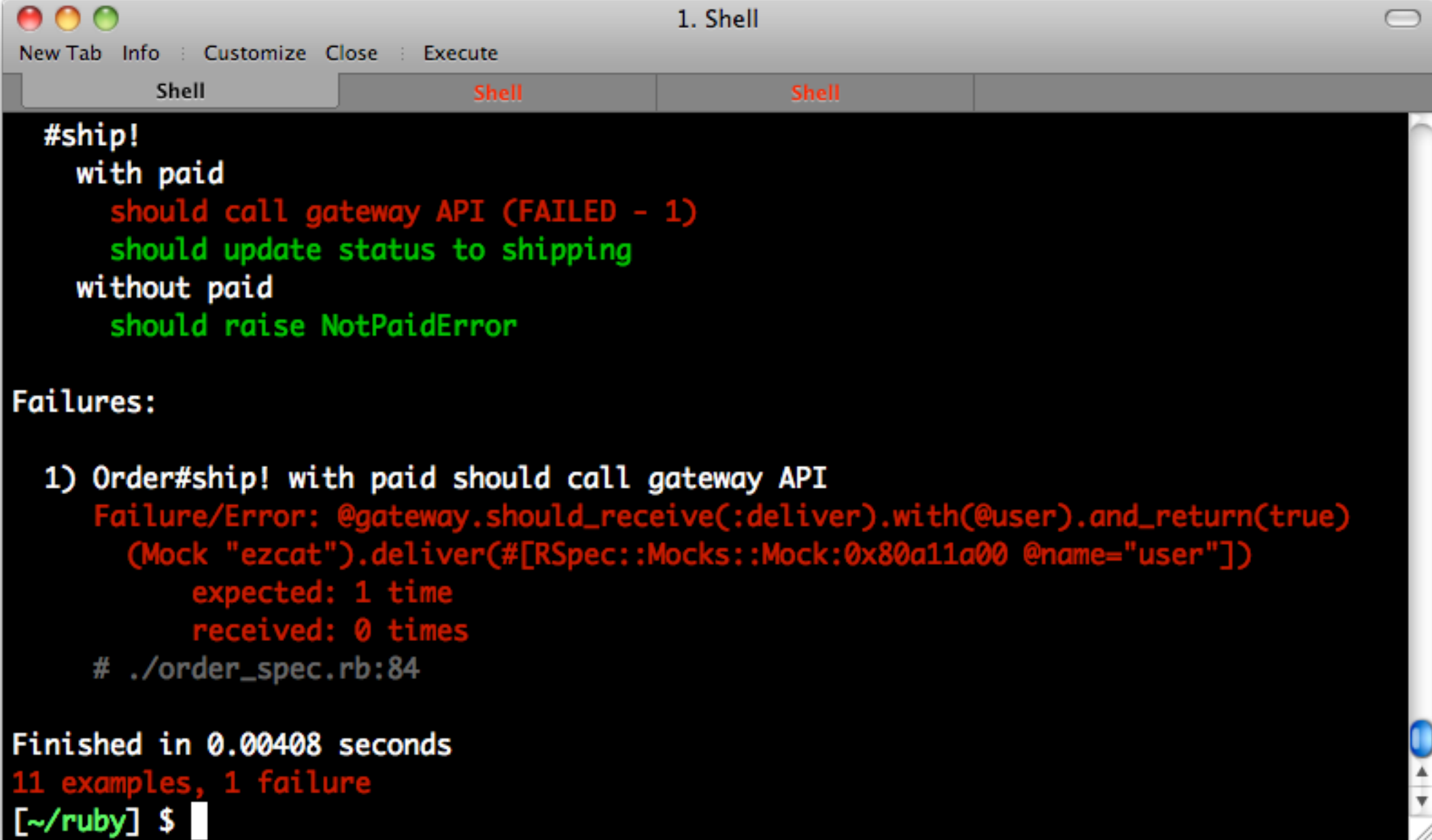
```
    it "should call ezship API" do
```

➔ `expect(@gateway).to receive(:deliver).with(@user).and_return(true)`
`@order.ship!`

```
  end
```

```
end
```


行為沒發生...



```
1. Shell
New Tab Info : Customize Close : Execute
Shell Shell Shell
#ship!
  with paid
    should call gateway API (FAILED - 1)
    should update status to shipping
  without paid
    should raise NotPaidError

Failures:

1) Order#ship! with paid should call gateway API
   Failure/Error: @gateway.should_receive(:deliver).with(@user).and_return(true)
     (Mock "ezcat").deliver(#[RSpec::Mocks::Mock:0x80a11a00 @name="user"])
       expected: 1 time
       received: 0 times
   # ./order_spec.rb:84

Finished in 0.00408 seconds
11 examples, 1 failure
[~/ruby] $
```

Partial mocking and stubbing

- 如果 Collaborator 類別已經有了，我們可以 reuse 它
- 只有在有需要的時候 stub 或 mock 特定方法

可以用在任意物件及類別上 (Partial Stubbing and Mocking)

➔ `user = User.new`
`allow(user).to receive(:find).and_return("ihower")`

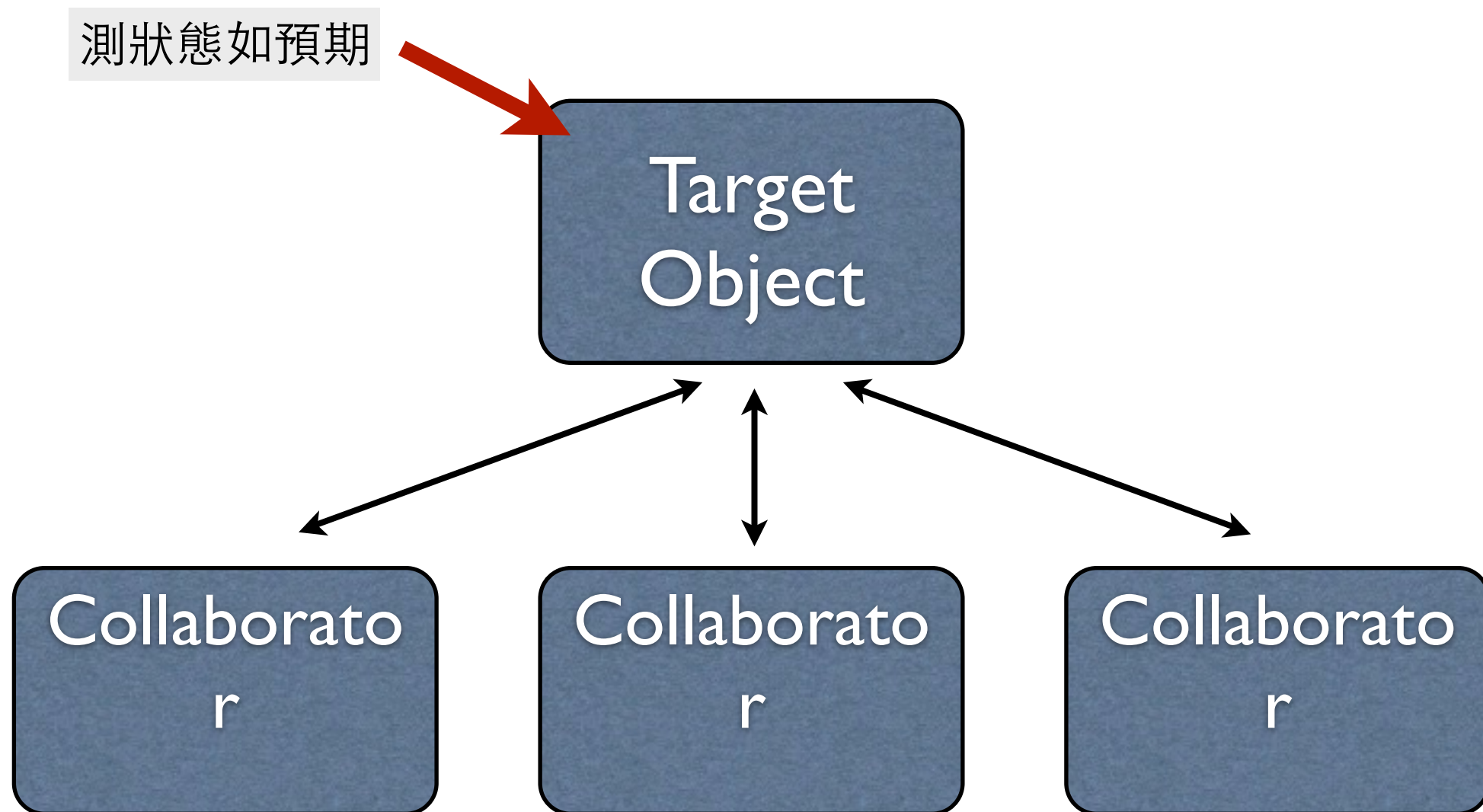
➔ `gateway = Gateway.new`
`expect(gateway).to receive(:deliver).with(user).and_return(true)`

一般測試流程 vs. Mock 測試流程

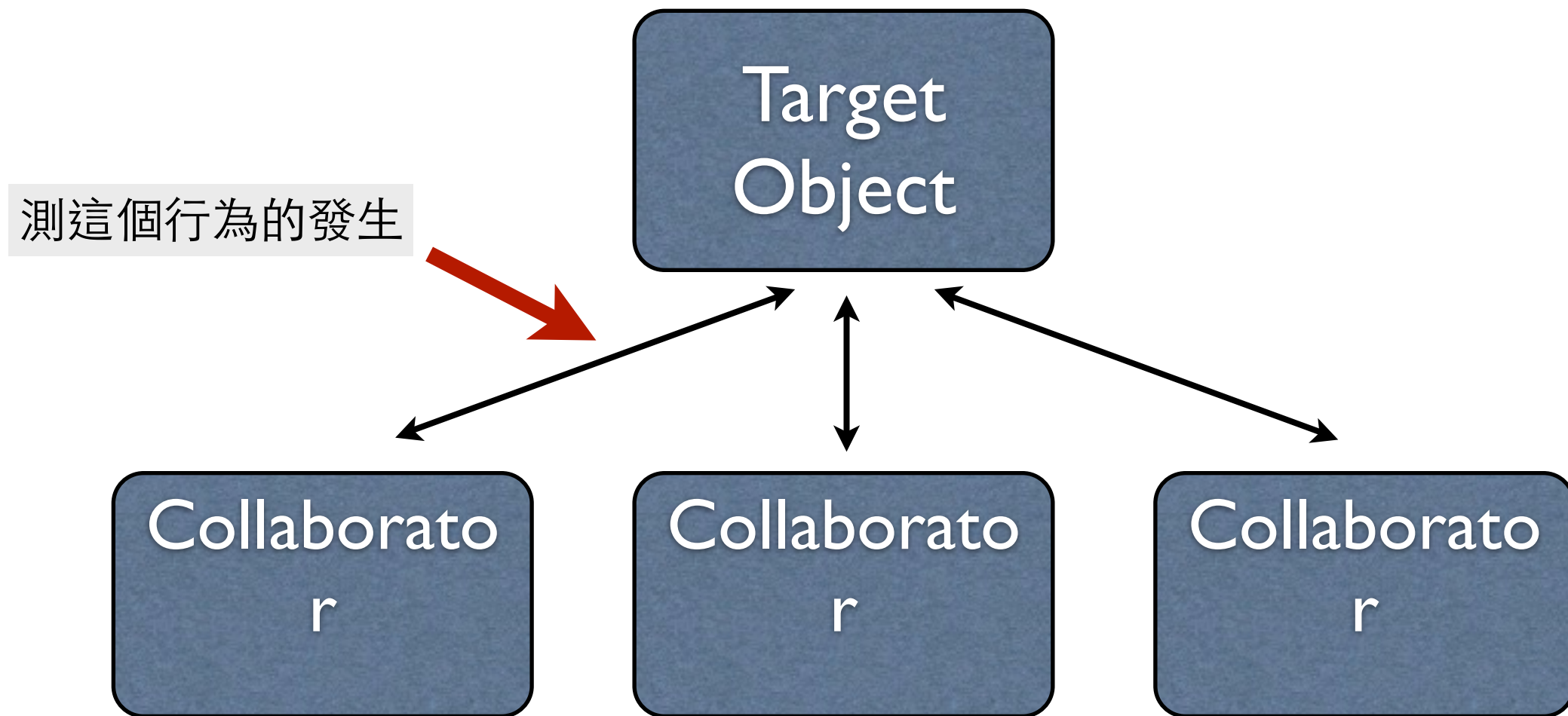
Given 給定條件
When 當事情發生
Then 則結果要是如何

Given 給定條件
Expect 預期會發生什麼
When 當事情發生

一般測試是檢查物件最後的狀態

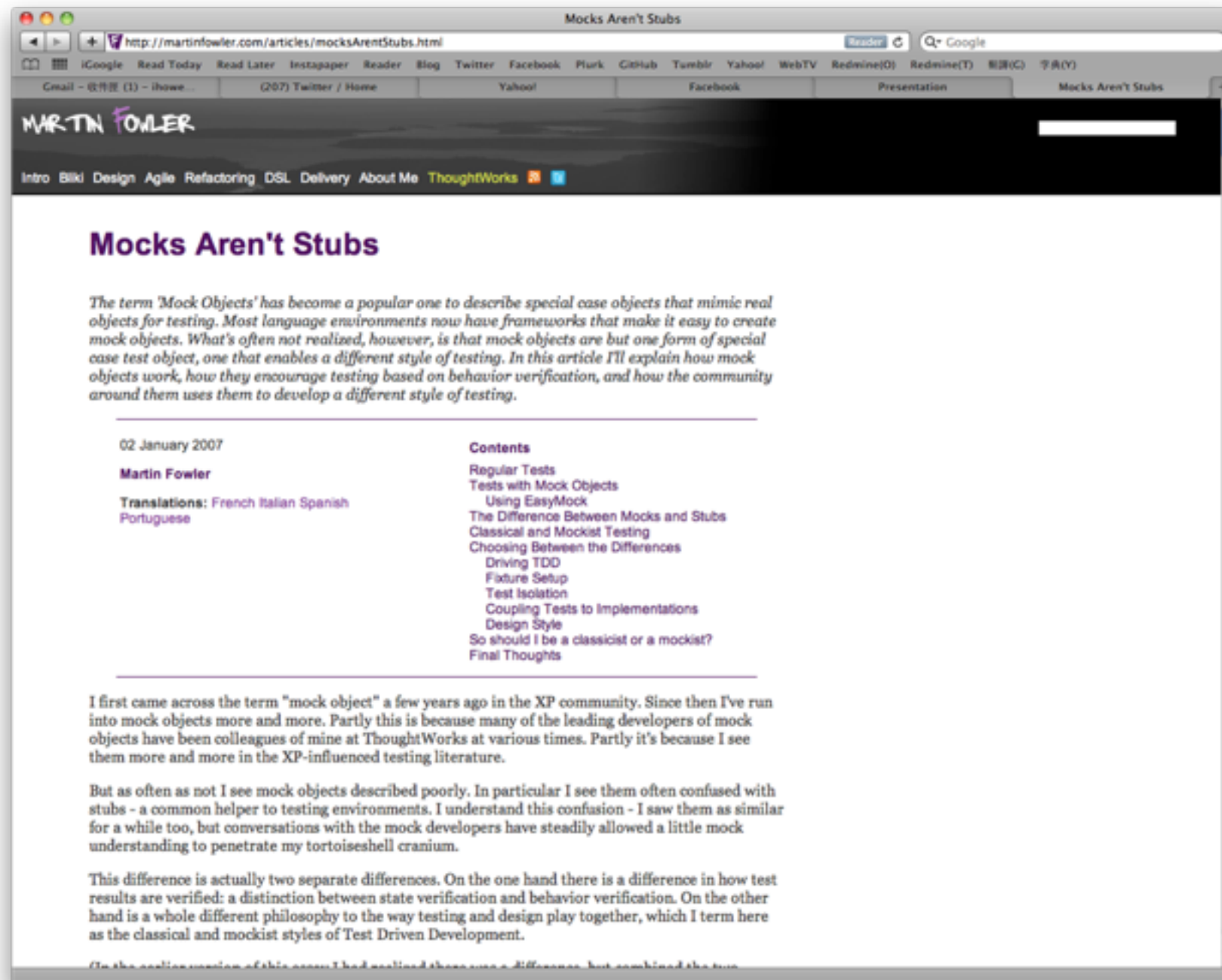


Mocks 可以讓我們測試物件之間的行為



Mocks Aren't Stubs

(2004/Java)



<http://martinfowler.com/articles/mocksArentStubs.html>

Classical Testing v.s. Mockist Testing

- 你先決定你要完成的範圍，然後實作這個範圍的所有物件。
- 只針對一個類別做測試及實作，其他都用假的。

哀號聲浪也不少...

- 你可能需要 stub 很多東西
- stub 的回傳值可能難以建構
- 與 Collaborator 的行為太耦合，改介面就要跟著改一堆測試。
- 承上，即使忘了跟著改實作，單元測試還是會過... 例如在動態語言中，可能 Collaborator 後來改了方法名稱，但是這邊的測試仍然會過...

如何趨吉避凶？

- 採用 TDD，會讓最後設計出來的 API 趨向：
 - 不需要 stub 太多
 - 回傳值簡單
 - 類別之間的耦合降低
- 擁有更高層級的整合測試，使用真的物件來做測試。

我的推論

採用 TDD 並搭配
Integration Test



雖然有些 stub 和 mock



但是會設計出比較好的
API 讓副作用較少

我的推論

濫用 stub 和 mock



沒有用 TDD，也沒有 Integration Test



API 設計的不好，stub 的缺點就顯現出來的

不然，請小心：

- 用 Mocks 來完全隔離內部 Internal dependencies
- 用 Mocks 來取代還沒實作的物件

版本一：

這個 Order 完全隔離 Item

```
describe Order do
  before do
    @order = Order.new
  end
```

```
  describe "#<<" do
    it "should push item into order.items" do
      Item = double("Item")
      item = double("item", :name => "foobar")
      → allow(Item).to receive(:new).and_return(item)
```

```
      @order << "foobar"
      → expect(@order.items.last.name).to eq "foobar"
    end
  end
```

```
end
```

測試通過，即使還沒 實作 Item

```
class Order

  attr_accessor :items

  def initialize
    self.items = []
  end

  def <<(item_name)
    self.items << Item.new(item_name)
  end

end
```

版本二：

實作真的 Item 物件

(拿掉剛才的所有 stub code 即可)

```
describe Order do
  before do
    @order = Order.new
  end

  describe "#<<" do
    it "should push item into order.items" do
      @order << "foobar"
      expect(@order.items.last.name).to eq "foobar"
    end
  end
end

end
```


需要寫真的 Item 讓測試通過

```
class Item
  attr_accessor :name

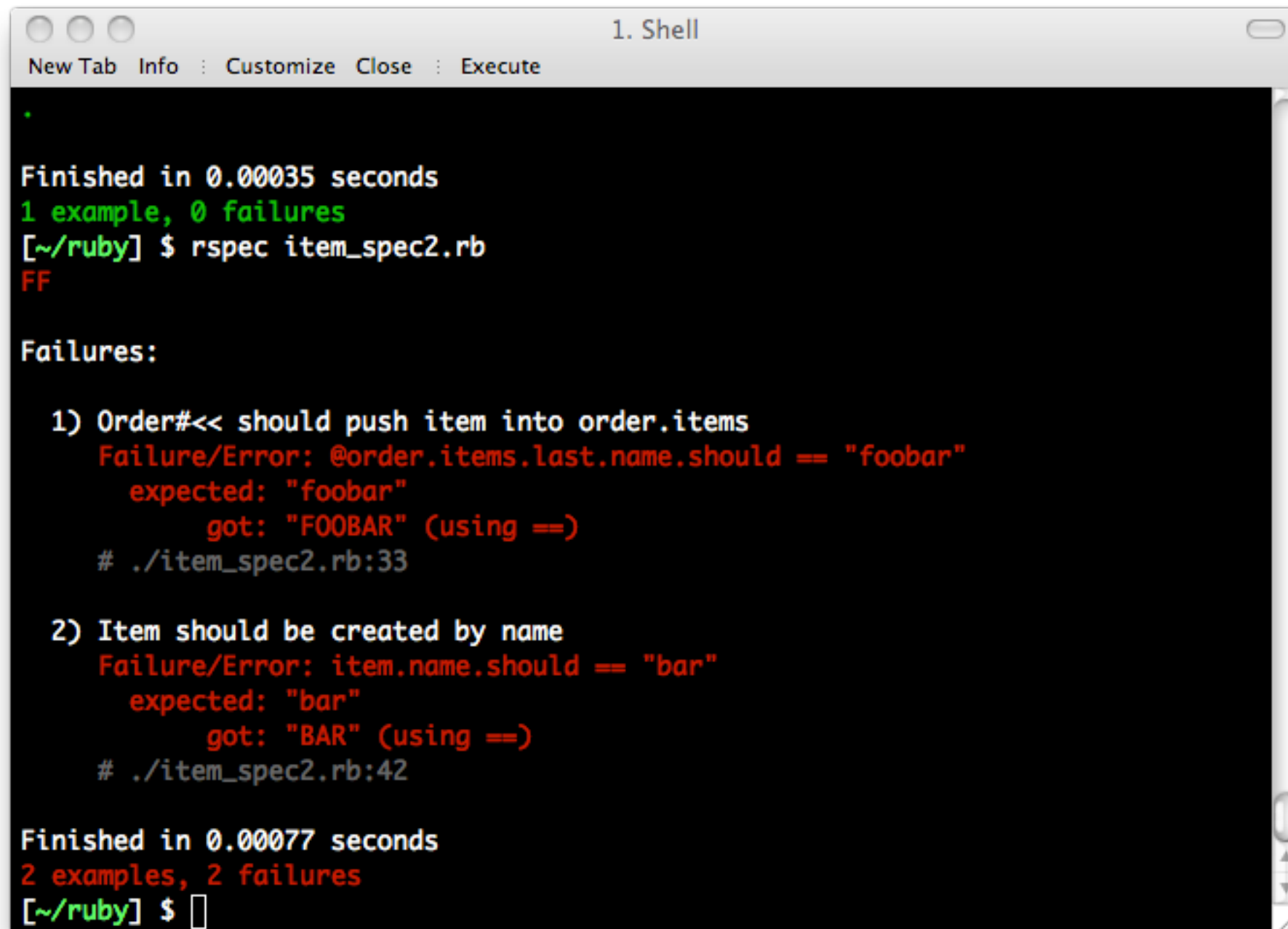
  def initialize(name)
    self.name = name
  end
end

describe Item do
  it "should be created by name" do
    item = Item.new("bar")
    expect(item.name).to eq "bar"
  end
end
```

差在哪？假設我們後來 修改了 Item 的實作

```
class Item  
  
  attr_accessor :name  
  
  def initialize(name)  
    → self.name = name.upcase  
  end  
  
end
```

真的版本會有多個 Failure :/
Mocks 版本則不受影響 :)



```
1. Shell
New Tab Info : Customize Close : Execute

Finished in 0.00035 seconds
1 example, 0 failures
[~/ruby] $ rspec item_spec2.rb
FF

Failures:

  1) Order#<< should push item into order.items
     Failure/Error: @order.items.last.name.should == "foobar"
     expected: "foobar"
     got: "FOOBAR" (using ==)
     # ./item_spec2.rb:33

  2) Item should be created by name
     Failure/Error: item.name.should == "bar"
     expected: "bar"
     got: "BAR" (using ==)
     # ./item_spec2.rb:42

Finished in 0.00077 seconds
2 examples, 2 failures
[~/ruby] $
```

但是如果 Item 改介面呢?

```
class Item
```

→

```
attr_accessor :title
```

```
  def initialize(name)
    self.title = name.upcase
  end
```

```
end
```

即使我們忘了改 Order，
Mocks 版的測試還是照過
不誤：(

兩害相權取其輕

- 寧願看到重複原因的 Failure?
- 還是連 Failure 都捕捉不到，失去測試的意義?

So, 有限度使用

- 造真物件不難的時候，造假的時間拿去造真的。
有需要再用 Partial Stubbing/Mocking。
(相較於 Java，在 Ruby 裡造物件很快)
- 只在 Collaborator 不好控制的時候，Stub 它
- 只在 Collaborator 還沒實作的時候，Mock 它
- Collaborator 的狀態不好檢查或它的實作可能會改變，這時候才改成測試行為。

造假舉例：Logger

已知有一個 Logger 其 log 方法運作正常

測狀態

logger.log("Report generated")
expect(File.read("log.log")).to eq "Report generated"

如果 Logger 換別種實作就死了

測行為

expect(logger).to receive(:log).with(/Report generated/)
logger.log("Report generated")

造假舉例： MVC 中的Controller 測試

測狀態

it "should be created successful" do

post :create, :name => "ihower"

expect(response).to be_success

expect(User.last.name).to eq("ihower") # 會真的存進資料庫

end

測行為

it "should be created successful" do

expect(Order).to receive(:create).with(:name => "ihower").and_return(true)

post :create, :name => "ihower"

expect(response).to be_success

end

Mocks 小結

- 多一種測試的方法：除了已經會的測試狀態，多了可以測試行為的方法
- 有需要用到來讓測試更好寫，就用。不必要用，就不要用。

Code Kata

- Train Reservation HTTP Client library
 - GET /train/{:id} 拿列車座位資料
 - POST /train/{:id}/reservation 定位
- Ticket Office Web Service (Part 2)

What have we learned?

- 利用 Mocks 處理測試邊界(第三方服務)
- 設計 Web Service API (Part 2 會繼續沿用)

Reference:

- The RSpec Book
- The Rails 3 Way
- Foundation Rails 2
- xUnit Test Patterns
- everyday Rails Testing with RSpec
- <http://pure-rspec-rubynation.heroku.com/>
- <http://jamesmead.org/talks/2007-07-09-introduction-to-mock-objects-in-ruby-at-lrug/>
- <http://martinfowler.com/articles/mocksArentStubs.html>
- <http://blog.rubybestpractices.com/posts/gregory/034-issue-5-testing-antipatterns.html>
- <http://blog.carbonfive.com/2011/02/11/better-mocking-in-ruby/>

Thanks.