MI-RUB Testing

Pavel Strnad
pavel.strnad@fel.cvut.cz

Dept. of Computer Science, FEE CTU Prague,
Karlovo nám. 13, 121 35
Praha, Czech Republic

MI-RUB, WS 2011/12

Contents

Unit Testing

Unit Testing

- Focuses on small chunks (units) of code, typically individual methods or lines within methods.
- Layers principle - If there is a bug on a lower layer then it will surely propagate to a higher level!
- So, We need unit testing. It goes hand in hand with Ruby.

Test Driven Development

TDD

Write a test before implementation. You will write a better code!

Testing Example

Roman Numbers Implementation

```
class Roman
MAX_ROMAN = 4999
  def initialize(value)
  .........
      @value = value
  end
  FACTORS = [["m", 1000], ["cm", 900], ["d", 500], ["cd",
      400],...]
  def to_s
    value = @value
    roman = ""
    for code, factor in FACTORS
      count, value = value.divmod(factor)
      roman << code unless count.zero?
    end
    roman
  end
end
```

Roman Numbers Naive Test

```
require 'roman'
r = Roman.new(1)
fail "'i' expected" unless r.to_s == "i"
r = Roman.new(9)
fail "'ix' expected" unless r.to_s == "ix"
```

This will be unmanageable for bigger projects.

The Testing Framework

The Testing Framework Properties

The Ruby testing framework is basically three facilities wrapped into a neat package:

- It gives you a way of expressing individual tests.
- It provides a framework for structuring the tests.
- It gives you flexible ways of invoking the tests.

Ruby provides two main frameworks Test::Unit and Minitest::Unit (from 1.9).

Assertions == Expected Results

Assertions

Rather than have you write series of individual if statements in your tests, the testing framework provides a set of assertions that achieve the same thing.

Assertions

Roman Numbers Unit Test

```
require 'roman'
require 'test/unit'
class TestRoman < Test::Unit::TestCase
    def test_simple
        assert_equal("i", Roman.new(1).to_s)
        assert_equal("ii", Roman.new(2).to_s)
        assert_equal("iii", Roman.new(3).to_s)
        assert_equal("iv", Roman.new(4).to_s)
        assert_equal("ix", Roman.new(9).to_s)
    end
end
```

Roman Numbers Exercise

Exercise 1

Fix the implementation of Roman Numbers to pass the test *test_simple*.

Roman Numbers Test Improved

The Test::Unit framework uses reflection to run methods starting with a word *test*.

```
require 'roman'
require 'test/unit'
class TestRoman < Test::Unit::TestCase
    NUMBERS = [
    [ 1, "i" ], [ 2, "ii" ], [ 3, "iii" ],
    [ 4, "iv"], [ 5, "v" ], [ 9, "ix" ]
    ]
    def test_simple
        NUMBERS.each do |arabic, roman|
            r = Roman.new(arabic)
            assert_equal(roman, r.to_s)
        end
    end
end
```

Asserts

```
assert_raises(RuntimeError) { Roman.new(0) }
refute_nil(user, "User with ID=1 should exist")
assert_equal(roman, r.to_s)
refute_equal(roman, r.to_s)
```

Structuring Tests

- Unit tests are groupped into high-level groupings called test cases.
- The test cases generally contain all the tests relating to a particular facility or feature.
- The classes that represent test cases must be subclasses of Test::Unit::TestCase.

Test methods

setup and *teardown*

If we have a common code which should be run in the beginning and in the end of each test method (e.g. database connection, initialization of resources) we can extract it to setup and teardown methods. These methods act like brackets around each test method.

Running Tests

```
ruby test_roman.rb
ruby test_roman.rb -n test_range
ruby test_roman.rb -n /range/
```

Running Tests

Use meaningful names, and you'll be able to run (for example) all the shopping-cart-related tests by simply running tests with names matching /cart/.

Where to Put Tests

Project Organization

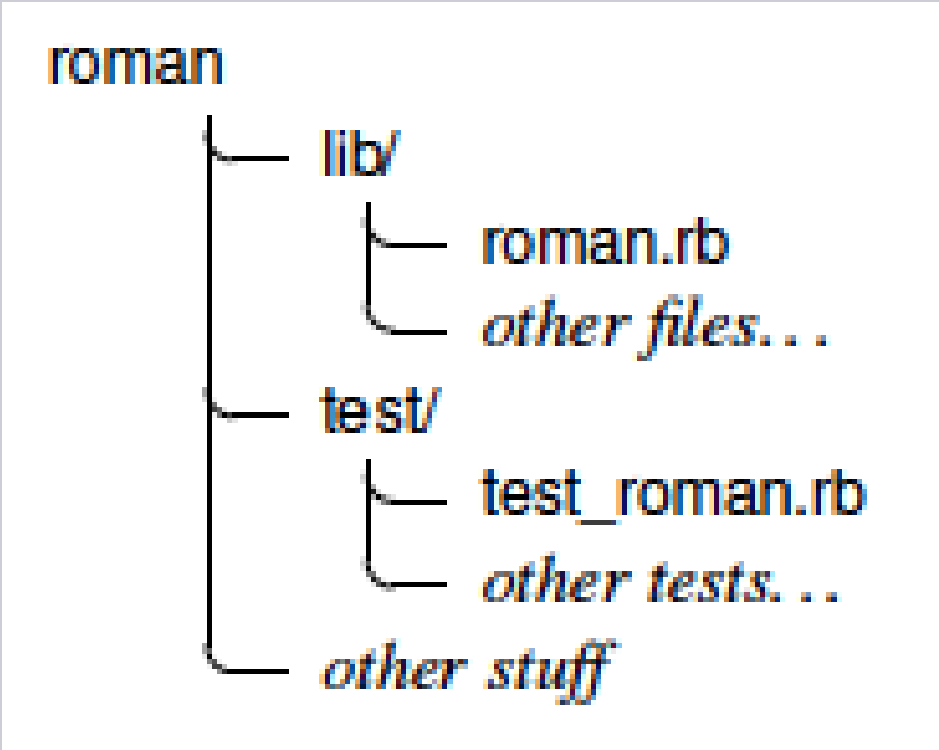

Figure: The above figure is from the Programming Ruby 1.9 book.

Require Problem

```
require 'test/unit'
require '../lib/roman'
class TestRoman < Test::Unit::TestCase
# ...
end
```

Why doesn't this work?

Require Problem

Solution

```
require 'test/unit'
require 'lib/roman'
class TestRoman < Test::Unit::TestCase
# ...
end

ruby -I . test/test_roman.rb
```

Test Suites

You can group *test cases* together into test suites, letting you run them all as a group. Test suites have following advantages:

- You can run individual tests by name.
- You can run all the tests in a file by running that file.
- You can group a number of files into a test suite and run them as a unit.
- You can group test suites into other test suites.

Test Suite Example

```
# file ts_dbaccess.rb
require 'test/unit'
require 'test_connect'
require 'test_query'
require 'test_update'
require 'test_delete'
```

RSpec and Shoulda

RSpec and Shoulda provides different style of testing called **behavior-driven development**. In many ways, this is like testing according to the content of user stories, a common requirements gathering technique in agile methodologies. In these frameworks focus is not on assertions. Instead, you should write expactations. Shoulda offers integration to Test::Unit tests.

RSpec example

```
describe "TennisScorer", "basic scoring" do
    it "should start with a score of 0–0"
    it "should be 15–0 if the server wins a point"
    it "should be 0–15 if the receiver wins a point"
    it "should be 15–15 after they both win a point"
    # ...
end

spec ts_spec.rb
```

This file contains nothing more than a description of an aspect of the tennis scoring class (that we haven't yet written, by the way). We have written four expectations to class *TennisScorer*, and story *basic scoring*.

RSpec example

```
require "tennis_scorer"
describe TennisScorer, "basic scoring" do
    it "should start with a score of 0–0" do
        ts = TennisScorer.new
        ts.score.should == "0–0"
    end
    it "should be 15–0 if the server wins a point"
    it "should be 0–15 if the receiver wins a point"
    it "should be 15–15 after they both win a point"
end
```

Exercise 2

Exercise 2

Implement class TennisScorer to pass all expectations. If you have a time implement Roman numbers :-).