Essential TDD

Bala Paranj

Section 1 : Basics

This section is about the basics of TDD. It introduces the concepts using code exercises. It is deliberately code centric with concise explanation. You will get the most benefit out of the book by first making an attempt to write the tests by reading the problem description in each example. Then look at the solution and compare it with your version. If you are stuck just type in the code from the book and run the examples to see how it works.

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

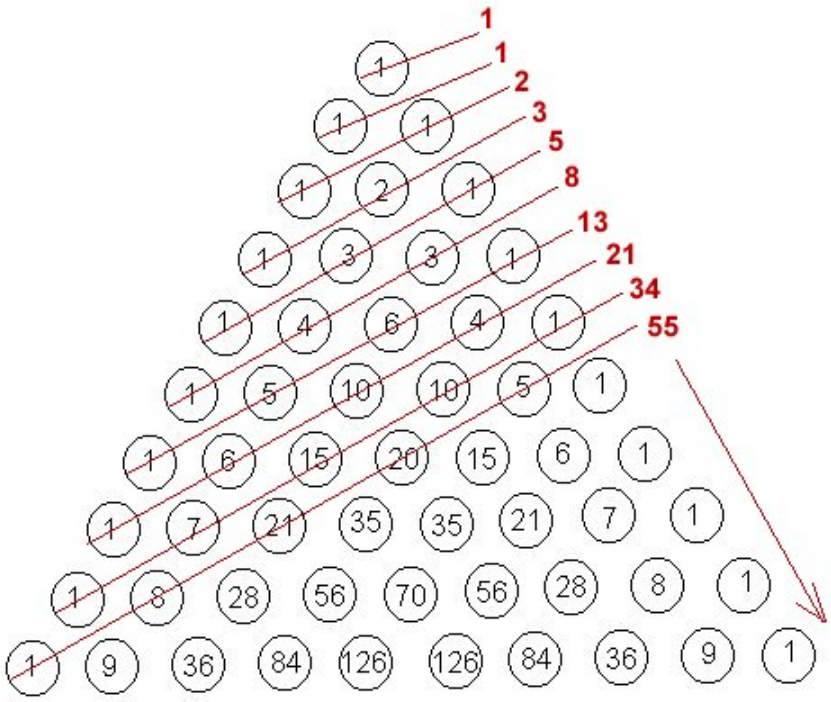

Figure 2: Fibonacci Numbers

Guidelines

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

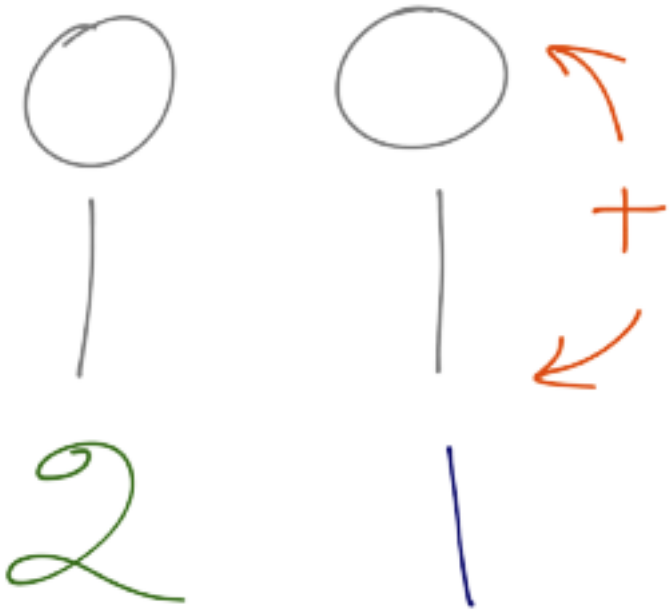

Figure 3: Calculating Fibonacci Numbers

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

```ruby
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

```ruby
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

```ruby
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

```
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

```
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

```
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

```
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

```
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.
```

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 commands.

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. # Test Spy #

Objective

- Using Stubs with Test Spy in Ruby

Problem

I came across a problem during testing. I had to test the cookie setting logic of my controllers. It was straightforward to test that the cookie was set for the happy path. For the alternative scenario it became tricky to test because RSpec and Rails framework did not play that well together. I even read Devise Rails plugin code to see how Jose Valim handled cookie related problems during testing. No luck. One solution I found was on Stackoverflow: How do I test cookie expiry?

app/controllers/widget_controller.rb

```
...
def index
    cookies[:expiring_cookie] = { :value   => 'All that we see or seem...',
                                  :expires => 1.hour.from_now }
end
...
```

spec/controllers/widget_controller_spec.rb

```
...
it "sets the cookie" do
  get :index
  response.cookies['expiring_cookie'].should eq('All that we see or seem...')
end

it "sets the cookie expiration" do
  stub_cookie_jar = HashWithIndifferentAccess.new
  controller.stub(:cookies) { stub_cookie_jar }

  get :index
  expiring_cookie = stub_cookie_jar['expiring_cookie']
  expiring_cookie[:expires].to_i.should be_within(1).of(1.hour.from_now.to_i)
end
```

This technique is a great example of Test Spy described in Gerard Meszaros book xUnit Test Patterns. Basically, you install a spy and check the results collected by the test spy in the verification phase. In this case the Hash is the Test Spy that collects data. See how the stub is used to install the spy in the SUT? It overcomes the problems and isolates the SUT from the Rails framework and allows the code to be tested easily.

In my TDD bootcamps, the topic on Stubs and Mocks generates lot of discussion. To clear confusion that surrounds the stubs and mocks, I would state : Read Martin Fowler's paper on Mocks Aren't Stubs. Stub can never fail your test,

only mocks can fail your test. Using stubs in combination with a spy like this makes stubs seem like they can in fact fail your test. But only the data collected by the Test Spy decides whether the test passes or not. So the stub's main purpose is to just isolate the production code from Rails framework and allow access to the internal state of the SUT where there is no direct way to access it. # Honey Boo Boo #

Scanner

Objectives

- How to use Fakes and Mocks ?
- When to delete a test ?

Writing the First Test

The first example is about a scanner that is used in a checkout counter. You can scan item, the name and price of the item is sent to the output console.

scanner_spec.rb

```
require_relative 'scanner'

describe Scanner do

  it 'should respond to scan with barcode as the input parameter' do
    scanner = Scanner.new

    scanner.should respond_to(:scan)
  end
end
```

scanner.rb

```
class Scanner
  def scan

  end
end
```

The first spec does not do much. The main purpose of writing the first spec is to help setup the directory structure, require statements etc to get the specs

running. You can run this spec by typing the following command from the root of the project:

$ rspec scanner/scanner_spec.rb

In your home directory create a .rspec directory with the following contents:

–color –format documentation

This will show the output in color and formatted to read as documentation. The doc string says that the barcode is the input parameter. Let's add this detail to our spec:

```
require_relative 'scanner'

describe Scanner do

  it 'should respond to scan with barcode as the input argument' do
    scanner = Scanner.new

    scanner.should respond_to(:scan).with(1)
  end
end
```

Run the test, watch it fail due to the input parameter and change the scanner.rb as follows :

```
class Scanner

  def scan(barcode)

  end
end
```

The test now passes.

Deleting a Test

The first test is no longer required. It is like a scaffold of a building once we complete the construction of the building the scaffold will go away. Here is the new test that captures the description in the first paragraph of this chapter:

scanner_spec.rb

```
require_relative 'scanner'
require_relative 'r2d2_display'
```

```ruby
describe Scanner do

  it "scan & display the name & price of the scanned item on a cash register display" do
    real_display = R2d2Display.new
    scanner = Scanner.new(real_display)
    scanner.scan("1")

    real_display.last_line_item.should == "Milk $3.99"
  end
end
```

r2d2_display.rb

```ruby
class R2d2Display
  attr_reader :last_line_item

  def display(line_item)
    p "Executing complicated logic"
    sleep 5

    @last_line_item = "Milk $3.99"
  end
end
```

Real object used in the test is slow. Here is the scanner.rb to make the new test pass:

```ruby
class Scanner
  def initialize(display)
    @display = display
  end

  def scan(barcode)
    @display.display("Milk $3.99")
  end
end
```

Speeding Up The Test

How can we test if the scanner can scan a given item and display the item name and price on a cash register display? Let's speed up the test by using a fake display. The scanner_spec.rb now becomes:

```
require_relative 'scanner'
require_relative 'fake_display'

describe Scanner do

  it "scan & display the name & price of the scanned item on a cash register display" do
    fake_display = FakeDisplay.new
    scanner = Scanner.new(fake_display)
    scanner.scan("1")

    fake_display.last_line_item.should == "Milk $3.99"
  end
end
```

fake_display.rb

```
class FakeDisplay
  attr_reader :last_line_item

  def display(line_item)
    @last_line_item = "Milk $3.99"
  end
end
```

The spec now runs fast. This solution assumes that we can access the last line item to display by doing:

```
attr_reader :last_line_item
```

We broke the dependency on external display object by using a fake object that mimicked the interface of the real object. Dependency injection is used to create scanner object with a fake display.

When we write tests, we have to divide and conquer. This test tells us how scanner objects affect displays. This test helps us to see whether a problem is due to scanner. Is scanner fulfilling its responsibility?. This helps us localize errors and save time.

When we write tests for individual units, we end up with small well-understood pieces. This makes it easy to reason about code.

Using Mocks

Writing a lot of fakes can become tedious. In such cases, mocks can be used. Mock objects are fakes that perform assertions internally. The solution that uses mocks is faster than using Fake display object.

```
require_relative 'scanner'

describe Scanner do

  it "scan & display the name & price of the scanned item on a cash register display" do
    fake_display = mock
    fake_display.should_receive(:display).with("Milk $3.99")
    scanner = Scanner.new(fake_display)
    scanner.scan("1")
  end
end
```

The display method is under our control so we can use Mock. Mock is a design technique that is used to discover API. This is an example of right way to use Mock.

Reference

Working Effectively with Legacy Code

Uncommenter

Objective

- Using fake objects to speed up test

The Ugly Before Version

test_file.rb

```
# This is a comment
This is not a comment
# Another comment
```

uncommenter_spec.rb

```
require_relative 'uncommenter'

describe Uncommenter do
  it "should uncomment a given file" do
    infile = File.new(Dir.pwd + "/uncommenter/test_file.rb")
```

```
    outfile = File.new(Dir.pwd + "/uncommenter/test_file.rb.out", "w")

    Uncommenter.uncomment(infile, outfile)
    outfile.close

    resultfile = File.open(Dir.pwd + "/uncommenter/test_file.rb.out","r")
    result_string = resultfile.read
    result_string.should == "This is not a comment\n"
    resultfile.close
  end
end
```

uncommenter.rb

```
class Uncommenter
  def self.uncomment(infile, outfile)
    infile.each do |line|
      outfile.print line unless line =~ /\A\s*#/
    end
  end
end
```

This requires manual deleting of the file test_file.rb.out after every test run. Also whenever you access a file system, it is not a unit test anymore. It will run slow. It becomes an integration test and requires setup and cleanup of external resources.

The Sexy After Version

Here is the spec that runs fast:

uncommenter_spec.rb

```
require_relative 'uncommenter'
require 'stringio'

describe Uncommenter do
  it "should uncomment a given file" do
    input = <<-EOM
    # This is a comment.
      This is not a comment.
    # This is another comment
    EOM
    infile = StringIO.new(input)
```

```
    outfile = StringIO.new("")

    Uncommenter.uncomment(infile, outfile)

    result_string = outfile.string
    result_string.strip.should == "This is not a comment."
  end
end
```

This example illustrates using Ruby builtin StringIO as a Fake object. File accessing is involved with using the right read or write mode. It requires closing and opening the file at the appropriate times.

StringIO is a ruby builtin class that mimics the interface of the file. This version of spec runs faster than the file accessing version. The spec is also smaller. In this case, StringIO is a Fake object. You don't have to manually write and maintain a Fake object for file processing. Just use the StringIO.

To run the spec:

rspec uncommenter/uncommenter_spec.rb –format doc –color

Reference

The Well Grounded Rubyist

Canonical Test Structure

Objective : Canonical test structure practice for Given, When, Then

```
    - Given - Precondition
    - When  - Exercise the SUT
    - Then  - Postcondition
    - Example uses State Verification
```

stack_spec.rb

```
require_relative 'stack'

describe Stack do
  it "should push a given item" do
    stack = Stack.new
    stack.push(1)
```

```
    stack.size.should == 1
  end
  it "should pop from the stack" do
    stack = Stack.new
    stack.push(2)
    result = stack.pop

    result.should == 2
    stack.size.should == 0
  end
end
```

Simple stack implementation that can push and pop.

stack.rb

```
class Stack
  def initialize
    @elements = []
  end
  def push(item)
    @elements << item
  end
  def pop
    @elements.pop
  end
  def size
    @elements.size
  end
end
```

Identifying Given, When, Then

Here is an example of how to identify Given, When, Then in a test.

Copy the following given_when_then.rb to canonical directory:

```
def Given
  yield
end

def When
  yield
end
```

```
def Then
  yield
end
```

Now the stack_spec.rb looks like this:

```
require_relative 'stack'
require_relative 'given_when_then'

describe Stack do
  it "should push a given item" do
    Given { @stack = Stack.new }

    When  { @stack.push(1) }

    Then  { @stack.size.should == 1 }
  end
  it "should pop from the stack" do
    stack = Stack.new
    stack.push(2)
    result = stack.pop
    result.should == 2
    stack.size.should == 0
  end
end
```

Exercise

Identify the Given, When, Then for the second spec "should pop from the stack".

Code Mutation

Objective

To illustrate the need to mutate the code when the test passes without failing the first time.

The ruby_extensions.rb has extensions to builtin Ruby classes that preserves the semantics. It provides:

- Array union and intersection methods.

- Fixnum inclusive and exclusive methods

ruby_extensions_spec.rb

```
require_relative 'ruby_extensions'

describe Array do
  it "return an array with elements common to both arrays with no duplicates" do
    a = [1,1,3,5]
    b = [1,2,3]
    result = a.intersection(b)

    result.should == [1,3]
  end

  it "return a new array built by concatenating two arrays" do
    a = [1,2,3]
    b = [4,5]
    result = a.union(b)

    result.should == [1,2,3,4,5]
  end

  it "should include the end value for an inclusive range" do
    a = 0.inclusive(2)

    a.first.should == 0
    a.last.should == 2
    a.include?(1).should be_true
    a.include?(2).should be_true
  end

  it "should exclude the end value for an exclusive range" do
    a = 0...2

    a.first.should == 0
    a.last.should == 2
    a.include?(1).should be_true
    a.include?(2).should be_false
  end

  it "should return a comma separated list of items when to_s is called" do
    a = [1,2,3,4]
    result = a.to_s
```

```
    result.should == "1,2,3,4"
  end
end
```

ruby_extensions.rb

```
class Array
  # | operator is used for union operation in Array.
  def union(another)
    self | another
  end
  # & operator is used for intersection operation in Array.
  def intersection(another)
    self & another
  end
  # Better implementation that the default one provided by array
  def to_s
    join(",")
  end
end

class Fixnum
  # This eliminates the mental mapping from .. and ... to the behaviour of the methods.
  def inclusive(element)
    self..element
  end

  def exclusive(element)
    self...element
  end
end
```

When the test passes without failing, you must modify the production code to make the test fail to make sure that you the test is testing the right thing. This example illustrates:

```
- How to open classes that preserves the semantics of the core classes.
- What to do when the test passes without failing the first time.
- Hiding implementation related classes.
- Intention revealing variable names.
```

Eliminating Loops

Objective

To illustrate how to eliminate loops in specs. The tests must specify and focus on "What" instead of implementation, the "How".

Here is the code from Meszaros gem https://github.com/bparanj/meszaros.git:

loop_spec.rb

```
require 'spec_helper'
require 'meszaros/loop'

module Meszaros
  describe Loop do
    it "should allow data driven spec : 0" do
      result = []
      Loop.data_driven_spec([]) do |element|
        result << element
      end

      result.should be_empty
    end

    it "should allow data driven spec : 1" do
      result = []
      Loop.data_driven_spec([4]) do |element|
        result << element
      end

      result.should == [4]
    end

    it "should allow data driven spec : n" do
      result = []
      Loop.data_driven_spec([1,2,3,4]) do |element|
        result << element
      end

      result.should == [1,2,3,4]
    end

    it "should raise exception when nil is passed as the parameter" do
      expect do
        Loop.data_driven_spec(nil) do |element|
```

```
      true.should be_true
    end
  end.to raise_error

end

it "allow execution of a chunk of code for 0 number of times" do
  result = 0

  Loop.repeat(0) do
    result += 1
  end

  result.should == 0
end

it "allow execution of a chunk of code for 1 number of times" do
  result = 0

  Loop.repeat(1) do
    result += 1
  end

  result.should == 1
end

it "raise exception when nil is passed for the parameter to repeat" do
  expect do
    Loop.repeat(nil) do
      true.should be_true
    end
  end.to raise_error

end

it "raise exception when string is passed for the parameter to repeat" do
  expect do
    Loop.repeat("dumb") do
      true.should be_true
    end
  end.to raise_error
end

it "raise exception when float is passed for the parameter to repeat" do
  expect do
    Loop.repeat(2.2) do
```

```
          true.should be_true
        end
      end.to raise_error
    end

    it "allow execution of a chunk of code for n number of times" do
      result = 0

      Loop.repeat(3) do
        result += 1
      end

      result.should == 3
    end
  end
end
```

loop.rb

```
module Meszaros
  class Loop
    def self.data_driven_spec(container)
      container.each do |element|
        yield element
      end
    end

    def self.repeat(n)
      n.times { yield }
    end
  end
end
```

From the specs, you can see the cases 0, 1 and n. We gradually increase the complexity of the tests and extend the solution to a generic case of n. It also documents the behavior for illegal inputs. The developer can see how the API works by reading the specs. Data driven spec and repeat methods are available in meszaros gem.

1. See meszaros gem for how to eliminate loops in specs.
2. Data driven spec and repeat methods are available in meszaros gem.

From Alex Chaffe's presentation: https://github.com/alexch/test-driven

Before

Matrix Test

```
%w(a e i o u).each do |letter|
  it "#{letter} is a vowel" do
    assert { letter.vowel? }
  end
end
```

This mixes what and how.

After

Data Driven Spec

```
specify "a, e, i, o, u are the vowel set" do
  data_driven_spec(%w(a e i o u)) do |letter|
    letter.should be_vowel
  end
end
```

This is a specification that focuses only on "What".

Angry Rock

Objectives

- How to fix Command Query Separation violation?
- Refactoring : Retaining the old interface and the new one at the same time to avoid old tests from failing.
- Semantic quirkiness of Well Grounded Rubyist solution exposed by specs.
- Using domain specific terms to make the code expressive

Version 1 - Violation of Command Query Separation Principle

angry_rock_spec.rb

```
require 'spec_helper'

module Game
  describe AngryRock do
   it "should pick paper as the winner over rock" do
     choice_1 = Game::AngryRock.new(:paper)
     choice_2 = Game::AngryRock.new(:rock)
     winner = choice_1.play(choice_2)
     result = winner.move

     result.should == "paper"
   end
   it "picks scissors as the winner over paper" do
     choice_1 = Game::AngryRock.new(:scissors)
     choice_2 = Game::AngryRock.new(:paper)
     winner = choice_1.play(choice_2)
     result = winner.move

     result.should == "scissors"
   end
   it "picks rock as the winner over scissors " do
     choice_1 = Game::AngryRock.new(:rock)
     choice_2 = Game::AngryRock.new(:scissors)
     winner = choice_1.play(choice_2)
     result = winner.move

     result.should == "rock"
   end
   it "results in a tie when the same choice is made by both players" do
     [:rock, :paper, :scissors].each do |choice|
       choice_1 = Game::AngryRock.new(choice)
       choice_2 = Game::AngryRock.new(choice)
       winner = choice_1.play(choice_2)

       winner.should be_false
     end
   end
  end
end
```

angry_rock.rb

```
module Game
  class AngryRock
    include Comparable
```

```
    WINS = [ %w{rock scissors}, %w{scissors paper}, %w{paper rock}]

    attr_accessor :move

    def initialize(move)
      @move = move.to_s
    end
    def <=>(other)
      if move == other.move
        0
      elsif WINS.include?([move, other.move])
        1
      elsif WINS.include?([other.move, move])
        -1
      else
        raise ArgumentError, "Something's wrong"
      end
    end
    # Lousy design : Returns boolean instead of AngryRock winner object
    def play(other)
      if self > other
        self
      elsif other > self
        other
      else
        false
      end
    end
  end
end
```

Notice the play method implementation, the false case breaks the consistency of the returned value and violates the semantics of the API. Also the play is a "Command" not a "Query". This method violates the "Command Query Separation Principle".

Fixing the Bad Design

angry_rock_spec.rb

```
require 'spec_helper'

module Game
  describe AngryRock do
```

```
    it "should pick paper as the winner over rock" do
      choice_1 = Game::AngryRock.new(:paper)
      choice_2 = Game::AngryRock.new(:rock)
      winner = choice_1.play(choice_2)
      result = winner.move

      result.should == "paper"
    end
    it "picks scissors as the winner over paper" do
      choice_1 = Game::AngryRock.new(:scissors)
      choice_2 = Game::AngryRock.new(:paper)
      winner = choice_1.play(choice_2)
      result = winner.move

      result.should == "scissors"
    end
    it "picks rock as the winner over scissors " do
      choice_1 = Game::AngryRock.new(:rock)
      choice_2 = Game::AngryRock.new(:scissors)
      winner = choice_1.play(choice_2)
      result = winner.move

      result.should == "rock"
    end
    it "results in a tie when the same choice is made by both players" do
      choice_1 = Game::AngryRock.new(:rock)
      choice_2 = Game::AngryRock.new(:rock)
      winner = choice_1.play(choice_2)
      result = winner.move

      result.should == "TIE!"
    end
  end
end
```

angry_rock.rb

```
module Game
  class AngryRock
    include Comparable

    WINS = [ %w{rock scissors}, %w{scissors paper}, %w{paper rock}]

    attr_accessor :move
```

```
    def initialize(move)
      @move = move.to_s
    end
    def <=>(other)
      if move == other.move
        0
      elsif WINS.include?([move, other.move])
        1
      elsif WINS.include?([other.move, move])
        -1
      else
        raise ArgumentError, "Something's wrong"
      end
    end
    # Fixed design : Returns AngryRock Tie object for the Tie case.
    def play(other)
      if self > other
        self
      elsif other > self
        other
      else
        AngryRock.new("TIE!")
      end
    end
  end
end
```

The play method now returns a AngryRock tie object for the tie case.

Tie Cases : Spec Duplication

angry_rock_spec.rb

```
require 'spec_helper'

module Game
  describe AngryRock do
   it "should pick paper as the winner over rock" do
     choice_1 = Game::AngryRock.new(:paper)
     choice_2 = Game::AngryRock.new(:rock)
     winner = choice_1.play(choice_2)
     result = winner.move

     result.should == "paper"
   end
```

```
    it "picks scissors as the winner over paper" do
      choice_1 = Game::AngryRock.new(:scissors)
      choice_2 = Game::AngryRock.new(:paper)
      winner = choice_1.play(choice_2)
      result = winner.move

      result.should == "scissors"
    end
    it "picks rock as the winner over scissors " do
      choice_1 = Game::AngryRock.new(:rock)
      choice_2 = Game::AngryRock.new(:scissors)
      winner = choice_1.play(choice_2)
      result = winner.move

      result.should == "rock"
    end
    it "results in a tie when the same choice is made by both players : rock" do
      choice_1 = Game::AngryRock.new(:rock)
      choice_2 = Game::AngryRock.new(:rock)
      winner = choice_1.play(choice_2)
      result = winner.move

      result.should == "TIE!"
    end

    it "results in a tie when the same choice is made by both players : paper" do
      choice_1 = Game::AngryRock.new(:paper)
      choice_2 = Game::AngryRock.new(:paper)
      winner = choice_1.play(choice_2)
      result = winner.move

      result.should == "TIE!"
    end
    it "results in a tie when the same choice is made by both players : scissors" do
      choice_1 = Game::AngryRock.new(:scissors)
      choice_2 = Game::AngryRock.new(:scissors)
      winner = choice_1.play(choice_2)
      result = winner.move

      result.should == "TIE!"
    end
  end
end
```

The last three specs show three possible tie scenarios.

Removing the Duplication in Specs : The Before Picture

angry_rock_spec.rb

```
require 'spec_helper'

module Game
  describe AngryRock do
   it "should pick paper as the winner over rock" do
     choice_1 = Game::AngryRock.new(:paper)
     choice_2 = Game::AngryRock.new(:rock)
     winner = choice_1.play(choice_2)
     result = winner.move

     result.should == "paper"
   end
   it "picks scissors as the winner over paper" do
     choice_1 = Game::AngryRock.new(:scissors)
     choice_2 = Game::AngryRock.new(:paper)
     winner = choice_1.play(choice_2)
     result = winner.move

     result.should == "scissors"
   end
   it "picks rock as the winner over scissors " do
     choice_1 = Game::AngryRock.new(:rock)
     choice_2 = Game::AngryRock.new(:scissors)
     winner = choice_1.play(choice_2)
     result = winner.move

     result.should == "rock"
   end
   it "results in a tie when the same choice is made by both players" do
     [:rock, :paper, :scissors].each do |choice|
       choice_1 = Game::AngryRock.new(choice)
       choice_2 = Game::AngryRock.new(choice)
       winner = choice_1.play(choice_2)
       result = winner.move

       result.should == "TIE!"
     end
   end
  end
end
```

The duplication in specs is removed by using a loop.

Removing the Duplication in Specs : The After Picture

angry_rock_spec.rb

```
require 'spec_helper'

module Game
  describe AngryRock do
   it "should pick paper as the winner over rock" do
     choice_1 = Game::AngryRock.new(:paper)
     choice_2 = Game::AngryRock.new(:rock)
     winner = choice_1.play(choice_2)
     result = winner.move

     result.should == "paper"
   end
   it "picks scissors as the winner over paper" do
     choice_1 = Game::AngryRock.new(:scissors)
     choice_2 = Game::AngryRock.new(:paper)
     winner = choice_1.play(choice_2)
     result = winner.move

     result.should == "scissors"
   end
   it "picks rock as the winner over scissors " do
     choice_1 = Game::AngryRock.new(:rock)
     choice_2 = Game::AngryRock.new(:scissors)
     winner = choice_1.play(choice_2)
     result = winner.move

     result.should == "rock"
   end
   it "results in a tie when the same choice is made by both players" do
     data_driven_spec([:rock, :paper, :scissors]) do |choice|
       choice_1 = Game::AngryRock.new(choice)
       choice_2 = Game::AngryRock.new(choice)
       winner = choice_1.play(choice_2)
       result = winner.move

       result.should == "TIE!"
     end
   end
  end
end
```

spec_helper.rb

```
require 'game/angry_rock'

def data_driven_spec(container)
  container.each do |element|
   yield element
  end
end
```

Original solution had the following logic :

```
if winner
  result = winner.move
else
  result = "TIE!"
end
```

with play returning false for a tie scenario.

Command Query Separation Principle

angry_rock.rb

```
module Game
  class AngryRock
    include Comparable

    WINS = [ %w{rock scissors}, %w{scissors paper}, %w{paper rock}]

    attr_accessor :move

    def initialize(move)
      @move = move.to_s
    end
    def <=>(other)
      if move == other.move
        0
      elsif WINS.include?([move, other.move])
        1
      elsif WINS.include?([other.move, move])
        -1
      else
        raise ArgumentError, "Something's wrong"
      end
    end
```

```
    def play(other)
      if self > other
        self
      elsif other > self
        other
      else
        AngryRock.new("TIE!")
      end
    end
  end
end
```

Is the play() method a command and a query? It is ambiguous because play seems to be a name of a command and it is returning the winning AngryRock object (result of a query operation). It combines command and query.

Refactoring While Staying Green

angry_rock.rb

```
module Game
  class AngryRock
    include Comparable

    WINS = [ %w{rock scissors}, %w{scissors paper}, %w{paper rock}]

    attr_accessor :move

    def initialize(move)
      @move = move.to_s
    end
    def <=>(other)
      if move == other.move
        0
      elsif WINS.include?([move, other.move])
        1
      elsif WINS.include?([other.move, move])
        -1
      else
        raise ArgumentError, "Something's wrong"
      end
    end
    # Problem : Is this method is a command and a query?
    # It is ambiguous because play seems to be a name of a command and
    # it is returning the winning AngryRock object
```

```
    def play(other)
      if self > other
        self
      elsif other > self
        other
      end
    end
    def winner(other)
      if self > other
        self
      elsif other > self
        other
      end
    end
  end

  class Play
    def initialize(first_choice, second_choice)
      @winner = first_choice.winner(second_choice)
    end
    def has_winner?
      !@winner.nil?
    end
    def winning_move
      @winner.move
    end
  end
end
```

Retaining the old interface and the new one at the same time to avoid old tests from failing. Start refactoring in green state and end refactoring in green state (version 8).

Dealing With Violation of Command Query Separation

angry_rock.rb

```
module Game
  class AngryRock
    include Comparable

    WINS = [ %w{rock scissors}, %w{scissors paper}, %w{paper rock}]

    attr_accessor :move
```

```
    def initialize(move)
      @move = move.to_s
    end
    def <=>(other)
      if move == other.move
        0
      elsif WINS.include?([move, other.move])
        1
      elsif WINS.include?([other.move, move])
        -1
      else
        raise ArgumentError, "Something's wrong"
      end
    end
    # Problem : Is this method a command and a query?
    # It is ambiguous because play seems to be a name of a command and
    # it is returning the winning AngryRock object
    # play method that violated Command Query Separation is now gone.
    # This is a query method
    def winner(other)
      if self > other
        self
      elsif other > self
        other
      end
    end
  end

  class Play
    def initialize(first_choice, second_choice)
      @winner = first_choice.winner(second_choice)
    end
    def has_winner?
      !@winner.nil?
    end
    def winning_move
      @winner.move
    end
  end
end
```

The play() method that violated Command Query Separation is now gone. The new winner method is a query method.

Using Domain Specific Term

angry_rock.rb

```
module Game
  class AngryRock
    include Comparable

    WINS = [ %w{rock scissors}, %w{scissors paper}, %w{paper rock}]

    attr_accessor :move

    def initialize(move)
      @move = move.to_s
    end
    def <=>(opponent)
      if move == opponent.move
        0
      elsif WINS.include?([move, opponent.move])
        1
      elsif WINS.include?([opponent.move, move])
        -1
      else
        raise ArgumentError, "Something's wrong"
      end
    end
    def winner(opponent)
      if self > opponent
        self
      elsif opponent > self
        opponent
      end
    end
  end

  class Play
    def initialize(first_choice, second_choice)
      @winner = first_choice.winner(second_choice)
    end
    def has_winner?
      !@winner.nil?
    end
    def winning_move
      @winner.move
    end
```

```
  end
end
```

This version (10) the variable other is renamed to opponent. This reveals the intent of the variable.

Refactoring the Specs

angry_rock_spec.rb

```
require 'spec_helper'

module Game
  describe AngryRock do
   it "should pick paper as the winner over rock" do
     play = Play.new(:paper, :rock)

     play.should have_winner
     play.winning_move.should == "paper"
   end
   it "picks scissors as the winner over paper" do
     play = Play.new(:scissors, :paper)

     play.should have_winner
     play.winning_move.should == "scissors"
   end
   it "picks rock as the winner over scissors " do
     play = Play.new(:rock, :scissors)

     play.should have_winner
     play.winning_move.should == "rock"
   end
   it "results in a tie when the same choice is made by both players" do
     data_driven_spec([:rock, :paper, :scissors]) do |choice|
       play = Play.new(choice, choice)

       play.should_not have_winner
     end
   end
  end
end
```

angry_rock.rb

```ruby
module Game
  class AngryRock
    include Comparable

    WINS = [ %w{rock scissors}, %w{scissors paper}, %w{paper rock}]

    attr_accessor :move

    def initialize(move)
      @move = move.to_s
    end
    def <=>(opponent)
      if move == opponent.move
        0
      elsif WINS.include?([move, opponent.move])
        1
      elsif WINS.include?([opponent.move, move])
        -1
      else
        raise ArgumentError, "Something's wrong"
      end
    end
    def winner(opponent)
      if self > opponent
        self
      elsif opponent > self
        opponent
      end
    end
  end

  class Play
    def initialize(first_choice, second_choice)
      choice_1 = AngryRock.new(first_choice)
      choice_2 = AngryRock.new(second_choice)

      @winner = choice_1.winner(choice_2)
    end
    def has_winner?
      !@winner.nil?
    end
    def winning_move
      @winner.move
    end
  end
end
```

The specs are now simplified.

Handling Illegal Inputs

angry_rock_spec.rb

```
require 'spec_helper'

module Game
  describe AngryRock do
   it "should pick paper as the winner over rock" do
     play = Play.new(:paper, :rock)

     play.should have_winner
     play.winning_move.should == "paper"
   end
   it "picks scissors as the winner over paper" do
     play = Play.new(:scissors, :paper)

     play.should have_winner
     play.winning_move.should == "scissors"
   end
   it "picks rock as the winner over scissors " do
     play = Play.new(:rock, :scissors)

     play.should have_winner
     play.winning_move.should == "rock"
   end
   it "results in a tie when the same choice is made by both players" do
     data_driven_spec([:rock, :paper, :scissors]) do |choice|
       play = Play.new(choice, choice)

       play.should_not have_winner
     end
   end
   it "should raise exception when illegal input is provided" do
     expect do
       play = Play.new(:junk, :hunk)
     end.to raise_error
   end
  end
end
```

This version now has specs for illegal inputs.

angry_rock.rb

```ruby
module Game
  class AngryRock
    include Comparable

    WINS = [ %w{rock scissors}, %w{scissors paper}, %w{paper rock}]

    attr_accessor :move

    def initialize(move)
      @move = move.to_s
    end
    def <=>(opponent)
      if move == opponent.move
        0
      elsif WINS.include?([move, opponent.move])
        1
      elsif WINS.include?([opponent.move, move])
        -1
      else
        raise ArgumentError, "Only rock, paper, scissors are valid choices"
      end
    end
    def winner(opponent)
      if self > opponent
        self
      elsif opponent > self
        opponent
      end
    end
  end

  class Play
    def initialize(first_choice, second_choice)
      choice_1 = AngryRock.new(first_choice)
      choice_2 = AngryRock.new(second_choice)

      @winner = choice_1.winner(choice_2)
    end
    def has_winner?
      !@winner.nil?
    end
    def winning_move
      @winner.move
    end
  end
end
```

This implementation has domain specific error message instead of vague error message that is not helpful during troubleshooting.

Hiding the Implementation

angry_rock.rb

```
module Game
  class Play
    def initialize(first_choice, second_choice)
      choice_1 = Internal::AngryRock.new(first_choice)
      choice_2 = Internal::AngryRock.new(second_choice)

      @winner = choice_1.winner(choice_2)
    end
    def has_winner?
      !@winner.nil?
    end
    def winning_move
      @winner.move
    end
  end

  module Internal # no-rdoc
    # This is implementation details. Not for client use.
    class AngryRock
      include Comparable

      WINS = [ %w{rock scissors}, %w{scissors paper}, %w{paper rock}]

      attr_accessor :move

      def initialize(move)
        @move = move.to_s
      end
      def <=>(opponent)
        if move == opponent.move
          0
        elsif WINS.include?([move, opponent.move])
          1
        elsif WINS.include?([opponent.move, move])
          -1
        else
          raise ArgumentError, "Only rock, paper, scissors are valid choices"
        end
```

```
      end
      def winner(opponent)
        if self > opponent
          self
        elsif opponent > self
          opponent
        end
      end
    end
  end
end
```

angry_rock_spec.rb

```
require 'spec_helper'

module Game
  describe Play do

   it "should pick paper as the winner over rock" do
     play = Play.new(:paper, :rock)

     play.should have_winner
     play.winning_move.should == "paper"
   end
   it "picks scissors as the winner over paper" do
     play = Play.new(:scissors, :paper)

     play.should have_winner
     play.winning_move.should == "scissors"
   end
   it "picks rock as the winner over scissors " do
     play = Play.new(:rock, :scissors)

     play.should have_winner
     play.winning_move.should == "rock"
   end
   it "results in a tie when the same choice is made by both players" do
     data_driven_spec([:rock, :paper, :scissors]) do |choice|
       play = Play.new(choice, choice)

       play.should_not have_winner
     end
   end
   it "should raise exception when illegal input is provided" do
```

```
    expect do
      play = Play.new(:junk, :hunk)
    end.to raise_error
  end
 end
end
```

Concise Solution

play_spec.rb

```
require 'spec_helper'
require 'angryrock/play'

module AngryRock
 describe Play do
  it "should pick paper as the winner over rock" do
    play = Play.new(:paper, :rock)

    play.should have_winner
    play.winning_move.should == :paper
  end
  it "picks scissors as the winner over paper" do
    play = Play.new(:scissors, :paper)

    play.should have_winner
    play.winning_move.should == :scissors
  end
  it "picks rock as the winner over scissors " do
    play = Play.new(:rock, :scissors)

    play.should have_winner
    play.winning_move.should == :rock
  end
  it "results in a tie when the same choice is made by both players" do
    data_driven_spec([:rock, :paper, :scissors]) do |choice|
      play = Play.new(choice, choice)

      play.should_not have_winner
    end
  end
  it "should raise exception when illegal input is provided" do
    expect do
      play = Play.new(:junk, :hunk)
    end.to raise_error
```

```
   end
  end
end
```

play.rb

```
module AngryRock
  class Play
    def initialize(first_choice, second_choice)
      @choice_1 = Internal::AngryRock.new(first_choice)
      @choice_2 = Internal::AngryRock.new(second_choice)

      @winner = @choice_1.winner(@choice_2)
    end
    def has_winner?
      @choice_1.has_winner?(@choice_2)
    end
    def winning_move
      @winner.move
    end
  end

  module Internal # no-rdoc
    # This is implementation details. Not for client use. Don't touch me.
    class AngryRock
      WINS = {rock: :scissors, scissors: :paper, paper: :rock}

      attr_accessor :move

      def initialize(move)
        @move = move
      end
      def has_winner?(opponent)
        self.move != opponent.move
      end
      # fetch will raise exception when the key is not one of the allowed choice
      def winner(opponent)
        if WINS.fetch(self.move)
          self
        else
          opponent
        end
      end
    end
  end
end
```

This concise solution is based on Sinatra Up and Running book example. In this chapter, we saw Rock Paper Scissors Game Engine. It has two solutions:

1. Well Grounded Rubyist based solution refactored to a better design.
2. Sinatra Up and Running based concise solution.

Double Dispatch

Objective

How to use double dispatch to make your code object oriented.

Analysis

Possible combinations = 9

Rock Rock Rock Paper Rock Scissor

Paper Rock Paper Paper Paper Scissor

Scissor Rock Scissor Paper Scissor Scissor

Number of items Rock Paper Scissor

game.rb

```
require_relative 'game_coordinator'

module AngryRock
  class Game
    def initialize(player_one, player_two)
      @player_one = player_one
      @player_two = player_two
    end
    def winner
      coordinator = GameCoordinator.new(@player_one, @player_two)
      coordinator.winner
    end
  end
end
```

game_coordinator.rb

```ruby
require_relative 'paper'
require_relative 'rock'
require_relative 'scissor'

module AngryRock
  class GameCoordinator
    def initialize(player_one, player_two)
      @player_one = player_one
      @player_two = player_two
      @choice_one = player_one.choice
      @choice_two = player_two.choice
    end
    def winner
      result = pick_winner

      winner_name(result)
    end

    private

    def select_winner(receiver, target)
      receiver.beats(target)
    end
    def classify(string)
      Object.const_get(@choice_two.capitalize)
    end
    def winner_name(result)
      if result
        @player_one.name
      else
        @player_two.name
      end
    end
    def pick_winner
      result = false
       if @choice_one == 'scissor'
         result = select_winner(Scissor.new, classify(@choice_two).new)
       else
         result = select_winner(classify(@choice_one).new, classify(@choice_two).new)
       end
       result
    end
  end
end
```

paper.rb

```
class Paper
  def beats(item)
    !item.beatsPaper
  end
  def beatsRock
    true
  end
  def beatsPaper
    false
  end
  def beatsScissor
    false
  end
end
```

rock.rb

```
class Rock
  def beats(item)
    !item.beatsRock
  end
  def beatsRock
    false
  end
  def beatsPaper
    false
  end
  def beatsScissor
    true
  end
end
```

scissor.rb

```
class Scissor
  def beats(item)
    !item.beatsScissor
  end
  def beatsRock
    false
  end
  def beatsPaper
    true
  end
  def beatsScissor
```

```
    false
  end
end
```

player.rb

```
Player = Struct.new(:name, :choice)
```

game_spec.rb

```
require 'spec_helper'

module AngryRock
  describe Game do
    before(:all) do
      @player_one = Player.new
      @player_one.name = "Green_Day"
      @player_two = Player.new
      @player_two.name = "minder"
    end
    it "picks paper as the winner over rock" do
      @player_one.choice = 'paper'
      @player_two.choice = 'rock'

      game = Game.new(@player_one, @player_two)
      game.winner.should == 'Green_Day'
    end
    it "picks scissors as the winner over paper" do
      @player_one.choice = 'scissor'
      @player_two.choice = 'paper'

      game = Game.new(@player_one, @player_two)
      game.winner.should == 'Green_Day'
    end
    it "picks rock as the winner over scissors " do
      @player_one.choice = 'rock'
      @player_two.choice = 'scissor'

      game = Game.new(@player_one, @player_two)
      game.winner.should == 'Green_Day'
    end
    it "picks rock as the winner over scissors. Verify player name. " do
       @player_one.choice = 'scissor'
       @player_two.choice = 'rock'
```

```
      game = Game.new(@player_one, @player_two)
      game.winner.should == 'minder'
    end
  end
end
```

1. Run the specs by : $ rspec spec/angry_rock/game_spec.rb –color –format doc
2. Are we ready to deploy this code to production?
3. All tests pass. Test code is bad. Production code is bad. Can you ship the product ?
4. Refactored the test code. Started in Green state and ended in Green state.
5. We minimized if conditional statements. Moved it to the main partition and kept our application partition clean.
6. The game rules are encapsulated in the Rock, Paper and Scissors class.

Twitter Client

Objectives

- Dealing with third party API.
- Thin adapter layer to insulate your application from external API.
- What abusing mocks looks like
- Brittle tests that break even when the behavior does not change caused by mock abuse
- Integration tests should test the layer that interacts with external API.
- Using too many mocks indicate badly designed API. So called fluent interface is actually a train wreck. Fluent interface is ok for languages like Java where it is the only option.

Running the Specs

Run $ autotest from the root of the project to run the specs.

Version 1

Initial commit to twits.

Version 2

Test hits the live server.

twits_spec.rb

```
require File.expand_path(File.dirname(__FILE__) + '/spec_helper')
require 'user'

describe "Twitter User" do
  context "with a username" do
    before(:each) do
      @user = User.new
      @user.twitter_username = 'logosity'
    end

    it "provides the last five tweets from twitter" do
      tweets = ["race day! http://t.co/nHVyd7s3 #fb",
                "toy to inspire: http://t.co/koMadie2 #fb",
                "just drove the route: http://t.co/nHVyd7s3 #fb",
                "Son is declaring that the Honey Badger is his second favorite animal.",
                "If you want to sail your ship in a different direction."]

      @user.last_five_tweets.should == tweets
    end

  end
end
```

user.rb

```
require 'twitter'

class User
  attr_accessor :twitter_username

  def last_five_tweets
    return Twitter::Search.new.per_page(5).from(@twitter_username).map do |tweet|
      tweet[:text]
    end.to_a
  end
end
```

Version 3

Abuse of mocks. Spec is coupled to the implementation of the method. Spec is brittle. It will break even when the behavior does not change but when the implementation changes. That is likely to happen when you upgrade Twitter gem.

twits_spec.rb

```
require File.expand_path(File.dirname(__FILE__) + '/spec_helper')
require 'user'

describe "Twitter User" do
  context "with a username" do
    before(:each) do
      @user = User.new
      @user.twitter_username = 'logosity'
    end

    it "provides the last five tweets from twitter" do
      tweets = [
        {text: 'tweet1'},
        {text: 'tweet2'},
        {text: 'tweet3'},
        {text: 'tweet4'},
        {text: 'tweet5'},
        ]

      mock_client = mock('client')
      mock_client.should_receive(:per_page).with(5).and_return(mock_client)
      mock_client.should_receive(:from).with('logosity').and_return(tweets)
      Twitter::Search.should_receive(:new).and_return(mock_client)

      @user.last_five_tweets.should == %w{tweet1 tweet2 tweet3 tweet4 tweet5}
    end

  end
end
```

user.rb

```
require 'twitter'

class User
  attr_accessor :twitter_username
```

```
  def last_five_tweets
    return Twitter::Search.new.per_page(5).from(@twitter_username).map do |tweet|
      tweet[:text]
    end.to_a
  end
end
```

Version 4

Fixed the mock abuse. Stub used to disconnect from Twitter client API. Twits must hit the Twitter sandbox in an integration test.

twits_spec.rb

```
require File.expand_path(File.dirname(__FILE__) + '/spec_helper')
require 'user'

describe "Twitter User" do
  context "with a username" do
    before(:each) do
      @user = User.new
      @user.twitter_username = 'logosity'
    end
    # The test now depends on our API fetch_tweets in our Twits Twitter client class
    # This is stable than directly depending on a third party API.
    it "provides the last five tweets from twitter" do
      tweets = %w{tweet1 tweet2 tweet3 tweet4 tweet5}
      Twits.stub(:fetch_tweets).and_return(tweets)
      @user.last_five_tweets.should == %w{tweet1 tweet2 tweet3 tweet4 tweet5}
    end

  end
end
```

twits.rb

```
require 'twitter'

class Twits
  # The following method must hit the Twitter sandbox in the integration test.
  # It is now in Twits (TwitterClient). Ideally nested within a module.
  # This API is a thin wrapper around the actual Twitter API.
  # It insulates the changes in Twitter API from impacting the application.
  def self.fetch_tweets(username)
```

```
    Twitter::Search.new.per_page(5).from(username).map do |tweet|
      tweet[:text]
    end.to_a
  end
end
```

user.rb

```
require 'twits'

class User
  attr_accessor :twitter_username

  def last_five_tweets
     Twits.fetch_tweets(@twitter_username)
  end
end
```

Version 5

Used dependency injection to inject a fake twitter client to break the dependency. Also refactored to move the method from domain model to the service layer object Twits.

twits_spec.rb

```
require File.expand_path(File.dirname(__FILE__) + '/spec_helper')
require 'user'
require 'fake_twitter_client'

describe "Twitter User" do
  context "with a username" do
    before(:each) do
      @user = User.new
      @user.twitter_username = 'logosity'
    end

    # The following is not a good idea due to the headache of keeping the fake
    # object in synch with Twitter API changes. Shows dependency injection.
    it "should provide the last five tweets from twitter" do
      twits = Twits.new(FakeTwitterClient.new)

            expected_tweets = %w{tweet1 tweet2 tweet3 tweet4 tweet5}
      twits.fetch_five(@user.twitter_username).should == expected_tweets
    end
```

```
  end
end
```

twits.rb

```
class Twits

  def initialize(client)
    @client = client
  end
  # The following method must hit the Twitter sandbox in the integration test.
  # It is now in Twits (TwitterClient). Ideally nested within a module.
  # This API is a thin wrapper around the actual Twitter API. It
  # insulates the changes in Twitter API from impacting the application.
  def fetch_five(username)
    @client.per_page(5).from(username).map do |tweet|
      tweet[:text]
    end.to_a
  end
end
```

user.rb

```
require 'twits'

class User
  attr_accessor :twitter_username

end
```

fake_twitter_client.rb

```
class FakeTwitterClient
  def per_page(n)
    self
  end

  def from(username)
    tweets = [{ :text => 'tweet1'},
              { :text => 'tweet2'},
              { :text => 'tweet3'},
              { :text => 'tweet4'},
              { :text => 'tweet5'}]
  end
end
```

Version 6

Deleted unnecessary code.

user.rb

```
require 'twits'

class User
  attr_accessor :twitter_username

end
```

twits.rb

```
class Twits

  def initialize(client)
    @client = client
  end
  # The following method must hit the Twitter sandbox in the integration test.
  # It is now in Twits (TwitterClient). Ideally nested within a module.
  # This API is a thin wrapper around the actual Twitter API.
  # It insulates the changes in Twitter API from impacting the application.
  def fetch_five(username)
    @client.per_page(5).from(username).map do |tweet|
      tweet[:text]
    end
  end
end
```

fake_twitter_client.rb

```
class FakeTwitterClient
  def per_page(n)
    self
  end

  def from(username)
    tweets = [{ :text => 'tweet1'},
              { :text => 'tweet2'},
              { :text => 'tweet3'},
              { :text => 'tweet4'},
              { :text => 'tweet5'}]
  end
end
```

twits_spec.rb

```
require File.expand_path(File.dirname(__FILE__) + '/spec_helper')
require 'user'
require 'fake_twitter_client'

describe "Twitter User" do
  context "with a username" do
    before(:each) do
      @user = User.new
      @user.twitter_username = 'logosity'
    end

    # The following is not a good idea due to the headache of keeping the fake
    # object in synch with Twitter API changes. Shows dependency injection
    it "should provide the last five tweets from twitter" do
      twits = Twits.new(FakeTwitterClient.new)
            expected_tweets = %w{tweet1 tweet2 tweet3 tweet4 tweet5}
      twits.fetch_five(@user.twitter_username).should == expected_tweets
    end
  end
end
```

Discussion

Continuous Testing with Ruby uses mocks in the tests to write the tests for Mongodb. Because we have never used this db before, it shows breaking dependencies by testing against a real service and then replacing those interactions with mocks. This results in lot of mocks in the tests.

Using mocks in this case is improper usage of mocks. Because you cannot drive the design of a third-party API (Mongodb API in this case). There is a better way to breaking the external dependencies.

1. First write learning tests.
2. Then create a thin adapter layer that has well defined interface. This adapter layer will encapsulate the interaction with Mongodb. Now you can mock the thin adapter layer in your code and write integration tests for the adapter tests that will interact with Mongodb.

This prevents the changes in Mongodb API from impacting the domain code. See https://github.com/bparanj/mongodb_specs for example of learning specs.