Essential TDD

Bala Paranj

1. Calculator

- Learn about assertion

2. Canonical Test Structure

- What is Canonical Test Structure
- Practice Canonical Test Structure

3. Code Mutation

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

4. Eliminate Loops

- How to eliminate loops in specs
- Focus on "What" instead of implementation, the "How"

5. Role

- Being minimal when implementing the production code.

6. Fibonacci

- 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.

7. Scanner

- How to use Fakes and Mocks ?
- When to delete a test ?
- Learn about Open Closed Principle and how to apply it

8. Week

- Introduction to Contract tests.
- How to write Contract tests?
- Using Contract tests to explicitly document the behavior of an API for invalid inputs.
- Learn about reliable test. A test that fails when it should.

9. Guess Game

- 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.

- Make methods small, 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 production code ?

10. Uncommenter

- Using fake objects to speed up test

11. Test Spy

- Using Stubs with Test Spy in Ruby

12. Command Query Separation

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

13. Angry Rock

- 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

14. Bowling Game

- Using domain specific term and eliminating implementation details in the spec.
- Focus on the 'What' instead of 'How'. Declarative vs Imperative.
- Fake it till you make it.
- When to delete tests?
- State Verification
- Scoring description and examples were translated to specs.
- BDD style tests read like sentences in a specification.
- Updating the specs as we learn more about the bowling game instead of blindly appending specs to the existing specs.

15. Double Dispatch

- Learn how to use double dispatch to make your code object oriented.

16. Twitter Client

- How to deal with third party API?
- How to use thin adapter layer to insulate domain code from external API?
- What does abusing mocks look like?
- Example of 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.

17. Learning Specs

- Why do we need learning specs?
- How to write learning specs?

18. String Calculator

- Triangulate to solve the problem
- Experiment to learn and explore possible solution
- Refactoring when there is no duplication to write intent revealing code
- Simplifying method signatures

Appendix

- A. RSpec Test Structure
- B. Fibonacci Exercise Answer
- C. Interactive Spec
- D. Gist by Pat Maddox at https://gist.github.com/730609
- E. FAQ
- F. Difficulty in Writing a Test
- G. Side Effect
- H. dev/null in Unix
- I. Stub
- J. Notes from Martin Fowler's article and jMock Home Page
- K. Notes on Mock Objects
- L. Why use mock objects?

- M. A Pattern for Unit Testing
- N. Tautology
- O. Interactive Spec
- P. The Rspec Book
- Q. Direct Input
- R. Indirect Input
- S. Direct Output
- T. Indirect Output
- U. Angry Rock : Possible Solution
- V. Angry Rock : Concise Solution
- W. Double Dispatch : Angry Rock Game Solution

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. The chapters are arranged in a progressively increasing level of complexity. Subsequent chapters build on the concepts already introduced.

If there's something you still don't understand after you're finished, email me at bala.paranj@zepho.com with your question. Make the subject 'ETDD Question'. I'll make sure you get the help you need.

Calculator

Objective

- Learn about assertion

Dicussion

Let's write a simple calculator program driven by test. What statements can you make about the calculator program that is true? How about :

- It should add given two numbers.

Let's write a spec for this statement. Create a file called calculator_spec.rb with the following contents:

```
describe Calculator do
  it "should add given two numbers" do
    calculator = Calculator.new

    result = calculator.add(1,2)

    result.should == 3
  end
end
```

We first create an instance of Calculator class. The second step is invoking the method add() to calculate sum of two numbers. The third step is checking if the result is the same as we expect. In this step, we have converted the statement that is true to an assertion.

According to the dictionary assertion is a confident and forceful statement of fact or belief. If we did not write any test then we would manually check the result for correctness. We automate this manual check by using an assertion. Go to the directory where the spec file resides and run the test like this:

```
$ rspec calculator_spec.rb --color --format documentation
     or
$ rspec calculator_spec.rb --color --format nested
```

This test fails. Define Calculator class at the top of the calculator_spec.rb file with the code shown below:

```
class Calculator
end
```

The error message you get now is different. It is because you have not defined the add method. Add the method to the class :

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

Run the test again. Now the test passes. You can now move the Calculator class to its own file called calculator.rb. Add

```
require_relative 'calculator'
```

to the top of the calculator_spec.rb. Run the test again. It should pass.

Conclusion

In this first excercise we took little baby steps. We wrote code with the intent to change the error message. Initially your error messages are related to setting up the environment. Once you get past that, you can make the test fail for the right reason. Failing for the right reason means, the test will fail to satisfy the requirments instead of syntax mistakes, missing require statements etc.

Exercises

1. Write specs for the following statements:

- It should subtract given two numbers.
- It should multiply given two numbers.
- It should divide given two numbers.

2. Refactor the duplication you see by using let() or before() method.

Refer the rspec documentation for examples on how to use the rspec API. You can search for let and look at the examples on how to use it. It is a good idea to run the examples to learn the API. Then you can incorporate the changes to your specs.

3. Write specs for edge cases such as invalid input, division by 0 etc.
4. Create a .rspec file with the following contents:

```
--color
--format documentation
```

Now you can run the specs without giving it any options like this:

```
rspec calculator_spec.rb
```

What do you see as the output in the terminal?

Canonical Test Structure

Objective

- To practice Canonical Test Structure : Given, When, Then

According to the Dictionary, the term canonical is defined as:

- `Mathematics : relating to a general rule or standard formula.`

In our case, the following the three steps is a standard formula for writing any test.

- Step 1 - Given : Precondition (System is in a known state)
- Step 2 - When : Exercise the System Under Test (SUT)
- Step 3 - Then : Postcondition (Check the outcome is as expected)

Create a file stack_spec.rb with the following contents:

```
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
```

Create stack.rb with a simple stack implementation that can push and pop as follows:

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

Run the specs. You should see them all pass.

Instead of thinking about 'How do I write a test?'. Ask yourself the following questions:

- What is the given condition?
- How do I excercise the system under test?
- How do I verify the outcome?

The answers to these questions will help you write the test.

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
```

The following code identifies the three steps for the stack_spec.rb:

```
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"
end
```

Run the stack_spec.rb. It should pass. This is an example for State Verification. We check the state of the system after we exercise the SUT for correctness.

Exercise

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

Question

1. What if the method does many things that needs to be tested?

Ideally a method should be small and do just one thing. If a method has three steps with different scenarios, then you will write three different specs for each scenario.

2. What if I want to test push and pop in one test?

The structure of the test is Arrange, Act, Assert. There should be only one Arrange, one Act and one Assert per test. In this case you would have multiple of each of these steps. So it does not follow the best practice. Why do we need just one AAA in our test?

Because if you had multiple of each of the steps, you would have the state at the end of the first assertion interact with the state at the end of the second assertion. Ideally we want each test to be isolated. Isolation means that the state is clean in the beginning of each test and it cleans up the state at the end of each test.

Code Mutation

Objective

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

Version 1

Create a ruby_extensions_spec.rb with the following contents:

```
require_relative 'ruby_extensions'

describe 'Ruby extensions' 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
end
```

To make the test pass, create ruby_extensions.rb with the following contents:

```
class Array
  # & operator is used for intersection operation in Array.
  def intersection(another)
    self & another
  end
end
```

Add the second spec for the boundary condition like this:

```
require_relative 'ruby_extensions'

describe 'Array extensions' do
  ...
  it "should return an empty array if there is no common elements to both arrays"
    a = [1,1,3,5]
    b = [7,9]
    result = a.intersection(b)

    result.should == []
  end
end
```

This test passes without failing. The question is how do you know if this test is correct? We don't have a test to test this test. To validate the test, we have to mutate the production code to make it fail for the scenario under test.

Change the ruby_extensions.rb so that only the second spec fails like this:

```
class Array
  def intersection(another)
    return [10] if another.size == 2
    self & another
  end
end
```

Now the second spec breaks with the error:

```
1) Array Array extensions should return an empty array
     if there is no common elements to both arrays
   Failure/Error: result.should == []
     expected: []
          got: [10] (using ==)
```

Delete the short circuiting condition from the ruby_extensions.rb:

```
return [10] if another.size == 2
```

Now both the specs should pass.

Final Version

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 Extensions' 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 an empty array if there is no common elements to both arrays" do
    a = [1,1,3,5]
    b = [7,9]
    result = a.intersection(b)

    result.should == []
  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 "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 behavior of the me
  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 the test is testing the right thing. In this example we saw:

- What to do when the test passes without failing the first time.

- How to open classes that preserves the semantics of the core classes.
- Intention revealing variable and method names.

Exercise

1. Think of edge cases for the ruby_extensions.rb. Write specs for them. When the spec passes without failing, mutate the code to make only the boundary condition spec fail. Then make all the specs pass.

Eliminating Loops

Objective

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

Example

Just quickly read the following code for meszaros gem (https://github.com/bparanj/meszaros.git) to see the utility methods that help to eliminate loops in specs:

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.

The 'What' is like a blue print for a house. Blue print does not get buried under implementation details of a house. Why is the mechanics of 'How' something is implemented.

Let's take a look at an example to see how the code would look like when it mixes the "What" with "How". The code for before section is stolen from Alex Chaffe's presentation: https://github.com/alexch/test-driven

Before

Matrix Test

```
class String
  def vowel?
    %w(a e i o u).include?(self)
  end
end

describe 'Vowel checker' do
  %w(a e i o u).each do |letter|
    it "#{letter} is a vowel" do
      letter.should be_vowel
    end
  end
end
```

This mixes what and how. It is not clear. Since the implementation details buries the intent of the spec. It passes with the message:

```
$ rspec ruby_extensions_spec.rb --color --format doc

Vowel Checker
  a is a vowel
  e is a vowel
  i is a vowel
  o is a vowel
  u is a vowel

Finished in 0.0048 seconds
5 examples, 0 failures
```

After

Data Driven Spec

```
class String
  def vowel?
    %w(a e i o u).include?(self)
  end
end

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

describe 'Vowel Checker' do
  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
end

$rspec ruby_extensions_spec.rb --color --format doc
```

```
Vowel Checker
  a, e, i, o, u are the vowel set

Finished in 0.00358 seconds
1 example, 0 failures
```

This is a specification that focuses only on "What". It separates the "What" from the "How". The "How" is hidden behind a library call data_driven_spec. The doc string is easily understood without running the program inside your head.

Since the spec passed without failing, let's mutate the code like this:

```
class String
  def vowel?
    !(%w(a e i o u).include?(self))
  end
end
```

It now fails with the error message:

```
1) Vowel Checker a, e, i, o, u are the vowel set
    Failure/Error: letter.should be_vowel
      expected vowel? to return true, got false
```

Now revert back the change. The spec should pass.

Exercise

1. Can you think of another way to mutate the vowel? method so that the test fails first?
2. Bonus : Can you use a custom matcher for the vowel check?
3. Can we use the utility method in the Loop class to square or cube all elements in a given array?

Role

Objective

- Being minimal when implementing the production code.

Discussion

We should be able to assign a role to a given user. Create a user_spec.rb file with the following contents:

```
describe User do

end
```

We get the error:

```
uninitialized constant User
```

Add the User class definition to the top of the user_spec.rb as follows:

```
class User

end
```

Run the test again. We are green but there are no examples. Add the first spec:

```
  it 'should be in any role assigned to it'
```

Run the spec again. You see a pending spec in the output:

```
Pending:
  User should be in any role assigned to it
```

Change the spec to the following:

```
it 'should be in any role assigned to it' do
  user = User.new
  user.assign_role("some role")
  user.should be_in_role('some role')
end
```

We get the error:

```
undefined method `assign_role' for User
```

Define the assign_role method in User class like this:

```
def assign_role

end
```

We get the error:

```
wrong number of arguments (1 for 0)
```

Add the argument like this:

```
def assign_role(role)

end
```

We get the error:

```
undefined method `in_role?' for User
```

Add in_role? method to user.rb as follows:

```
def in_role?

end
```

We get the error:

```
wrong number of arguments (1 for 0)
```

Add an argument to in_role? method:

```
def in_role?(role)

end
```

We get the error:

```
expected in_role?("some role") to return true, got nil
```

We are now failing for the right reason. Notice that each small step we took was guided by the failure messages given by running the test. We only did just enough to get pass the current error message. We were lazy in writing the production code. Change the in_role? implementation like this:

```
def in_role?(role)
  true
end
```

The example will now pass. We know that this implementation is bogus. We want the specs to force us to write the logic that can handle assigning roles.

```
it 'should not be in any role that is not assigned to it' do
  user = User.new
  user.should_not be_in_role('admin')
end
```

We get the error:

```
expected in_role?("admin") to return false, got true
```

Change the user.rb as follows:

```
class User
  def assign_role(role)
    @role = role
  end

  def in_role?(role)
    @role == 'role'
  end
end
```

The specs will pass.

Exercises

1. Move user.rb to its own class. Make sure all the specs pass.
2. Implement the feature where a user can be in many roles. Write the test first.
3. Watch BDD_Basics_II.mov

Appendix

RSpec Test Structure

```
describe Movie, "Definition. Make sure Single Responsibility Principle is obeyed.'

end
```

The first argument of the describe block in a spec is name of the class or module under test. It is the subject. It can also be a string. The second is an optional string. It is a good practice to include the second string argument that describes the class and make sure that it does not have 'And', 'Or' or 'But'. If it obeys Single Responsibility Principle that it will not contain those words.

"'ruby specify "[Method Under Test] [Scenario] [Expected Behavior]" do

end "' Same thing can be accomplished by using describe, context and specify methods together. Refer the RSpec book to learn more.

Given When Then

1. Fibonacci Exercise Answer

fibonacci_spec.rb

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

describe Fibonacci do
  it "should return 0 for 0 input" do
    fib = Fibonacci.new
    result = fib.output(0)
```

```
    result.should == 0
  end

  it "should return 1 for 1 input" do
    fib = Fibonacci.new
    result = fib.output(1)
    result.should == 1
  end

  it "should return 1 for 2 input" do
    fib = Fibonacci.new
    result = fib.output(2)
    result.should == 1
  end

  it "should return 2 for 3 input" do
    fib = Fibonacci.new
    result = fib.output(3)
    result.should == 2
  end
end
```

2. Interactive Spec

How to use Interactive Spec gem to experiment with RSpec.

Standalone:

```
1. gem install interactive_spec
2. irspec
3. > (1+1).should == 3
```

Rails:

```
1. Include gem 'interactive_rspec' in Gemfile
2. bundle
3. rails c
3. > irspec
4. > User.new(:name => 'matz').should_not be_valid
5. > irspec 'spec/requests/users_spec.rb'
```

5. Gist by Pat Maddox at https://gist.github.com/730609

```
module Codebreaker
  class Game
    def initialize(output)
      @output = output
    end
    def start
      @output.puts("Welcome to Codebreaker!")
      @output << "You smell bad"
    end
  end
end

module Codebreaker
  describe Game do
    describe "#start" do
      it "sends a welcome message" do
        output = double('output')
        game = Game.new(output)
        output.should_receive(:puts).with('Welcome to Codebreaker!')
        game.start
      end
    end
  end
end
```

This example is from the RSpec Book. The problem here is the Game object has no purpose. It is ignoring the system boundary and is tightly coupled to the implementation. It violates Open Closed Principle.

FAQ

1. cover rspec matcher is not working in ruby 1.8.7. Create a custom matcher called between(lower, upper) as an example.

2. Composing objects occurs in the Game.new(fake_console) step. The mock is basically an interface that plays the role of console.

3. In the refactoring stage, you must look beyond just eliminating duplication. You must apply OO principles and make sure the classes are cohesive and loosely coupled.

4. Why you should not begin refactoring in red state? If you start refactoring in the red state then you will not know which of the changes you made is causing the problem. It becomes difficult to fix the problem.

5. Specs should read like a story with a beginning, middle and an end. Once upon a time... lot of exciting things happen... then they lived happily ever after.

6. How do you know the code is working? A test should fail when the code is broken. It should pass when it is good.

7. Do not tie the test to the data structure. It will lead to brittle test.

Difficulty in Writing a Test

1. How can you express the domain? What should happen when you start a game?

2. What statements can you make about the program that is true?

3. Side Effect

A function or expression modifies some state or has an observable interaction with calling functions or the outside world in addition to returning a value. For example, a function might modify a global or static variable, modify one of its arguments, raise an exception, write data to a display or file, read data, or call other side-effecting functions. In the presence of side effects, a program's behavior depends on history; that is, the order of evaluation matters. Understanding a program with side effects requires knowledge about the context and its possible histories; and is therefore hard to read, understand and debug.

Side effects are the most common way to enable a program to interact with the outside world (people, filesystems, other computers on networks). But the degree to which side effects are used depends on the programming paradigm. Imperative programming is known for its frequent utilization of side effects. In functional programming, side effects are rarely used.

Source: Wikipedia

4. dev/null in Unix

In Unix, /dev/null represents a null device that is a special file. It discards all data written to it and provides no data to anyone that read from it.

Stub

1. In irb:

```
> require 'rspec/mocks/standalone'
> s = stub.as_null_object
```

acts as a UNIX's dev/null equivalent for tests. It ignores any messages. Useful for incidental interactions that is not relevant to what is being tested. It implements the Null Object pattern.

In E-R modeling you have relationships such as 1-n, n-n, 1-1 and so on. In domain modeling you have relationships such as aggregation, composition,

inheritance, delegation etc. Most of these have constructs provided by the language or the framework such as Rails. Example: composed_of in Rails, delegate in Ruby, symbol < for inheritance. The interface relationship for roles has to be explicitly specified in the specs to make the relationship between objects explicit.

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.

Notes on Mock Objects

A Mock Object is a substitute implementation to emulate or instrument other domain code. It should be simpler than the real code, not duplicate

its implementation, and allow you to set up private state to aid in testing. The emphasis in mock implementations is on absolute simplicity, rather than completeness. For example, a mock collection class might always return the same results from an index method, regardless of the actual parameters.

A warning sign of a Mock Object becoming too complex is that it starts calling other Mock Objects – which might mean that the unit test is not sufficiently local. When using Mock Objects, only the unit test and the target domain code are real.

Why use mock objects?

- Deferring Infrastructure Choices
- Lightweight emulation of required complex system state
- On demand simulation of conditions
- Interface Discovery
- Loosely coupled design achieved via dependency injection

A Pattern for Unit Testing

Create instances of Mock Objects

- Set state in the Mock Objects
- Set expectations in the Mock Objects
- Invoke domain code with Mock Objects as parameters
- Verify consistency in the Mock Objects

With this style, the test makes clear what the domain code is expecting from its environment, in effect documenting its preconditions, postconditions, and intended use. All these aspects are defined in executable test code, next to the domain code to which they refer. Sometimes arguing about which objects to verify gives us better insight into a test and, hence, the domain. This style

makes it easy for new readers to understand the unit tests as it reduces the amount of context they have to remember. It is also useful for demonstrating to new programmers how to write effective unit tests.

Testing with Mock Objects improves domain code by preserving encapsulation, reducing global dependencies, and clarifying the interactions between classes.

Tautology

Objective

To illustrate common beginner's mistake of stubbing yourself out.

```
describe "Don't mock yourself out" do
  it "should illustrate tautology" do
    paul = stub(:paul, :age => 20)

    expect(paul.age).to eq(20)
  end
end
```

This test does not test anything. It will always pass.

Reference

Working Effectively with Legacy Code

Interactive Spec

Standalone:

1. gem install interactive_spec
2. irspec
3. (1+1).should == 3

Rails:

1. gem 'interactive_rspec' in Gemfile
2. bundle
3. rails c

```
> irspec
> User.new(:name => 'matz').should_not be_valid
> irspec 'spec/requests/users_spec.rb'
```

The Rspec Book

The Good

1. Good discussion of Double, Mock and Stubs.

The Bad

1. Mocking the ActiveRecord library methods is a bad practice. It is shown with partial mocking example. This leads to brittle tests. Because the test is tightly coupled to the implementation. For instance, when Rails is upgraded the specs using old ActiveRecord calls will fail when the new syntax for the ORM is used. Even though the behavior does not change it breaks the tests that is tightly coupled to ORM syntax.

Direct Input

A test may interact with the SUT directly via its public API or indirectly via its back door. The stimuli injected by the test into the SUT via its public API are direct inputs of the SUT. Direct inputs may consist of method calls to another component or messages sent on a message channel and the arguments or contents.

Indirect Input

When the behavior of the SUT is affected by the values returned by another component whose services it uses, we call those values indirect inputs of the SUT. Indirect inputs may consist of return values of functions and any errors or exceptions raised by the DoC. Testing of the SUT behavior with indirect inputs requires the appropriate control point on the back side of the SUT. We often use a test stub to inject the indirect inputs into the SUT.

Direct Output

A test may interact with the SUT directly via its public API or indirectly via its back door. The responses received by the test from the SUT via its public API are direct outputs of the SUT. Direct outputs may consist of the return values of method calls, updated arguments passed by reference, exceptions raised by the SUT or messages received on a message channel from the SUT.

Indirect Output

When the behavior of the SUT includes actions that cannot be observed through the public API of the SUT but that are seen or experienced by other systems or application components, we call those actions the indirect outputs of the SUT. Indirect outputs may consist of calls to another component, messages sent on a message channel and records inserted into a database or written to a file. Verification of the indirect output behaviors of the SUT requires the use of appropriate observation points on the back side of SUT. Mock objects are often used to implement the observation point by intercepting the indirect outputs of the SUT and comparing them to the expected values.

Source : xUnit Test Patterns: Refactoring Test Code by Gerard Meszaros

Angry Rock : Possible Solution

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?
```

```ruby
  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
```

Angry Rock : 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
```

Double Dispatch : Angry Rock Game Solution

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

```
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).r
      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
```