Essential TDD

Bala Paranj

Section 1 : Basics

This section is about the basics of TDD. It introduces the concepts using code exercises.

Fibonacci

Objectives

- To learn TDD Cycle : Red, Green, Refactor.
- Focus on getting it to work first, cleanup by refactoring and then focus on optimization.
- When refactoring, start green and end in green.
- Learn recursive solution and optimize the execution by using non-recursive solution.
- Using existing tests as regression tests when making major changes to existing code.

Problem Statement

In mathematics, the Fibonacci numbers are the numbers in the following integer sequence: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144...

F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9	F_{10}	F_{11}	F_{12}	F_{13}	F_{14}	F_{15}
0	1	1	2	3	5	8	13	21	34	55	89	144	233	377	610

Figure 1: Fibonacci Numbers

Solution

By definition, the first two numbers in the Fibonacci sequence are 0 and 1, and each subsequent number is the sum of the previous two.

Algebraic Equation

In mathematical terms, the sequence fibonacci(n) of Fibonacci numbers is defined by the recurrence relation fibonacci(n) = fibonacci(n-1) + fibonacci(n-2) with seed values fibonacci(0) = 0, fibonacci(1) = 1

Visual Representation

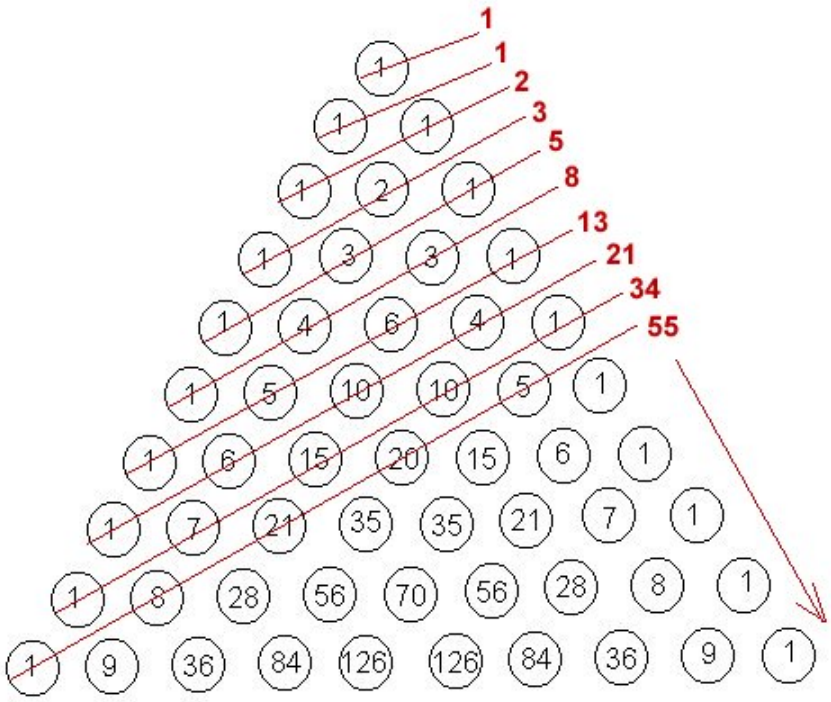

Figure 2: Fibonacci Numbers

Guidelines

1. Each row in the table is an example. Make each example executable.

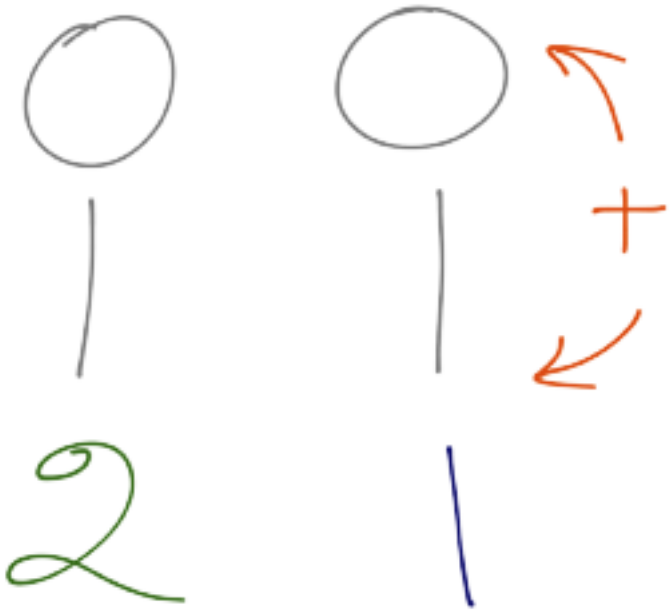

Figure 3: Calculating Fibonacci Numbers

Input	Output
0	0
1	1
2	1
3	2
4	3
5	5

2. The final solution should be able to take any random number and calculate the Fibonacci number without any modification to the production code.

Set Up Environment

Version 1

```
require 'test/unit'

class FibonacciTest < Test::Unit::TestCase
  def test_fibonacci_of_zero_is_zero
    fail "fail"
  end
end
```

Got proper require to execute the test. Proper naming of test following naming convention.

This example illustrates how to convert Requirements –> Examples –> Executable Specs. Each test for this problem takes an argument, does some computation and returns a result. It illustrates Direct Input and Direct Output. There are no side effects. Side effect free functions are easy to test.

Discovery of Public API

Version 2

finonacci_test.rb

```
require 'test/unit'

class Fibonacci
```

```
  def self.of(number)
    0
  end
end

class FibonacciTest < Test::Unit::TestCase
  def test_fibonacci_of_zero_is_zero
    fib_of_zero = Fibonacci.of(0)
    assert_equal(fib_of_zero, 0)
  end
end
```

Don't Change the Test code and Code Under Test at the Same Time

Version 3

```
require 'test/unit'

class Fibonacci
  def self.of(number)
    0
  end
end

class FibonacciTest < Test::Unit::TestCase
  def test_fibonacci_of_zero_is_zero
    fib_of_zero = Fibonacci.of(0)
    assert_equal(0, fib_of_zero)
  end
end
```

Found the right assertion to use.Overcame the temptation to change the test code and code under test at the same time.Thereby test driving the development of the production code.Got the test to pass quickly by using a fake implementation.The implementation returns a constant.

Dirty Implementation

Version 4

Made fib(1) = 1 pass very quickly using a dirty implementation.

```
require 'test/unit'

class Fibonacci
  def self.of(number)
    number
  end
end

class FibonacciTest < Test::Unit::TestCase
  def test_fibonacci_of_zero_is_zero
    fib_of_zero = Fibonacci.of(0)
    assert_equal(0, fib_of_zero)
  end

  def test_fibonacci_of_one_is_one
    fib_of_one = Fibonacci.of(1)
    assert_equal(1, fib_of_one)
  end
end
```

Forcing the Implementation to Change via Tests

Version 5

Broken test forced the implementation to change. Dirty implementation passes the test.

```
require 'test/unit'

class Fibonacci
  def self.of(number)
    if number == 2
      return 1
    else
      return number
    end
  end
end

class FibonacciTest < Test::Unit::TestCase
  def test_fibonacci_of_zero_is_zero
    fib_of_zero = Fibonacci.of(0)
    assert_equal(0, fib_of_zero)
  end
```

```
  def test_fibonacci_of_one_is_one
    fib_of_one = Fibonacci.of(1)
    assert_equal(1, fib_of_one)
  end

  def test_fibonacci_of_two_is_one
    fib_of_two = Fibonacci.of(2)
    assert_equal(1, fib_of_two)
  end
end
```

Refactoring in the Green State

Version 6

The new test broke the implementation. Commented out the new test to refactor the test in green state. This code is ready to be generalized.

```
require 'test/unit'

class Fibonacci
  def self.of(number)
    if number == 0
      return 0
    elsif number <= 2
      return 1
    end
  end
end

class FibonacciTest < Test::Unit::TestCase
  def test_fibonacci_of_zero_is_zero
    fib_of_zero = Fibonacci.of(0)
    assert_equal(0, fib_of_zero)
  end

  def test_fibonacci_of_one_is_one
    fib_of_one = Fibonacci.of(1)
    assert_equal(1, fib_of_one)
  end

  def test_fibonacci_of_two_is_one
    fib_of_two = Fibonacci.of(2)
    assert_equal(1, fib_of_two)
  end
```

```
  def xtest_fibonacci_of_three_is_two
    fib_of_three = Fibonacci.of(3)
    assert_equal(2, fib_of_three)
  end
end
```

Generalized Solution

Version 7

Input	Output
0	0
1	1
2	1
3	2

So the pattern emerges and we see the result is the sum of previous to fibonacci numbers return 2 is actually return 1 + 1 which from the above table is fib(n-1) + fib(n-2), so the solution is fib(n-1) + fib(n-2)

```
require 'test/unit'

class Fibonacci
  def self.of(number)
    if number == 0
      return 0
    elsif number <= 2
      return 1
    end
    return 2
  end
end

class FibonacciTest < Test::Unit::TestCase
  def test_fibonacci_of_zero_is_zero
    fib_of_zero = Fibonacci.of(0)
    assert_equal(0, fib_of_zero)
  end

  def test_fibonacci_of_one_is_one
```

```
    fib_of_one = Fibonacci.of(1)
    assert_equal(1, fib_of_one)
  end

  def test_fibonacci_of_two_is_one
    fib_of_two = Fibonacci.of(2)
    assert_equal(1, fib_of_two)
  end

  def test_fibonacci_of_three_is_two
    fib_of_three = Fibonacci.of(3)
    assert_equal(2, fib_of_three)
  end
end
```

Recursive Solution

Version 8

Input	Output
0	0
1	1
2	1
3	2

```
require 'test/unit'

class Fibonacci
  def self.of(number)
    if number == 0
      return 0
    elsif number <= 2
      return 1
    end
    return of(number - 1) + of(number - 2)
  end
end

class FibonacciTest < Test::Unit::TestCase
  def test_fibonacci_of_zero_is_zero
    fib_of_zero = Fibonacci.of(0)
```

```
    assert_equal(0, fib_of_zero)
  end

  def test_fibonacci_of_one_is_one
    fib_of_one = Fibonacci.of(1)
    assert_equal(1, fib_of_one)
  end

  def test_fibonacci_of_two_is_one
    fib_of_two = Fibonacci.of(2)
    assert_equal(1, fib_of_two)
  end

  def test_fibonacci_of_three_is_two
    fib_of_three = Fibonacci.of(3)
    assert_equal(2, fib_of_three)
  end
end
```

The generalized solution uses recursion.

Cleanup

Version 9

Recursive solution:

Input	Output
0	0
1	1
2	1
3	2

```
require 'test/unit'

class Fibonacci
  def self.of(n)
    return 0 if n == 0
    return 1 if n == 1
    return of(n - 1) + of(n - 2)
  end
end
```

```ruby
class FibonacciTest < Test::Unit::TestCase
  def test_fibonacci_of_zero_is_zero
    fib_of_zero = Fibonacci.of(0)
    assert_equal(0, fib_of_zero)
  end

  def test_fibonacci_of_one_is_one
    fib_of_one = Fibonacci.of(1)
    assert_equal(1, fib_of_one)
  end

  def test_fibonacci_of_two_is_one
    fib_of_two = Fibonacci.of(2)
    assert_equal(1, fib_of_two)
  end

  def test_fibonacci_of_three_is_two
    fib_of_three = Fibonacci.of(3)
    assert_equal(2, fib_of_three)
  end

  def test_fibonacci_of_ten_is_what
    fib_of_ten = Fibonacci.of(10)
    assert_equal(55, fib_of_ten)
  end
end
```

Green before and after refactoring. Used idiomatic Ruby to cleanup code. Named variables expressive of the domain.

Optimization

Version 10

Non-Recursive solution:

Input	Output
0	0
1	1
2	1
3	2

```
require 'test/unit'

class Fibonacci
  def self.of(n)
    current, successor = 0,1
    n.times do
      current, successor = successor, current + successor
    end
    return current
  end
end

class FibonacciTest < Test::Unit::TestCase
  def test_fibonacci_of_zero_is_zero
    fib_of_zero = Fibonacci.of(0)
    assert_equal(0, fib_of_zero)
  end

  def test_fibonacci_of_one_is_one
    fib_of_one = Fibonacci.of(1)
    assert_equal(1, fib_of_one)
  end

  def test_fibonacci_of_two_is_one
    fib_of_two = Fibonacci.of(2)
    assert_equal(1, fib_of_two)
  end

  def test_fibonacci_of_three_is_two
    fib_of_three = Fibonacci.of(3)
    assert_equal(2, fib_of_three)
  end

  def test_fibonacci_of_ten_is_what
    fib_of_ten = Fibonacci.of(10)
    assert_equal(55, fib_of_ten)
  end

end
```

This version illustrates using existing tests as safety net when making major changes to the code. Notice that we only focus only on one thing at a time, the focus can shift from one version to the other.

Exercises:

1. Run the mini-test based fibonacci and make sure all tests pass. ($ ruby fibonacci_test.rb)
2. Move the fibonacci class into its own file and make all the tests pass.
3. Convert the given mini-test test to rspec version fibonacci_spec.rb.
4. Get the output of the mini-test in color.
5. Watch the Factorial screencast and convert the unit test to rspec spec.

Guess Game

Objectives

- How to test random behavior ?
- Illustrate inverting dependencies.
- How to make your code depend on abstractions instead of concrete implementation ?
- Illustrate Single Responsibility Principle. No And, Or, or But.
- Illustrate programming to an interface not to an implementation.
- When to use partial stub on a real object ? Illustrated by spec 7 and 8.
- Random test failures due to partial stub. Fixed by isolating the random number generation.
- Small methods focused on doing one thing.
- How to defer decisions by using Mocks ?
- Using mock that complies with Gerard Meszaros standard.
- How to use as_null_object ?
- How to write contract specs to keep mocks in sync with code ?

Guessing Game Description

Write a program that generates a random number between 0 and 100 (inclusive). The user must guess this number. Each correct guess (if it was a number) will receive the response "Guess Higher!" or "Guess Lower!". Once the user has successfully guessed the number, you will print various statistics about their performance as detailed below:

- The prompt should display : "Welcome to the Guessing Game"
- When the program is run it should generate a random number between 0 and 100 inclusive
- You will display a command line prompt for the user to enter the number representing their guess. Quitting is not an option. The user can only end the game by guessing the target number. Be sure that your prompt explains to them what they are to do.
- Once you have received a value from the user, you should perform validation. If the user has given you an invalid value (anything other than a number between 1 and 100), display an appropriate error message. If the user has given you a valid value, display a message either telling them that there were correct or should guess higher or lower as described above. This process should continue until they guess the correct number.
- Once the user has guessed the target number correctly, you should display a "report" to them on their performance. This report should provide the following information:
  - The target number
  - The number of guesses it took the user to guess the target number
  - A list of all the valid values guessed by the user in the order in which they were guessed.
  - A calculated value called "Cumulative error". Cumulative error is defined as the sum of the absolute value of the difference between the target number and the values guessed. For example : if the target number was 30 and the user guessed 50, 25, 35, and 30, the cumulative error would be calculated as follows:

  |50-30| + |25-30| + |35-30| + |30-30| = 35

  Hint: See http://www.w3schools.com/jsref/jsref_abs.asp for assistance

  - A calculated value called "Average Error" which is calculated as follows: cumulative error / number of valid guesses. Using the above number set, the average error is 8.75.
  - A text feedback response based on the following rules:

- If average error is 10.0 or lower, the message “Incredible guessing!”
- If average error is higher than above but under 20.0, “Good job!”
- If average error is higher than 20 but under 30.0, “Fair!”
- Anything other score: “You are horrible at this game!”

Version 1

The random generator spec will never pass.

guess_game_spec.rb

```
require_relative 'guess_game'

describe GuessGame do
  it "should generate random number between 1 and 100 inclusive" do
    game = GuessGame.new
    result = game.random

    result.should == 50
  end
end
```

guess_game.rb

```
class GuessGame
  def random
    Random.new.rand(1..100)
  end
end
```

Version 2

The above spec deals with the problem of randomness. You cannot use stub to deal with this spec because you will stub yourself out. The spec checks only the range of the generated random number is within the expected range.

guess_game_spec.rb

```
require_relative 'guess_game'

describe GuessGame do
  it "should generate random number between 1 and 100 inclusive" do
    game = GuessGame.new
```

```
    result = game.random

    expected_range = 1..100
    expected_range.should cover(result)
  end
end
```

guess_game.rb

```
class GuessGame
  def random
    Random.new.rand(1..100)
  end
end
```

Note: Using expected.include?(result) is also ok (does not use rspec matcher).

Version 3

guess_game_spec.rb

```
require_relative 'guess_game'

describe GuessGame do
  it "should generate random number between 1 and 100 inclusive" do
    game = GuessGame.new
    result = game.random

    expected = 1..100
    expected.should cover(result)
  end

  it "should display greeting when the game begins" do
    fake_console = mock('Console')
    fake_console.should_receive(:output).with(greeting)
    game = GuessGame.new(fake_console)
    game.start
  end
end
```

This spec shows how you can defer decisions about how you will interact with the user, it could be standard out, GUI, client server app etc. Fake object is injected into the game object.

The interface output is discovered during the mocking and it hides the details about the type of interface that must be implemented to communicate with an user. Game delegates any user interfacing code to a concrete console object therefore it obeys single responsibility principle. Console objects also obey the single responsibility principle.

We could have implemented this similar to the code breaker game in RSpec book by calling the puts method on output variable, by doing so we tie our game object to the implementation. This results in tightly coupled objects which is not desirable. We want loosely coupled objects with high cohesion.

guess_game.rb

```
class GuessGame
  def initialize(console=STDOUT)
    @console = console
  end

  def random
    Random.new.rand(1..100)
  end

  def start
    @console.output("Welcome to the Guessing Game")
  end
end
```

Version 4

Using mock that complies with Gerard Meszaros standard. Use double and if expectation is set, then it is a mock, otherwise it can be used as a stub.

guess_game_spec.rb

```
require_relative 'guess_game'

describe GuessGame do
  it "should generate random number between 1 and 100 inclusive" do
    game = GuessGame.new
    result = game.random

    expected = 1..100
    expected.should cover(result)
  end

  it "should display greeting when the game begins" do
```

```
    fake_console = double('Console')
    fake_console.should_receive(:output).with(greeting)
    game = GuessGame.new(fake_console)
    game.start
  end
end
```

guess_game.rb

```
class GuessGame
  def initialize(console=STDOUT)
    @console = console
  end

  def random
    Random.new.rand(1..100)
  end

  def start
    @console.output("Welcome to the Guessing Game")
  end
end
```

Version 5

Spec exposes the bug : constructor default value is not correct.

guess_game_spec.rb

```
require_relative 'guess_game'

describe GuessGame do
  it "should generate random number between 1 and 100 inclusive" do
    game = GuessGame.new
    result = game.random

    expected = 1..100
    expected.should cover(result)
  end

  it "should display greeting when the game begins" do
    fake_console = double('Console')
    fake_console.should_receive(:output).with(greeting)
    game = GuessGame.new(fake_console)
    game.start
```

```
  end

  it "should display greeting to the standard output when the game begins" do
    game = GuessGame.new
    game.start
  end
end
```

This spec exposes the bug due to wrong default value in the constructor.

guess_game.rb

```
class GuessGame
  def initialize(console=STDOUT)
    @console = console
  end

  def random
    Random.new.rand(1..100)
  end

  def start
    @console.output("Welcome to the Guessing Game")
  end
end
```

Version 6

Fixed the bug due to wrong default value in the constructor. Concrete classes depend on an abstract interface called output and not specific things like puts or gui related method.

guess_game_spec.rb

```
require_relative 'guess_game'

describe GuessGame do
  it "should generate random number between 1 and 100 inclusive" do
    game = GuessGame.new
    result = game.random

    expected = 1..100
    expected.should cover(result)
  end
```

```
  it "should display greeting when the game begins" do
    fake_console = double('Console')
    fake_console.should_receive(:output).with(greeting)
    game = GuessGame.new(fake_console)
    game.start
  end

  it "should display greeting to the standard output when the game begins" do
    game = GuessGame.new
    game.start
  end
end
```

The fix shows how to invert dependencies on concreted classes to abstract interface. In this case the abstract interface is 'output' and not specific method like 'puts' or GUI related method that ties the game logic to a concrete implementation.

standard_output.rb

```
class StandardOutput
  def output(message)
    puts message
  end
end
```

guess_game.rb

```
require_relative 'standard_output'

class GuessGame
  def initialize(console=StandardOutput.new)
    @console = console
  end

  def random
    Random.new.rand(1..100)
  end

  def start
    @console.output("Welcome to the Guessing Game")
  end
end
```

Version 7

Added spec #4. Illustrates the use of as_null_object.

```
 In irb type:
> require 'rspec/mocks/standalone'

s = stub.as_null_object
```

s acts as a dev/null equivalent for tests. It ignores any messages that it receives. Useful for incidental interactions that is not relevant to what is being tested.

guess_game_spec.rb

```
require_relative 'guess_game'

describe GuessGame do
  it "should generate random number between 1 and 100 inclusive" do
    game = GuessGame.new
    result = game.random

    expected = 1..100
    expected.should cover(result)
  end

  it "should display greeting when the game begins" do
    fake_console = double('Console').as_null_object
    fake_console.should_receive(:output).with(greeting)
    game = GuessGame.new(fake_console)
    game.start
  end

  it "should display greeting to the standard output when the game begins" do
    game = GuessGame.new
    game.start
  end

  it "should prompt the user to enter the number representing their guess." do
    fake_console = double('Console').as_null_object
    fake_console.should_receive(:prompt).with('Enter a number between 1 and 100')
    game = GuessGame.new(fake_console)
    game.start
  end
end
```

Spec 4 breaks existing spec 2. It is fixed by using as_null_object which ignores any messages not set as expectation.

guess_game.rb

```
require_relative 'standard_output'

class GuessGame
  def initialize(console=StandardOutput.new)
    @console = console
  end

  def random
    Random.new.rand(1..100)
  end

  def start
    @console.output("Welcome to the Guessing Game")
    @console.prompt("Enter a number between 1 and 100")
  end
end
```

standard_output.rb

```
class StandardOutput
  def output(message)
    puts message
  end

  def prompt(message)
    output(message)
    puts ">"
  end
end
```

Version 8

Added validation. random method deleted because it is required once per game.

guess_game_spec.rb

```
require_relative 'guess_game'

describe GuessGame do
  it "should generate random number between 1 and 100 inclusive" do
```

```
    game = GuessGame.new
    result = game.random

    expected = 1..100
    expected.should cover(result)
  end

  it "should display greeting when the game begins" do
    fake_console = double('Console').as_null_object
    fake_console.should_receive(:output).with(greeting)
    game = GuessGame.new(fake_console)
    game.start
  end

  it "should display greeting to the standard output when the game begins" do
    game = GuessGame.new
    game.start
  end

  it "should prompt the user to enter the number representing their guess." do
    fake_console = double('Console').as_null_object
    fake_console.should_receive(:prompt).with('Enter a number between 1 and 100')
    game = GuessGame.new(fake_console)
    game.start
  end

  it "should perform validation of the guess entered by the user : lower than 1" do
    fake_console = double('Console')
    game = GuessGame.new(fake_console)
    game.guess = 0

    game.error.should == 'The number must be between 1 and 100'
  end

  it "should perform validation of the guess entered by the user : higher than 100" do
    fake_console = double('Console')
    game = GuessGame.new(fake_console)
    game.guess = 101

    game.error.should == 'The number must be between 1 and 100'
  end
end
```

guess_game.rb

```
require_relative 'standard_output'
```

```
class GuessGame
  attr_accessor :guess
  attr_accessor :error
  attr_reader :random

  def initialize(console=StandardOutput.new)
    @console = console
    @random = Random.new.rand(1..100)
  end

  def start
    @console.output("Welcome to the Guessing Game")
    @console.prompt("Enter a number between 1 and 100 to guess the number")
  end

  def guess=(n)
    if (n < 1) or (n > 100)
      @error = 'The number must be between 1 and 100'
    end
  end
end
```

standard_output.rb

```
class StandardOutput
  def output(message)
    puts message
  end

  def prompt(message)
    output(message)
    puts ">"
  end
end
```

Version 9

Refactored specs.

guess_game_spec.rb

```
require_relative 'guess_game'

describe GuessGame do
```

```
  it "should generate random number between 1 and 100 inclusive" do
    game = GuessGame.new
    result = game.random

    expected = 1..100
    expected.should cover(result)
  end

  it "should display greeting when the game begins" do
    fake_console = double('Console').as_null_object
    fake_console.should_receive(:output).with(greeting)
    game = GuessGame.new(fake_console)
    game.start
  end

  it "should display greeting to the standard output when the game begins" do
    game = GuessGame.new
    game.start
  end

  it "should prompt the user to enter the number representing their guess." do
    fake_console = double('Console').as_null_object
    fake_console.should_receive(:prompt).with('Enter a number between 1 and 100')
    game = GuessGame.new(fake_console)
    game.start
  end

  it "should perform validation of the guess entered by the user : lower than 1" do
    game = GuessGame.new
    game.guess = 0

    game.error.should == 'The number must be between 1 and 100'
  end

  it "should perform validation of the guess entered by the user : higher than 100" do
    game = GuessGame.new
    game.guess = 101

    game.error.should == 'The number must be between 1 and 100'
  end

  it "should give clue when the input is valid" do

  end
end
```

Spec 5 and 6 simplified by removing unnecessary double.

guess_game.rb

```
require_relative 'standard_output'

class GuessGame
  attr_accessor :guess
  attr_accessor :error
  attr_reader :random

  def initialize(console=StandardOutput.new)
    @console = console
    @random = Random.new.rand(1..100)
  end

  def start
    @console.output("Welcome to the Guessing Game")
    @console.prompt("Enter a number between 1 and 100")
  end

  def guess=(n)
    if (n < 1) or (n > 100)
      @error = 'The number must be between 1 and 100'
    end
  end
end
```

Version 10

Fixed random test failures by isolating random number generation to its own class (partial stub removed). Methods are smaller and focused.

guess_game_spec.rb

```
require_relative 'guess_game'

describe GuessGame do
  it "should generate random number between 1 and 100 inclusive" do
    game = GuessGame.new
    result = game.random

    expected = 1..100
    expected.should cover(result)
  end
```

```
it "should display greeting when the game begins" do
  fake_console = double('Console').as_null_object
  fake_console.should_receive(:output).with(greeting)
  game = GuessGame.new(fake_console)
  game.start
end

it "should display greeting to the standard output when the game begins" do
  game = GuessGame.new
  game.start
end

it "should prompt the user to enter the number representing their guess." do
  fake_console = double('Console').as_null_object
  fake_console.should_receive(:prompt).with('Enter a number between 1 and 100')
  game = GuessGame.new(fake_console)
  game.start
end

it "should perform validation of the guess entered by the user : lower than 1" do
  game = GuessGame.new
  game.guess = 0

  game.error.should == 'The number must be between 1 and 100'
end

it "should perform validation of the guess entered by the user : higher than 100" do
  game = GuessGame.new
  game.guess = 101

  game.error.should == 'The number must be between 1 and 100'
end

it "should give clue when the input is valid and is less than the computer pick" do
  fake_randomizer = stub(:get => 25)
  fake_console = double('Console').as_null_object
  fake_console.should_receive(:output).with('Your guess is lower')
  game = GuessGame.new(fake_console, fake_randomizer)
  game.stub(:random).and_return { 25 }
  game.guess = 10
end

it "should give clue when the input is valid and is greater than the computer pick" do
  fake_randomizer = stub(:get => 25)
  fake_console = double('Console').as_null_object
```

```
    fake_console.should_receive(:output).with('Your guess is higher')
    game = GuessGame.new(fake_console, fake_randomizer)
    game.guess = 35
  end
end
```

Spec 7 and 8 illustrates use of mocks and partial stubs. Minimize partial stubs and use them only when it is absolutely required.

guess_game.rb

```
require_relative 'standard_output'
require_relative 'randomizer'

class GuessGame
  attr_reader :guess
  attr_accessor :error
  attr_reader :random

  def initialize(console=StandardOutput.new, randomizer=Randomizer.new)
    @console = console
    @random = randomizer.get
  end

  def start
    @console.output("Welcome to the Guessing Game")
    @console.prompt("Enter a number between 1 and 100")
  end

  def guess=(n)
    @guess = n
    give_clue if valid
  end

  private

  def valid
    if (@guess < 1) or (@guess > 100)
      @error = 'The number must be between 1 and 100'
      false
    else
      true
    end
  end

  def give_clue
```

```
    if @guess < @random
      @console.output('Your guess is lower')
    elsif @guess > @random
      @console.output('Your guess is higher')
    else
      # @console.output('Your guess is correct')
    end
  end
end
```

randomizer.rb

```
class Randomizer
  def get
    Random.new.rand(1..100)
  end
end
```

guess_game.rb

```
require_relative 'standard_output'
require_relative 'randomizer'

class GuessGame
  attr_reader :guess
  attr_accessor :error
  attr_reader :random

  def initialize(console=StandardOutput.new, randomizer=Randomizer.new)
    @console = console
    @random = randomizer.get
  end

  def start
    @console.output("Welcome to the Guessing Game")
    @console.prompt("Enter a number between 1 and 100")
  end

  def guess=(n)
    @guess = n
    give_clue if valid
  end

  private
```

```
  def valid
    if (@guess < 1) or (@guess > 100)
      @error = 'The number must be between 1 and 100'
      false
    else
      true
    end
  end

  def give_clue
    if @guess < @random
      @console.output('Your guess is lower')
    elsif @guess > @random
      @console.output('Your guess is higher')
    else
      # @console.output('Your guess is correct')
    end
  end
end
```

Version 11

Added the spec for correct guess. Renamed private method to reflect its abstraction.

guess_game_spec.rb

```
require_relative 'guess_game'

describe GuessGame do
  it "should generate random number between 1 and 100 inclusive" do
    game = GuessGame.new
    result = game.random

    expected = 1..100
    expected.should cover(result)
  end

  it "should display greeting when the game begins" do
    fake_console = double('Console').as_null_object
    fake_console.should_receive(:output).with(greeting)
    game = GuessGame.new(fake_console)
    game.start
  end
```

```
it "should display greeting to the standard output when the game begins" do
  game = GuessGame.new
  game.start
end

it "should prompt the user to enter the number representing their guess." do
  fake_console = double('Console').as_null_object
  fake_console.should_receive(:prompt).with('Enter a number between 1 and 100')
  game = GuessGame.new(fake_console)
  game.start
end

it "should perform validation of the guess entered by the user : lower than 1" do
  game = GuessGame.new
  game.guess = 0

  game.error.should == 'The number must be between 1 and 100'
end

it "should perform validation of the guess entered by the user : higher than 100" do
  game = GuessGame.new
  game.guess = 101

  game.error.should == 'The number must be between 1 and 100'
end

it "should give clue when the input is valid and is less than the computer pick" do
  fake_randomizer = stub(:get => 25)
  fake_console = double('Console').as_null_object
  fake_console.should_receive(:output).with('Your guess is lower')
  game = GuessGame.new(fake_console, fake_randomizer)
  game.stub(:random).and_return { 25 }
  game.guess = 10
end

it "should give clue when the input is valid and is greater than the computer pick" do
  fake_randomizer = stub(:get => 25)
  fake_console = double('Console').as_null_object
  fake_console.should_receive(:output).with('Your guess is higher')
  game = GuessGame.new(fake_console, fake_randomizer)
  game.guess = 35
end

it "should recognize the correct answer when the guess is correct." do
  fake_randomizer = stub(:get => 25)
  fake_console = double('Console').as_null_object
```

```
    fake_console.should_receive(:output).with('Your guess is correct')
    game = GuessGame.new(fake_console, fake_randomizer)
    game.guess = 25
  end
end
```

guess_game.rb

```
require_relative 'standard_output'
require_relative 'randomizer'

class GuessGame
  attr_reader :guess
  attr_accessor :error
  attr_reader :random

  def initialize(console=StandardOutput.new, randomizer=Randomizer.new)
    @console = console
    @random = randomizer.get
  end

  def start
    @console.output("Welcome to the Guessing Game")
    @console.prompt("Enter a number between 1 and 100")
  end

  def guess=(n)
    @guess = n
    give_feedback if valid
  end

  private

  def valid
    if (@guess < 1) or (@guess > 100)
      @error = 'The number must be between 1 and 100'
      false
    else
      true
    end
  end

  def give_feedback
    if @guess < @random
      @console.output('Your guess is lower')
```

```
    elsif @guess > @random
      @console.output('Your guess is higher')
    else
      @console.output('Your guess is correct')
    end
  end
end
```

Version 12

Added contract specs to illustrate how to keep mocks in sync with code.

console_interface_spec.rb

```
shared_examples "Console Interface" do
  describe "Console Interface" do
    it "should implement the console interface: output(arg)" do
      @object.should respond_to(:output).with(1).argument
    end

    it "should implement the console interface: prompt(arg)" do
      @object.should respond_to(:prompt).with(1).argument
    end
  end
end
```

Console Interface spec illustrates how to write contract specs. This avoids the problem of specs passing / failing due to mocks going out of synch with the code. When to use them? If you are using lot of mocks you many not be able to write contract tests for all of them. In this case, think about writing contract tests for the most dependent and important module of your application.

standard_output.rb

```
class StandardOutput
  def output(message)
    puts message
  end
  def prompt(message)
    output(message)
    puts ">"
  end
end
```

standard_output_spec.rb

```
require_relative 'console_interface_spec'
require_relative 'standard_output'

describe StandardOutput do
  before(:each) do
    @object = StandardOutput.new
  end

  it_behaves_like "Console Interface"
end
```

guess_game_spec.rb

```
require_relative 'guess_game'

describe GuessGame do
  it "should generate random number between 1 and 100 inclusive" do
    game = GuessGame.new
    result = game.random

    expected = 1..100
    expected.should cover(result)
  end

  it "should display greeting when the game begins" do
    fake_console = double('Console').as_null_object
    fake_console.should_receive(:output).with(greeting)
    game = GuessGame.new(fake_console)
    game.start
  end

  it "should display greeting to the standard output when the game begins" do
    game = GuessGame.new
    game.start
  end

  it "should prompt the user to enter the number representing their guess." do
    fake_console = double('Console').as_null_object
    fake_console.should_receive(:prompt).with('Enter a number between 1 and 100')
    game = GuessGame.new(fake_console)
    game.start
  end

  it "should perform validation of the guess entered by the user : lower than 1" do
    game = GuessGame.new
```

```
    game.guess = 0

    game.error.should == 'The number must be between 1 and 100'
  end

  it "should perform validation of the guess entered by the user : higher than 100" do
    game = GuessGame.new
    game.guess = 101

    game.error.should == 'The number must be between 1 and 100'
  end

  it "should give clue when the input is valid and is less than the computer pick" do
    fake_randomizer = stub(:get => 25)
    fake_console = double('Console').as_null_object
    fake_console.should_receive(:output).with('Your guess is lower')
    game = GuessGame.new(fake_console, fake_randomizer)
    game.stub(:random).and_return { 25 }
    game.guess = 10
  end

  it "should give clue when the input is valid and is greater than the computer pick" do
    fake_randomizer = stub(:get => 25)
    fake_console = double('Console').as_null_object
    fake_console.should_receive(:output).with('Your guess is higher')
    game = GuessGame.new(fake_console, fake_randomizer)
    game.guess = 35
  end

  it "should recognize the correct answer when the guess is correct." do
    fake_randomizer = stub(:get => 25)
    fake_console = double('Console').as_null_object
    fake_console.should_receive(:output).with('Your guess is correct')
    game = GuessGame.new(fake_console, fake_randomizer)
    game.guess = 25
  end
end
```

guess_game.rb

```
require_relative 'standard_output'
require_relative 'randomizer'

class GuessGame
  attr_reader :guess
```

```
  attr_accessor :error
  attr_reader :random

  def initialize(console=StandardOutput.new, randomizer=Randomizer.new)
    @console = console
    @random = randomizer.get
  end

  def start
    @console.output("Welcome to the Guessing Game")
    @console.prompt("Enter a number between 1 and 100")
  end

  def guess=(n)
    @guess = n
    give_feedback if valid
  end

  private

  def valid
    if (@guess < 1) or (@guess > 100)
      @error = 'The number must be between 1 and 100'
      false
    else
      true
    end
  end

  def give_feedback
    if @guess < @random
      @console.output('Your guess is lower')
    elsif @guess > @random
      @console.output('Your guess is higher')
    else
      @console.output('Your guess is correct')
    end
  end
end
```

Command Query Separation Principle

Objectives

- How to fix Command Query Separation violation ?
- Illustrates how to fix abuse of mocks.
- Illustrates how to write focused tests.
- How to deal with external dependencies in your domain code ?

Before

Example of badly designed API that violates command query separation principle:

```
user = User.new(params)

if user.save
  do something
else
  do something else
end
```

The save is inserting the record in the database. It is a command because it has a side effect. It is also returning true or false so it is also a query.

After

```
user = User.new(params)
user.save

if user.persisted?
  do something
else
  do something else
end
```

Calculator Example

Before

Calculator example that violates command query separation principle.

calculator_spec.rb

```
require_relative 'calculator'

describe Calculator, "Computes addition of given two numbers" do

  it "should add given two numbers that are not trivial" do
    calculator = Calculator.new
    result = calculator.add(1,2)

    result.should == 3
  end
end
```

calculator.rb

```
class Calculator
  def add(x,y)
    x+y
  end
end
```

After

Fixed the command query separation violation.

calculator_spec.rb

```
require_relative 'calculator'

describe Calculator, "Computes addition of given two numbers" do
  it "should add given two numbers that are not trivial" do
    calculator = Calculator.new
    calculator.add(1,2)
    result = calculator.result

    result.should == 3
  end
end
```

add is a command. calculator.result is query.

calculator.rb

```ruby
class Calculator
  attr_reader :result

  def add(x,y)
    @result = x + y
    nil
  end
end
```

Tweet Analyser Example

Another Command Query Separation Principle violation example.

Before

Version 1 - tweet_analyser_spec.rb

```ruby
class TweetAnalyzer
  def initialize(user)
    @user = user
  end

  def word_frequency
    {"one" => 1}
  end
end

describe TweetAnalyzer do
  it "finds the frequency of words in a user's tweets" do
    user = double('user')
    analyzer = TweetAnalyzer.new(user)
    histogram = analyzer.word_frequency
    histogram["one"].should == 1
  end
end
```

It looks like client is tied to the implementation details (it is accessing a data structure) but it is actually any class that can respond to [] method.

After

Version 2 - tweet_analyser_spec.rb

```ruby
class TweetAnalyzer
  def initialize(user)
    @user = user
  end

  def word_frequency
    @histogram = {"one" => 1}
  end

  def histogram(text)
    @histogram[text]
  end
end

describe TweetAnalyzer do
  it "finds the frequency of words in a user's tweets" do
    user = double('user')
    analyzer = TweetAnalyzer.new(user)
    analyzer.word_frequency

    analyzer.histogram("one").should == 1
  end
end
```

Version 3

Second spec breaks the existing spec. This is an example for how mocks are abused.

tweet_analyzer_spec.rb

```ruby
class TweetAnalyzer
  def initialize(user)
    @user = user
  end

  def word_frequency
    @frequency = Hash.new{0}
    @user.recent_tweets.each do |tweet|
      tweet.split(/\s/).each do |word|
        @frequency[word] += 1
      end
    end
  end
```

```
  def histogram(text)
    @frequency[text]
  end
end

describe TweetAnalyzer do
  it "finds the frequency of words in a user's tweets" do
    user = double('user')
    analyzer = TweetAnalyzer.new(user)
    analyzer.word_frequency

    analyzer.histogram("one").should == 1
  end

  it "asks the user for recent tweets" do
    user = double('user')
    analyzer = TweetAnalyzer.new(user)
    expected_tweets = ["one two", "two"]
    user.should_receive(:recent_tweets).and_return expected_tweets

    histogram = analyzer.word_frequency
    analyzer.histogram("two").should == 2
  end
end
```

Version 4

Fixed abuse of mocks.

tweet_analyzer_spec.rb

```
class TweetAnalyzer
  def initialize(user)
    @user = user
  end

  def word_frequency
    @frequency = Hash.new{0}
    @user.recent_tweets.each do |tweet|
      tweet.split(/\s/).each do |word|
        @frequency[word] += 1
      end
    end
  end
```

```
  def histogram(text)
    @frequency[text]
  end
end

describe TweetAnalyzer do
  it "finds the frequency of words in a user's tweets" do
    user = double('user')
    expected_tweets = ["one two", "two"]
    user.stub(:recent_tweets).and_return expected_tweets

    analyzer = TweetAnalyzer.new(user)
    analyzer.word_frequency

    analyzer.histogram("one").should == 1
  end

  it "asks the user for recent tweets" do
    user = double('user')
    expected_tweets = ["one two", "two"]
    user.stub(:recent_tweets).and_return expected_tweets

    analyzer = TweetAnalyzer.new(user)
    analyzer.word_frequency

    analyzer.histogram("two").should == 2
  end
end
```

Version 5

Extracted common setup to before(:each) method.

tweet_analyzer_spec.rb

```
class TweetAnalyzer
  def initialize(user)
    @user = user
  end

  def word_frequency
    @frequency = Hash.new{0}
    @user.recent_tweets.each do |tweet|
      tweet.split(/\s/).each do |word|
        @frequency[word] += 1
```

```
      end
    end
  end

  def histogram(text)
    @frequency[text]
  end
end

describe TweetAnalyzer do
  before(:each) do
    @user = double('user')
    expected_tweets = ["one two", "two"]
    @user.stub(:recent_tweets).and_return expected_tweets
  end

  it "finds the frequency of words in a user's tweets" do
    analyzer = TweetAnalyzer.new(@user)
    analyzer.word_frequency

    analyzer.histogram("one").should == 1
  end

  it "asks the user for recent tweets" do
    analyzer = TweetAnalyzer.new(@user)
    analyzer.word_frequency

    analyzer.histogram("two").should == 2
  end
end
```

Version 6

Focused tests that test only one thing. If it is important that the user's recent tweets are used to calculate the frequency, write a separate test for that.

tweet_analyzer_spec.rb

```
class TweetAnalyzer
  def initialize(user)
    @user = user
  end

  def word_frequency
    @frequency = Hash.new{0}
```

```
    @user.recent_tweets.each do |tweet|
      tweet.split(/\s/).each do |word|
        @frequency[word] += 1
      end
    end
  end

  def histogram(text)
    @frequency[text]
  end
end

describe TweetAnalyzer do
  before(:each) do
    @user = double('user')
    expected_tweets = ["one two", "two"]
    @user.stub(:recent_tweets).and_return expected_tweets
  end

  it "finds the frequency of words in a user's tweets" do
    analyzer = TweetAnalyzer.new(@user)
    analyzer.word_frequency

    analyzer.histogram("one").should == 1
  end

  it "find the frequency of words in a user's tweets that appears multiple times" do
    analyzer = TweetAnalyzer.new(@user)
    analyzer.word_frequency

    analyzer.histogram("two").should == 2
  end

  it "asks the user for recent tweets" do
    user = double('user')
    expected_tweets = ["one two", "two"]
    user.should_receive(:recent_tweets).and_return expected_tweets

    analyzer = TweetAnalyzer.new(user)
    analyzer.word_frequency
  end

end
```

Version 7

Refactored version.

tweet_analyzer_spec.rb

```
require_relative 'tweet_analyzer'

describe TweetAnalyzer do

  context 'The Usual Specs' do
    before(:each) do
      @user = double('user')
      expected_tweets = ["one two", "two"]
      @user.stub(:recent_tweets).and_return expected_tweets
    end

    it "finds the frequency of words in a user's tweets" do
      analyzer = TweetAnalyzer.new(@user)
      analyzer.word_frequency

      analyzer.histogram("one").should == 1
    end

    it "find the frequency of words in a user's tweets that appears multiple times" do
      analyzer = TweetAnalyzer.new(@user)
      analyzer.word_frequency

      analyzer.histogram("two").should == 2
    end
  end

  context 'Calling recent_tweets is important' do
    it "asks the user for recent tweets" do
      user = double('user')
      expected_tweets = ["one two", "two"]
      user.should_receive(:recent_tweets).and_return expected_tweets

      analyzer = TweetAnalyzer.new(user)
      analyzer.word_frequency
    end
  end

end
```

tweet_analyzer.rb

```ruby
class TweetAnalyzer
  def initialize(user)
    @user = user
  end

  def word_frequency
    @frequency = Hash.new{0}
    @user.recent_tweets.each do |tweet|
      tweet.split(/\s/).each do |word|
        @frequency[word] += 1
      end
    end
  end

  def histogram(text)
    @frequency[text]
  end
end
```

Notes from Martin Fowler's article and jMock Home Page

Testing and Command Query Separation Principle

The term 'command query separation' was coined by Bertrand Meyer in his book 'Object Oriented Software Construction'.

The fundamental idea is that we should divide an object's methods into two categories:

```
Queries: Return a result and do not change the observable state
                    of the system (are free of side effects).
Commands: Change the state of a system but do not return a value.
```

Because the term 'command' is widely used in other contexts it is referred as 'modifiers' and 'mutators'.

It's useful if you can clearly separate methods that change state from those that don't. This is because you can use queries in many situations with much more confidence, changing their order. You have to be careful with modifiers.

The return type is the give-away for the difference. It's a good convention because most of the time it works well. Consider iterating through a collection in Java: the next method both gives the next item in the collection and advances the iterator. It's preferable to separate advance and current methods.

There are exceptions. Popping a stack is a good example of a modifier that modifies state. Meyer correctly says that you can avoid having this method, but it is a useful idiom. Follow this principle when you can.

From jMock home page: Tests are kept flexible when we follow this rule of thumb: Stub queries and expect commands, where a query is a method with no side effects that does nothing but query the state of an object and a command is a method with side effects that may, or may not, return a result. Of course, this rule does not hold all the time, but it's a useful starting point.