System Level Integration and Test
Leveraging Software Unit Testing Techniques

Ryan J. Melton
Ball Aerospace & Technologies Corp.
Boulder, CO

ABSTRACT

Ever try to decipher or debug a huge automated test procedure? Integration and Test (I&T) procedures have historically been organized as a single large file that might include a rough menu system. Software unit testing frameworks solve this problem by breaking down testing into small individual methods that are designed to be easily understood with each method testing a minimal set of functionality. Using these philosophies, test procedures can be broken down into Test Suites (high level testing such as an entire environmental test), Test Groups (testing often devoted to a specific subsystem such as a mechanism), and Test Cases (a specific test of a feature or requirement of the given subsystem). Adding on to this organization with features such as automated test report generation, standardized meta data collection (unit serial number, operator name, etc), and loop testing (executing the same test repeatedly to wring out timing and other issues), produces an extremely powerful system level integration and test solution. This paper discusses realizing these benefits using COSMOS, Ball Aerospace's recently open sourced ground test and operations system which brings the best features of software unit testing frameworks to the world of I&T.

KEY WORDS: I&T, System Level Test, Unit Test, Ball Aerospace COSMOS

INTRODUCTION

This paper begins by discussing the antipatterns occurring in system level test design, then discusses the strategies and benefits currently being exploited with software unit level testing, and finishes by providing a solution to receive the same benefits during system level test.

ANTIPATTERNS IN CURRENT SYSTEM LEVEL TESTING SYSTEMS

Large Monolithic Test Scripts – Satellite level test scripts that run linearly from start to finish often exceed 100,000 lines in length. Printed out at 50 lines per page, a 100,000 line script would produce a 2000 page novel. The cost to adequately review such a script is astronomical, and therefore existing test scripts are left with significant defects.

Unclear Results - Most automated testing leaves important test results intermingled in large output file with other irrelevant messages. Manually wading through these logs to produce a final test report is very time consuming, costly and error prone.

Lack of Automation – Many scripts claim to be automated test procedures, but are littered with required user input. Requiring user input immediately prevents a test procedure from being truly automated and thus unable to be run repeatedly to ring out difficult to find problems.

FUNDAMENTALS AND BENEFITS OF SOFTWARE UNIT TESTING

Software unit testing is the process where software developers verify that the functions that make up the programs they write are working as expected. It is a lower level of test than system level test and developers have full access to the inner workings of their code (white box testing). Despite these differences, system level test can benefit from all of the effort that has been put into optimizing software unit testing for the millions of software engineers around the world. The following best practices from software unit testing can be applied directly to system level test.

Short Test Cases – Good software unit tests focus on producing a large number of small test cases rather than a small number of large tests cases. A common goal is to have each test case verify one and only one expected behavior of the unit under test. This creates test cases that are easy to understand individually, run quickly, and allow for clear regression tests that can identify if any specific behavior has changed. Short test cases can also be used to allow the test procedure to act as the overall specification for the behavior of the system.

Test Case Independence – A properly written software unit test provides test case independence which means that each test case can be run by itself, and that individual test cases can be run in any order. Assuring that each test case can run individually allows for quickly retesting any given feature. Some unit testing frameworks even provide a feature that runs test cases in a random order to make sure that they are truly independent and that no test case leaves unexpected side effects behind. Test case independence is often enabled by providing test setup and teardown methods that ensure that the system is in the proper state before starting the test (setup) and is back to the expected default state after the test (teardown).

Binary Results – Unit testing results are very clear (PASS/FAIL) with well-defined expectations. This enables automated test report generation and makes it very clear if and where anything has gone wrong.

Automation – Well-written software unit tests are completely automated. The primary benefit being that the tests can be run at any time without user interaction. Software unit testing often pushes this benefit to the limit using something called continuous integration. Continuous integration automatically kicks off unit testing whenever a change is made to the software.

APPLYING UNIT TESTING TECHNIQUES TO SYSTEM LEVEL TEST

There is every reason why the above characteristics of software unit testing should be applied to system level integration and test. System level testing can be broken down into short test cases that verify one specific feature of the system such as moving a mechanism or measuring a temperature. Test cases can be written independently such that they can be run in any order or by themselves. Test cases can produce binary results that are placed directly into an automated test report and they can often be fully automated such that no user interaction is required. These characteristics drove the implementation of the Test Runner tool and framework that is part of Ball Aerospace COSMOS, a suite of tools to test and operate embedded systems.

SYSTEM LEVEL TESTING USING BALL AEROSPACE COSMOS TEST RUNNER

Test Suites, Groups, and Cases – COSMOS Test Runner breaks testing down into three categories: Test Suites, Test Groups, and Test Cases. Test Suites are the highest level and generally represent the overall test procedure, e.g. Formal Qualification Test (FQT) or Thermal Vacuum Test. Test Groups are a collection of test cases around a topic, e.g. a Mechanism Test group or an Image Processing test group. Finally, Test Cases test a specific behavior or requirement. Test Runner allows the user to run an entire Test Suite, an individual Test Group, an individual Test Case, or a custom subset of Test Groups and Cases.

Automated Test Report Generation - Test Runner creates an automated test report every time any Test Suite, Test Group, or Test Case is run. The report contains the start and completion time of each test case, the result of each test case (PASS/FAIL/SKIP), specific details on any failures that occurred, the settings Test Runner was started with, and an overall summary of the total test time, total passed test cases, and total failed test cases. The generated report is customizable and arbitrary text can be written to the report from within a test case. Below is an example test report:

```
--- Test Report ---

Files:
Report Filename:
C:/COSMOS/Fqt/outputs/logs/2015_08_10_12_08_53_testrunner_results.txt

Detailed Test Output Logged to:
C:/COSMOS/Fqt/outputs/logs/2015_08_10_12_08_52_sr_FqtTestSuite_messages.txt

Metadata:
OPERATOR_NAME = Ryan Melton
SERIAL_NUMBER = 3

Settings:
Pause on Error = true
Continue Test Case after Error = true
Abort Testing after Error = false
Manual = true
Loop Testing = false
Break Loop after Error = false

Results:
2015/08/10 12:08:53.947: Executing FqtTestSuite
2015/08/10 12:08:55.975: MechanismTestGroup:setup:PASS
2015/08/10 12:09:00.077: MechanismTestGroup:test_mechanism_homing:PASS
  Verifies requirement MECH-1
2015/08/10 12:09:01.021: MechanismTestGroup:test_clockwise_motion:PASS
  Verifies requirement MECH-2
2015/08/10 12:09:02.724: MechanismTestGroup:teardown:PASS
2015/08/10 12:09:03.077: TemperatureTestGroup:test_sensor_1:PASS
  Verifies requirement TEMP-1
2015/08/10 12:09:04.021: TemperatureTestGroup:test_sensor_2:PASS
  Verifies requirement TEMP-2
```

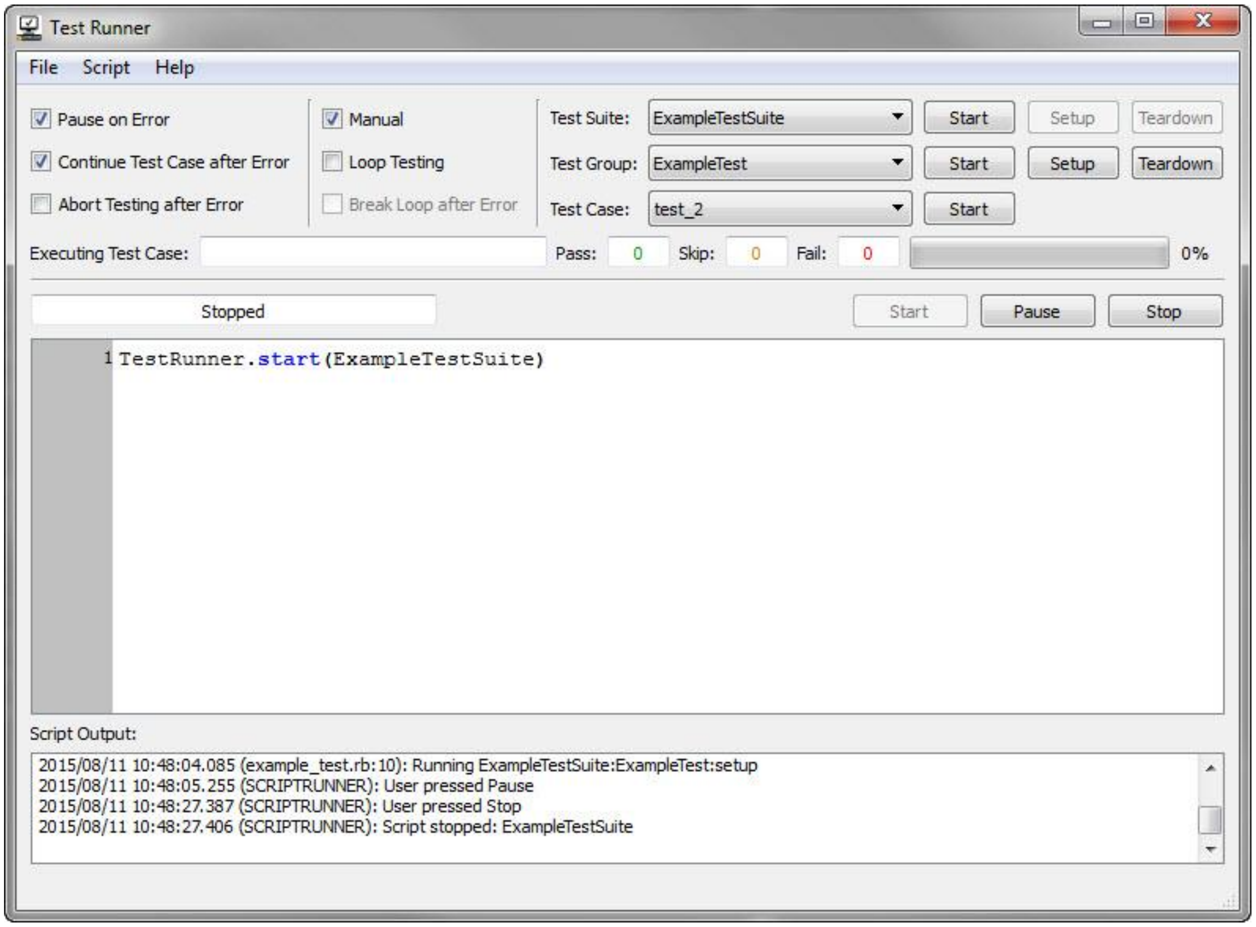
```
2015/08/10 12:09:05.781: Completed FqtTestSuite

--- Test Summary ---

Run Time : 11.87 seconds
Total Tests : 6
Pass : 6
Skip : 0
Fail : 0
```

Variable User Interactivity – Variable user interactivitly allows scripts to be written to support both users in the loop and a fully automated test. Test Runner enables this by allowing users to select whether or not to execute manual steps before they start a procedure. It also provides a syntax to wrap sections of a test script in “if $manual” that allows those sections to only be run if manual testing is desired. This allows for formal test runs to still enable the manual steps, but quick checkouts or automated regressions to run fully autonomously. Additionally, running unattended testing or loop testing usually requires disabling manual steps.

Loop Testing – Timing issues or race conditions are one of the most common types of problems that occur during formal test runs. For example, during a dry run a test completes successfully but during the formal run something takes 0.1 seconds longer and the test fails. A great way to avoid these types of failures is using something called loop testing. Loop testing allows the user to run specific test cases, groups, or entire suites repeatedly over and over without end. Options allow the tests to record any failures that occur and try again, or to stop the procedure if anything goes wrong. Performing loop testing on your test procedures before the formal test can wring out timing problems that cause costly anomaly investigations.



[✓] Pause on Error
[✓] Continue Test Case after Error
[ ] Abort Testing after Error
[✓] Manual
[ ] Loop Testing
[ ] Break Loop after Error

Test Suite: ExampleTestSuite
Test Group: ExampleTest
Test Case: test_2

Executing Test Case:
Pass: 0 Skip: 0 Fail: 0 0%

Stopped

```
TestRunner.start(ExampleTestSuite)
```

```
2015/08/11 10:48:04.085 (example_test.rb:10): Running ExampleTestSuite:ExampleTest:setup
2015/08/11 10:48:05.255 (SCRIPTRUNNER): User pressed Pause
2015/08/11 10:48:27.387 (SCRIPTRUNNER): User pressed Stop
2015/08/11 10:48:27.406 (SCRIPTRUNNER): Script stopped: ExampleTestSuite
```

EXAMPLE TEST SUITE, GROUP, and CASES

Below is an example test suite containing two test groups and several test cases that provides a great example for how system level tests can be organized using Test Runner. The goal is to break the overall test procedure down into pieces which are as small as possible. Each test case reports its PASS/FAIL status and other optional information such as the requirements it is verifying as demonstrated below.

```
class MechanismTestGroup < Cosmos::Test
  def setup
    # One-time setup required by the group
    # Might do something like power on the mechanism
    # that would be needed by all test cases
  end

  def test_mechanism_homing
    Cosmos::Test.puts "Verifies requirement MECH-1"
    # …
  end

  def test_clockwise_motion
    Cosmos::Test.puts "Verifies requirement MECH-2"
    # …
  end

  def teardown
    # One-time teardown required by the group
    # Might do something like power off the mechanism
    # that would be needed by all test cases
  end
end

class TemperatureTestGroup < Cosmos::Test
  def test_sensor_1
    Cosmos::Test.puts "Verifies requirement TEMP-1"
    # …
  end

  def test_sensor_2
    Cosmos::Test.puts "Verifies requirement TEMP-2"
    # …
  end
end

# This is an example Test Suite that runs all of the cases within
# the two test groups above
# Suites can be made that include entire test groups as below or
# individual test cases
class FormalQualificationTestSuite < Cosmos::TestSuite
  def initialize
    super()
    add_test('MechanismTestGroup')
    add_test('TemperatureTestGroup')
  end
end
```

SUMMARY

Over the years there have been numerous advances in software unit testing techniques and frameworks and relatively little in the realm of system level testing. By taking the lessons learned from unit testing, particularly by breaking down tests into small cases, fully automating procedures, generating automated test reports, and designing for test case independence, system level testing can move into the 21st century and provide a much higher quality of test. The Open Source Ball Aerospace COSMOS test system has a ready to use system level test framework within its Test Runner tool that can make these changes a reality.

REFERENCES

Beckman, Nels Eric, "Unit Testing: Philosophy and Tools," *Institute of Software Research* (February 1, 2007). Retrieved from http://www.cs.cmu.edu/~aldrich/courses/654-sp07/slides/unit_testing_lecture.pdf, August 11, 2015

Hunt, Andrew and Thomas, David, "Pragmatic Unit Testing In Java with JUnit", *The Pragmatic Programmers*

BIOGRAPHIES

Ryan Melton has been working at Ball Aerospace in Boulder Colorado and helping to develop, integrate, and test aerospace products for the past 14 years. Notable programs on which Ryan has worked include Kepler, GPM, and CALIPSO. Ryan is very active in the open source community with the recent release of Ball Aerospace COSMOS as well as for many years working with the Ruby bindings to the Qt GUI framework, qtbindings. He holds a Bachelor's of Science in Computer Engineering from Purdue University and an MBA from Regis University. Please address any questions on this paper to rmelton@ball.com.