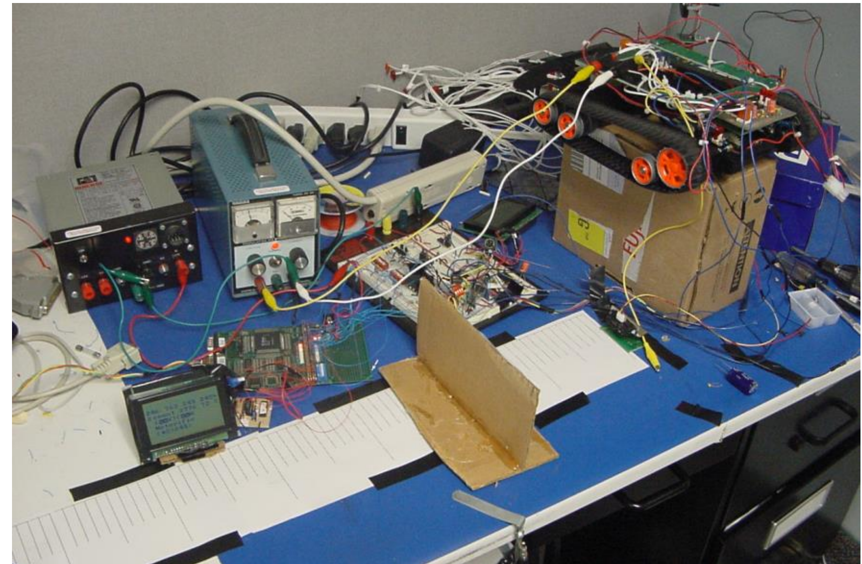
Unit Testing in Ruby

SWEN-250
Personal Software Engineering

Testing, 1 – 2 – 3 – 4, Testing…



What Does a Unit Test Test?

- The term “unit” predates the O-O era.
- Unit – “natural” abstraction unit of an O-O system: class or its instantiated form, object.
- Unit Tests – verify a small chunk of code, typically a path through a method or function.
- Not application level functionality.

Unit Testing Review

- Test a cohesive functional entity:
  - Class
  - Stand alone function or functions
- Verification testing – does the entity do what it's supposed to do.
- Greatly facilitated by unit test frameworks.
  - JUnit for Java
  - NUnit for .NET
  - MiniTest::Test for Ruby

How Do We Unit Test?

- Print Statements (diffs against benchmarks)
- Debuggers – examine variables, observe execution paths.
- Typically done by unit developer.
- Best benefit if running of tests is ***automated***.
- Tests best run in isolation from one another.
- Tests built incrementally as product code is developed.

The Typical Test Cycle

- Develop a suite of test cases
- Create some test fixtures to support the running of each test case.
- Run the test – capture test results.
- Clean-up fixtures, if necessary.
- Report and analyze the test results.

Why is Unit Testing Good?

- Identifies defects early in the development cycle.
- Many small bugs ultimately leads to chaotic system behavior
- Testing affects the design of your code.
- Successful tests breed confidence.
- Testing forces us to read our own code – spend more time reading than writing
- Automated tests support maintainability and extendibility.

Why Don’t We Unit Test?

- “Coding unit tests takes too much time”
- “I’m to busy fixing bugs to write tests”
- “Testing is boring – it stifles my creativity”
- “My code is virtually flawless…”
- “Testing is better done by the testing department”
- “We’ll go back and write unit tests after we get the code working”

Basic xUnit Components

- Create a test class that extends class Test
- Create a testxxx() method for each individual test to be run.
- Create a test fixture – resources needed to support the running of the test.
- Write the test, collect interesting test behavior
- Tear down the fixture (if needed)
- Run the tests.

Key xUnit Concepts

- assert -
  - assertEquals(expected, actual) – also NotEquals
  - assertNull(actual result) – also NotNull
  - assertTrue(actual result) - also False
- failures –
  - Exceptions raised by asserts (expected)
- errors –
  - Ruby runtime exceptions (not expected)

Unit Testing in Ruby

- MiniTest::Test
  - All unit test classes inherit from this class
  - Example: class MyClass < MiniTest::Test
  - setup / teardown
  - test* methods run in random order
- Assertions (change **assert** to **refute** for negative)
  - assert(*boolean*, [*message*])
  - assert_equal(*exp*, *act*, [*message*])
  - assert_raises(*Exception*) *block*
  - assert_nil(*obj*, [*message*])
  - Full list in *http://ruby-doc.org/stdlib-2.0.0/libdoc/minitest/rdoc/MiniTest/Assertions.html*

Queue: (queue.rb)

```
class Queue
  # Exception class for taking values from an empty queue.
  class Empty < StandardError
    def initialize
      super("Empty queue")
    end
  end

  # Initialization
  def initialize
    @contents = Array.new
    self
  end

  # Queue is empty if its size is zero
  def empty?
    size == 0
  end

  # Queue size - number of elements
  def size
    @contents.size
  end
```

Queue: (queue.rb)

```
  # Add a value to the tail of the queue
  def tail= value
    @contents[@contents.size] =  value
    value
  end

  # Return the first element in the queue without removing it
  def peek
    raise Empty if empty?
    @contents[0]
  end

  # Return and remove the first queue element
  def head
    value = peek
    @contents.delete_at(0)
    value
  end
end
```

TestQueue: (test_queue.rb)

```
require 'minitest/autorun'
require_relative 'queue'

class TestQueue < MiniTest::Test
  def setup
    @tq = Queue.new
  end

  # Check proper empty queue behavior
  def test_new_queue
    assert( @tq.size == 0, "New queue size not zero" )
    assert( @tq.empty?, "New queue not empty" )
    assert_raises(Queue::Empty) { @tq.peek }
    assert_raises(Queue::Empty) { @tq.head }
  end
```

TestQueue: (test_queue.rb)

```
# Check proper FIFO behavior. Must end with an empty queue.
def test_fifo_check
  test_values = %w{ A B C } # init an array of three values
  test_values.each { |v| @tq.tail = v } # add to the queue

  size = @tq.size # expect 3 for the queue size
  tvlen = test_values.length
  assert( size == tvlen,
           "#{tvlen} element queue gives size of #{size}" )
  refute( @tq.empty?, "Non-empty queue reports empty" )

  #Iterate through the array and remove each one entry
  test_values.each do |v|
    qv = nil #declare variable to pass between assertions
    qv = @tq.peek # no exception if code is correct
    assert_equal(v, qv, '@tq.peek:')
    qv = @tq.head
    assert_equal(v, qv, '@tq.head:')
  end

  assert_raises(Queue::Empty) { @tq.peek } #empty now
  assert_raises(Queue::Empty) { @tq.head }
end
end
```