Essential TDD

Bala Paranj

Section 1 : Basics

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

Fibonacci

Objectives

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

Problem Statement

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

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

Figure 1: Fibonacci Numbers

Solution

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

Algebraic Equation

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

Visual Representation

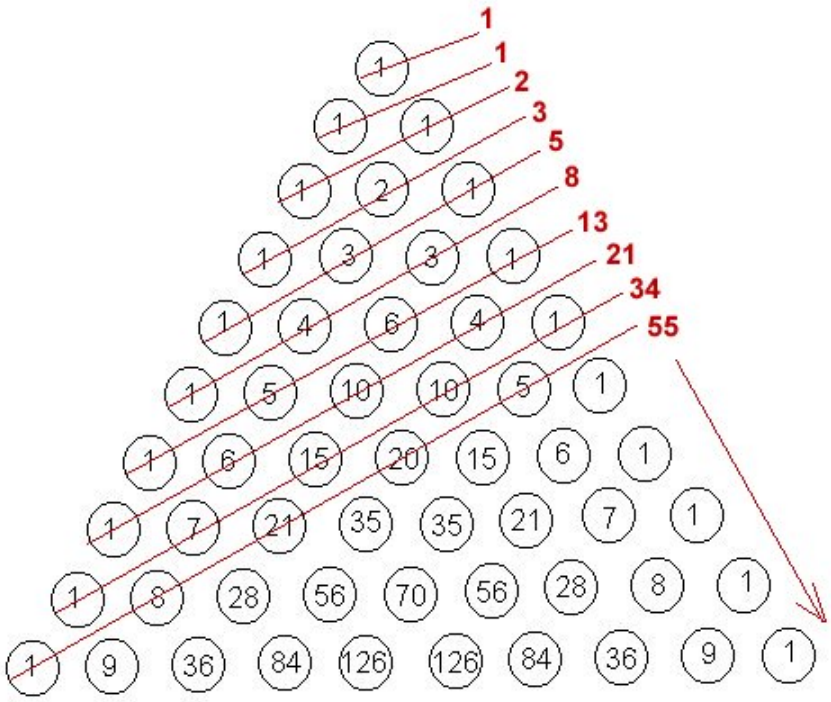

Figure 2: Fibonacci Numbers

Guidelines

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

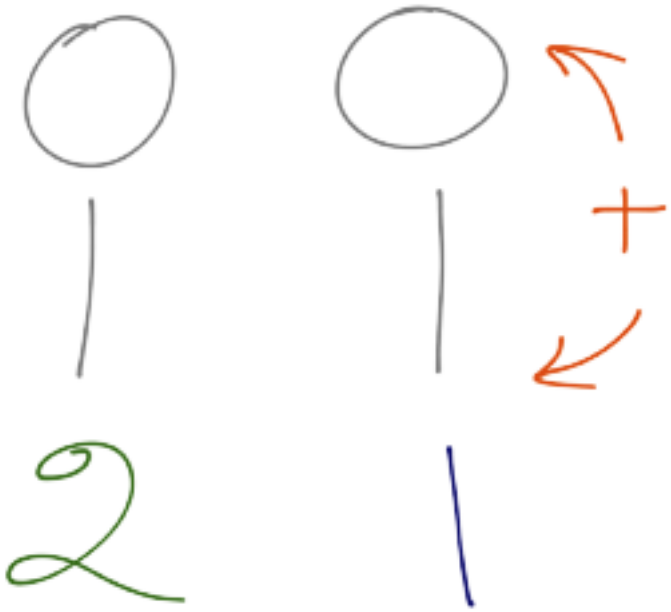

Figure 3: Calculating Fibonacci Numbers

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

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

Version 0

```
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 1

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 2

```
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

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 is 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

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
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)