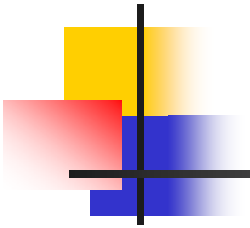
Implementing Munit for Smallworld

Mark Field

FIELD CONSULTING & SERVICES, INC
DOING THE IMPOSSIBLE
FCSI



Overview

- Very Brief Overview of Unit Testing
- What is Munit?
- Installing Munit
- Writing Munit Tests
- Running Munit
- Beyond Unit Testing

Unit Testing

- You know you should!
- Many books and papers on why and how.
- “We write tests to interrogate, taunt, bully, punish, and otherwise kick the crap out of our application code until it consistently gives us the correct answer. Think of it as “tough love”. ;) “ -http://manuals.rubyonrails.com/

Unit Testing Basics

- Unit Testing: in unit testing called components (or communicating components) are replaced with stubs, simulators, or trusted components. … The unit is tested in isolation.
- Component Testing: the same as unit testing except that all stubs and simulators are replaced with the real thing.
- Integration Testing: Tests general sense of data flow and is not restricted to just formal subroutine calls and returns -- for example, passage of data through global data structures and/or the use of pointers.

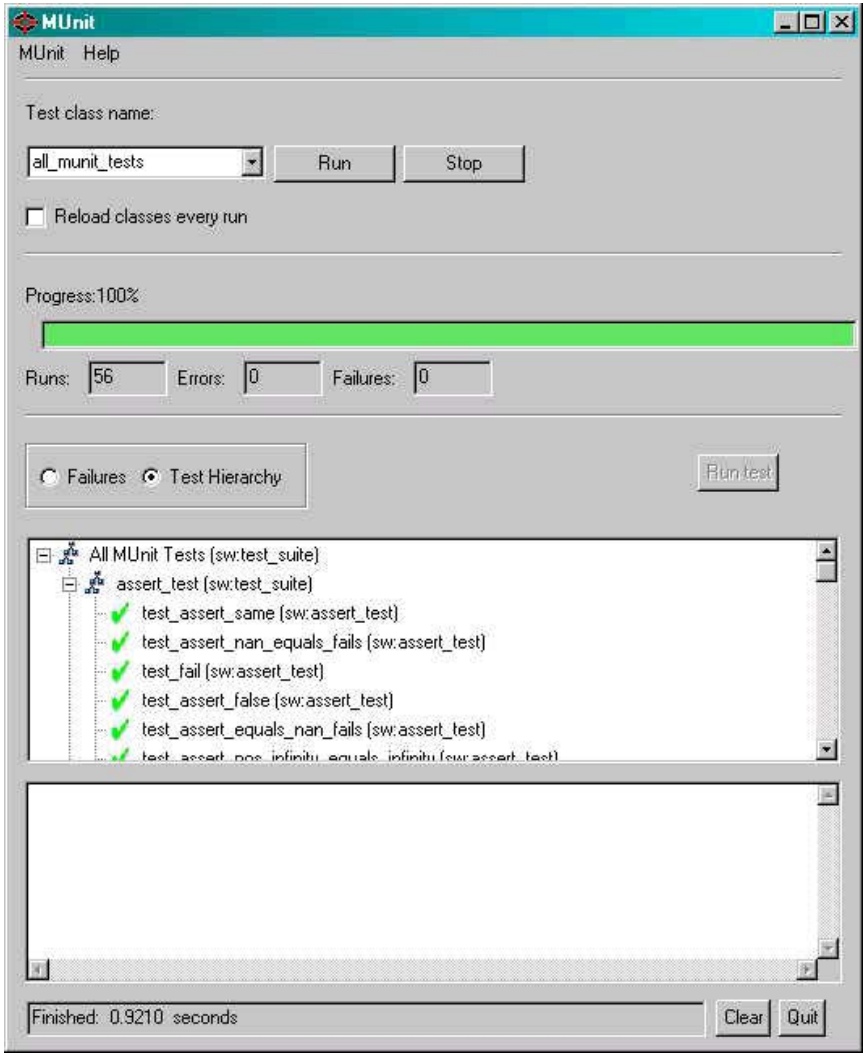
Source: Boris Beizer

Some Benefits of Unit Testing

- Identify code problems sooner.
- Verify changes very much faster!
  - My experiences 10X faster then hand testing.
- Ensure code coverage
- More stable changes
- Less complicated debugging.
  - Concentrate on the fun stuff!

What is MUnit?

- Download from www.janvorel.com?
- MUnit is derived from JUnit
- Features include
  - Assertions
  - Text Fixtures
  - Test Suites
  - Test Runners
- Possible version from GE



Installing MUnit

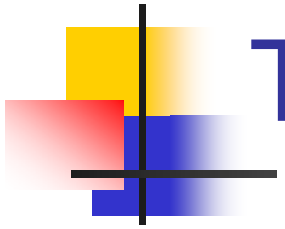
- Unzip the downloaded file to a product directory
- sw_module_manager.load_module_definition(“path/to/munit”)
- sw_module_manager. load_module(:munit)

MUnit Test Code (Simple)

- Exemplar inheriting from :test_case
- Create test methods with “test” included in method name
  - Automatically searches for “test” in method names within test_case exemplar
  - Use assert_xxxx() to do the test
- *test_case*.set_up() & *test_case*.tear_down()
  - Called each time an individual test is run

Results

- Warnings
  - Are not captured by default
  - Can suspend testing
- Errors
  - Code conditions that are generated (eg. index_out_of_bounds)
- Failures
  - Code ran, but wrong results
  - Assertion test the results of the code.
    - assert_equals()
    - assert_equals_within_delta()
    - assert_false()
    - assert_not_equals()
    - assert_not_same()
    - assert_not_unset()
    - assert_same()
    - assert_true()
    - assert_unset()
    - assert_true()
    - fail()



Test Suites

- More like component/integration type testing.
- *test_setup*.set_up() and *test_setup*.tear_down()
  - Ran once per test suite
- The individual test_cases run their own set_up() and tear_down() methods
- *test_setup*.suite() identifies the tests to be run and order to run them.
  - _local suite << test_suite.new()
  - suite.add_test(tc_fcsi_rw_ds.new(:test_1_ds_creation_basic))
  - suite.add_test(tc_fcsi_rw_ds.new(:test_2_ds_creation_soms_gui))
  - suite.add_test(tc_fcsi_rw_ds.new(:test_3_ds_creation_obj_ctrl))
  - _return _self.new(suite)
  - Must restart test_runner_dialog when modified!
- *test_case* methods with “test_” will be run when *test_case* exemplar is selected in test runner.

Gotchas

- Do you need access to private slots or methods?
  - Is this truly unit testing?
  - Rewrite code to make them public.
  - Use sys!perform(), sys!slot(), sys!perform_iter()
- Traceback not what you expected?
  - There are some cases where condition occurs inside a thread.
  - Traceback stops at test.
  - Need to capture the condition in thread, and set a global with the condition and then test for the global
- Test_case examplar is listed in the test exemplars in test runner dialog even if they are part of a test_setup suite.

Munit Examples

- Examplar

```
def_slotted_exemplar(:simple_test,{
   { :value1 , _unset },
   { :value2 , _unset }},
{:test_case})
```

- setup() method

```
_private _method simple_test.set_up()
   .value1 << 2.0
   .value2 << 3.0
_endmethod
```

- test_suite() method

```
method simple_test.suite()
   # Example how to build suite manually
   # suite << test_suite.new()
   # suite.add_test( simple_test.new( "test_add" ))
   # suite.add_test( simple_test.new( "test_div" ))
   # dynamic way / default
   >> test_suite.new( simple_test )
_endmethod
```

Munit Examples (cont.)

- Example test methods

```
_method simple_test.test_add()
   result << .value1 + .value2
   _self.assert_true( result = 6.0 , "add test failed" )
_endmethod
$
_method simple_test.test_div()
   result << .value1 _div .value2
   _self.fail("It should failed before this line")
_endmethod
$
```

Advanced Munit Example

- Test Case

```
_method
   tc_fcsi_rw_dataset.test_3_dataset_creation_obj_ctrl()
   # Get object control
   oc << .swaf.plugin(:object_control)
   ocf << oc.sys!slot(:embedded_object_control)
   opi << ocf.plugin(:ogres_oc)
   opi.action(:add_dataset).execute_action()
   gui << opi.ogres_items[:soms_gui]
   gui.fcsi_items[:dn].value << "obj_ctrl"
   # Get ok buton.
   _local btn <<
   gui.fcsi_items[:df].sys!slot(:children)[1].items[4]
   btn.doit()
_endmethod
```

- Test Suite

```
_method ts_fcsi_rw_dataset.suite()
   _local suite << test_suite.new()
   suite.add_test(tc_fcsi_rw_ds.new(:test_1_create_basic))
   suite.add_test(tc_fcsi_rw_ds.new(:test_2_create_gui))
_endmethod
```

How Many Tests?

- Depends on how much you want to test?
  - Typically means how much money & time in the budget & schedule.
- One test exemplar per Magik file
  - Do not need to test everything in the file
  - Some methods may call other methods so testing sub methods independently would be redundant.
  - Should test as many situations as possible.

FCSI Modifications

- Added: test_setup.test_count()
  - Wasn't in the release and was required
- Added: test_setup.tests()
  - Wasn't in the release and was required
- Created: fail_warn() assertion
  - Allows failure to be raised, but will continue

Event Recording and Playback

- Needed to develop a more system level/integration testing mechanism
- Needed to automate testing of wizards
  - Including GIS Map selection & function keys
- Developed code to intercept and record mouse and keyboard events.
- The position and size of the frame can be different between recording and playback.

Testing Wizards/Tools with MUnit

- The event playback can be integrated into Munit framework to support the testing of wizards
- Issues with event testing
  - Must select wizard/tool you want to start interaction
  - Must be in the correct mode (geometry, trail, etc.)
  - Must have handles to actions/items of wizard or tools and any subsequent dialogs
  - Fill out the items and trigger the actions in the correct order
  - Replay the events in the correct order

Demonstrations

- Test basic GUI interaction
- Event Replay of geometry manipulations

Questions/Comments

- Mark Field
- mark@field-csi.com
- 407-275-9351