Lab 3: Unit Testing in Ruby

Learning Objectives:

1. Introduction to Unit Testing in Ruby
2. Type of Available Assertions
3. Writing a Test Case
4. Structuring and Organizing Tests

Supporting files:

1. calculatorTest.rb
2. tc_calculatorTest.rb
3. tc_Setup_Teardown.rb

---

1. Introduction to Unit Testing in Ruby

Unit testing is a known way to detect errors early in the development process. There are well known unit testing frameworks for all the popular programming languages. In some cases like Ruby, we can find more than one framework for unit testing. Ruby being a dynamic programming language with no compiler around to check the code, unit testing is more important than in static languages.

The general idea behind unit testing is to write a test method that makes certain assertions about our code. A group of these test methods are bundled up into a test suite and can be run any time the developer wants. The results of these run are gathered in a test result and displayed to the user through some UI.

Ruby offers a few of test frameworks, overviews of these can be found here. Among these we will discuss the Test::Unit framework in the lab, which is available since Ruby 1.8.

The Test::Unit framework provides a few basic functionalities:

- A mechanism to define basic pass/fail tests.
- A way to gather related tests together and run them as a group (suite).
- Allows for running individual tests or whole groups of tests.

---

2. Type of Available Assertions

Assertions are the heart of a testing framework. Think of an assertion as a statement of expected outcome, i.e. "I assert that x should be equal to y". If, when the assertion is executed, it turns out to be correct, we pass the tests. If, however, any of the assertions turn out to be false, an error is propagated with required information so we you can go back and fix our code.

Test::Unit framework provides a rich set of assertions, which are thoroughly documented here. Here is a list of available assertions in Test::Unit framework.

assert(boolean, [message])	True if *boolean*
assert_block(message="assert_block failed.")	Passes if the block yields true.
assert_equal(expected, actual, [message]) assert_not_equal(expected, actual, [message])	True if *expected == actual* True if *expected != actual*

assert_match(pattern, string, [message]) assert_no_match(pattern, string, [message])	True if *string =~ pattern* Passes if *regexp !~ string*
assert_nil(object, [message]) assert_not_nil(object, [message])	True if *object == nil* Passes if *! object .nil*?
assert_in_delta(expected_float, actual_float, delta, [message])	True if *(actual_float - expected_float).abs <= delta*
assert_instance_of(class, object, [message])	True if *object.class == class*
assert_kind_of(class, object, [message])	True if *object.kind_of?(class)*
assert_same(expected, actual, [message]) assert_not_same(expected, actual, [message])	True if *actual.equal?(expected)*. Passes if *! actual .equal? expected*
assert_raise(Exception,...) {block} assert_nothing_raised(Exception,...) {block}	True if the block raises (or doesn't) one of the listed exceptions. Passes if block does not raise an exception.
assert_throws(expected_symbol, [message]) {block} assert_nothing_thrown([message]) {block}	True if the block throws (or doesn't) the expected_symbol. Passes if block does not raise an exception.
assert_respond_to(object, method, [message])	True if the object can respond to the given method.
assert_send(send_array, [message])	True if the method sent to the object with the given arguments return true.
assert_operator(object1, operator, object2, [message])	Compares the two objects with the given operator, passes if *true*
build_message(head, template=nil, *arguments)click to toggle source	Builds a failure message. head is added before the template and arguments replaces the '?'s positionally in the template.
flunk(message="Flunked")	Flunk always fails.

Examples of each of the above assertions can be found here.

---

3. Writing a Test Case

A few steps are involved in writing and executing unit test cases in Ruby. Test::Unit::TestCase binds everything together in a group. If you create your test class as subclass of Test::Unit::TestCase and add your own test methods, it takes care of turning them into tests and wrapping those tests into a suite. Necessary steps for writing test cases:

- Make sure Test::Unit is in your library path.
- Require 'test/unit' in your test script.
- Create a class that subclasses Test::Unit::TestCase.
- Add a method that begins with "test" to your class.

- Make assertions in your test method.

Let us try an example, at first let us create a class that we would like to test:

```
#Supporting File:  calculatorTest.rb

class CalculatorTest

def initialize(num)
   raise unless num.is_a?(Numeric)
   @num1 = num
end

def add(num2)
   @num1 + num2
 end

def subtract(num2)
   @num1 - num2
end

end
```

Now that our class is created, let us write some test cases for it:

```
#Supporting File:  tc_calculatorTest.rb

require_relative 'calculatorTest'
require 'test/unit'

class TestCalculator < Test::Unit::TestCase

  def test_operations
    assert_equal(10, CalculatorTest.new(5).add(5) )
    assert_equal(4, CalculatorTest.new(8).subtract(4) )
 end

 def test_type
    assert_raise(RuntimeError){CalculatorTest.new('number')}
 end
end
```

Output:

```
Loaded suite testCase_calculatorTest
Started
..
Finished in 0.001305 seconds.
```

```
---------------------------------------------------------------------------------------------------
2 tests, 3 assertions, 0 failures, 0 errors, 0 pendings, 0 omissions, 0 notifications
100% passed
---------------------------------------------------------------------------------------------------
1532.57 tests/s, 2298.85 assertions/s
```

In Test Driven Development (TDD) however, we write the test cases first and then we write the code such that it fulfills those test cases.

---

4. Structuring and Organizing Tests

Tests for a particular unit of code are grouped together into a test case, which is a subclass of Test::Unit::TestCase. Related assertions are grouped in tests, as member functions for the test cases whose names start with **test_**. When the test case is executed or required, Test::Unit will iterate through all of the tests (finding all of the member functions which start with test_ using reflection) in the test case, and perform the appropriate tasks.

Test case classes can be gathered together into test suites that are Ruby files that require other test cases:

```
# Example File Name: all_tests.rb
require 'test/unit'
require 'test_case1'
require 'test_case2'
require 'test_case3'
.....
```

On the other hand, individual test cases can be executed to check part of the code as follows:

```
$ ruby -w tc_calculatorTest.rb --name test_operations
```

Output:

```
Loaded suite testCase_calculatorTest
Started
.
Finished in 0.000608 seconds.
---------------------------------------------------------------------------------------------------
1 tests, 2 assertions, 0 failures, 0 errors, 0 pendings, 0 omissions, 0 notifications
100% passed
---------------------------------------------------------------------------------------------------
1644.74 tests/s, 3289.47 assertions/s
```

Setup and Teardown

Test::Unit::TestCase wraps up a collection of test methods together and allows you to easily set up and tear down the same test fixture for each test. There are many cases where a small bit of code needs to be run before and/or after each test. Test::Unit provides the setup and teardown member functions, which are run before and after every test. This is

done by overriding setup and/or teardown, which will be called before and after each test method that is run.
For example:

```
#Supporting File:  tc_Setup_Teardown.rb

require './calculatorTest'
require 'test/unit'

class TestCalculator < Test::Unit::TestCase

def setup
   @num1 = CalculatorTest.new(5)
 end

 def teardown
  ## You can type anything here
 end

def test_operations
    assert_equal(10, @num1.add(5) )
    assert_equal(1, @num1.subtract(4) )
 end

end
```

Note: As mentioned in your lecture slides, since, we are writing contracts we will not be using the testing package is this fashion. We don't need the test runner nor the testing class, we just need the assertions.
