CSE306 Software Quality in Practice

Dr. Carl Alphonse
aphonce@buffalo.edu
343 Davis Hall
Recall the rules

1. Understand the requirements
2. Make it fail
3. Simplify the test case
4. Read the right error message
5. Check the plug
6. Separate fact from fiction
7. Divide and conquer
8. Match the tool to the bug
9. One change at a time
10. Keep an audit trail
11. Get a fresh view
12. If you didn't fix it, it ain't fixed
13. Cover your bug fix with a regression test
Recall the rules

1. Understand the requirements
2. Make it fail
3. Simplify the test case
4. Read the right error message
5. Check the plug
6. Separate fact from fiction
7. Divide and conquer
8. Match the tool to the bug
9. One change at a time
10. Keep an audit trail
11. Get a fresh view
12. If you didn’t fix it, it ain’t fixed
13. Cover your bug fix with a regression test
Unit testing frameworks

- uniform way of expressing tests
- manage tests through suites
- automate testing process
Unit Testing frameworks

https://en.wikipedia.org/wiki/List_of_unit_testing_frameworks

CUnit - C
JUnit - Java
Mocha - JavaScript
pytest - Python
ScUnit - Scala
VUnit - Verilog/VHDL
among many, many others
We'll use CUnit

http://cunit.sourceforge.net

http://cunit.sourceforge.net/doc
CUnit is organized like a conventional unit testing framework:

```
  Test Registry
  |     |     |
  ----|-----|-----|
  |    |    |
  |Suite '1'    |Suite 'N'|
  |     |     |
  ----|-----|-----|
  |    |    |
  |Test '1l'    |Test '1M'|
  |    |    |
  |    |    |
  |Test 'Nl'    |Test 'NM'|
```

A typical sequence of steps for using the CUnit framework is:

1. Write functions for tests (and suite init/cleanup if necessary).
2. Initialize the test registry - `CU_initialize_registry()`
3. Add suites to the test registry - `CU_add_suite()`
4. Add tests to the suites - `CU_add_test()`
5. Run tests using an appropriate interface, e.g. `CU_console_run_tests`
6. Cleanup the test registry - `CU_cleanup_registry`
a test

- a void -> void function
- test functions must be part of a test suite, which must in turn be registered with registry before the suite will be run
Assertions
(the most common ones)

CU_ASSERT_TRUE(x)
CU_ASSERT_EQUAL(x,y)
CU_ASSERT_PTR_EQUAL(x,y)
CU_ASSERT_PTR_NULL(x,y)
CU_ASSERT_STRING_EQUAL(x,y)
CU_ASSERT_DOUBLE_EQUAL(x,y,\varepsilon)
CU_PASS(message)

CU_ASSERT_FALSE(x)
CU_ASSERT_NOT_EQUAL(x,y)
CU_ASSERT_PTR_NOT_EQUAL(x,y)
CU_ASSERT_PTR_NOT_NULL(x,y)
CU_ASSERT_STRING_NOT_EQUAL(x,y)
CU_ASSERT_DOUBLE_NOT_EQUAL(x,y,\varepsilon)
CU_FAIL(message)

http://cunit.sourceforge.net/doc/headers/CUnit.h
Test code is separate from production code, but calls production code to verify its functionality.
Demo

Write tests for a function named `eval` which takes two int values (x and y) and returns their sum as an int.
void test00(void) {
    int x = 5;
    int y = 10;
    int expected = 15;
    int actual = eval(x, y);
    CU_ASSERT_EQUAL(expected, actual);
}

A test is a void to void function. Set up inputs and an expected (correct) answer. Compute an actual answer using production code. Compare actual and expected values.
void testRunner(int x, int y, int expected) {
    int actual = eval(x,y);
    CU_ASSERT_EQUAL(expected, actual);
}

void test00(void) { testRunner(10,5,15); }
void test01(void) { testRunner(-5,5,0); }
void test02(void) { testRunner(0,10,10); }
void test03(void) { testRunner(10,0,10); }

Can simplify writing many tests by using a helper function, runTest.

Writing new test cases involves just specifying inputs and their corresponding correct output.
Live coding reconstruction

Needed #includes:

#include "CUnit.h"
#include "Basic.h"
int main()
{
    CU_pSuite SimpleAdditionTestSuite = NULL;

    /* initialize the CUnit test registry */
    if (CUE_SUCCESS != CU_initialize_registry()) { return CU_get_error(); }

    /* add a suite to the registry */
    SimpleAdditionTestSuite = CU_add_suite("Tests of eval(x,y) function", NULL, NULL);
    if (NULL == SimpleAdditionTestSuite) {
        CU_cleanup_registry();
        return CU_get_error();
    }

    /* add the tests to SimpleAdditionTestSuite */
    /* For each test, call CU_add_test to add to a given suite, check for NULL, || results to check for error */
    if (NULL == CU_add_test(SimpleAdditionTestSuite, "Expected: eval(10,5) => 15 ", test00))
        || (NULL == CU_add_test(SimpleAdditionTestSuite, "Expected: eval(-5,5) => 0 ", test01))
        || (NULL == CU_add_test(SimpleAdditionTestSuite, "Expected: eval(0,10) => 10 ", test02))
        || (NULL == CU_add_test(SimpleAdditionTestSuite, "Expected: eval(10,0) => 10 ", test03))
    {
        CU_cleanup_registry();
        return CU_get_error();
    }

    /* Run all tests using the CUnit Basic interface */
    CU_basic_set_mode(CU_BRM_VERBOSE);
    CU_basic_run_tests();
    CU_cleanup_registry();
    return CU_get_error();
}
The main() function for setting up and running the tests.
* Returns a CUE_SUCCESS on successful running, another
* CUnit error code on failure.

```c
int main()
{
    CU_pSuite SimpleAdditionTestSuite = NULL;

    /* initialize the CUnit test registry */
    if (CUE_SUCCESS != CU_initialize_registry()) { return CU_get_error(); }

    /* add a suite to the registry */
    SimpleAdditionTestSuite = CU_add_suite("Tests of eval(x,y) function", NULL, NULL);
    if (NULL == SimpleAdditionTestSuite) {
        CU_cleanup_registry();
        return CU_get_error();
    }

    /* add the tests to SimpleAdditionTestSuite */
    /* For each test, call CU_add_test to add to a given suite, check for NULL, || results to check for error */
    if (NULL == CU_add_test(SimpleAdditionTestSuite, "Expected: eval(10,5) => 15 ", test00))
        || (NULL == CU_add_test(SimpleAdditionTestSuite, "Expected: eval(-5,5) => 0 ", test01))
        || (NULL == CU_add_test(SimpleAdditionTestSuite, "Expected: eval(0,10) => 10 ", test02))
        || (NULL == CU_add_test(SimpleAdditionTestSuite, "Expected: eval(10,0) => 10 ", test03))
    {
        CU_cleanup_registry();
        return CU_get_error();
    }

    /* Run all tests using the CUnit Basic interface */
    CU_basic_set_mode(CU_BRM_VERBOSE);
    CU_basic_run_tests();
    CU_cleanup_registry();
    return CU_get_error();
}
```

Create a test suite.
In this case the suite is named `SimpleAdditionTestSuite`
/* The main() function for setting up and running the tests. 
 * Returns a CUE_SUCCESS on successful running, another 
 * CUnit error code on failure. */
int main()
{
    CU_pSuite SimpleAdditionTestSuite = NULL;
    /* initialize the CUnit test registry */
    if (CUE_SUCCESS != CU_initialize_registry()) { return CU_get_error(); }
    /* add a suite to the registry */
    SimpleAdditionTestSuite = CU_add_suite("Tests of eval(x,y) function", NULL, NULL);
    if (NULL == SimpleAdditionTestSuite) {
        CU_cleanup_registry();
        return CU_get_error();
    }
    /* add the tests to SimpleAdditionTestSuite */
    /* For each test, call CU_add_test to add to a given suite, check for NULL, || results to check for error */
    if (
        (NULL == CU_add_test(SimpleAdditionTestSuite, "Expected: eval(10,5) => 15 ", test00))
        || (NULL == CU_add_test(SimpleAdditionTestSuite, "Expected: eval(-5,5) => 0 ", test01))
        || (NULL == CU_add_test(SimpleAdditionTestSuite, "Expected: eval(0,10) => 10 ", test02))
        || (NULL == CU_add_test(SimpleAdditionTestSuite, "Expected: eval(10,0) => 10 ", test03))
    )
    {
        CU_cleanup_registry();
        return CU_get_error();
    }
    /* Run all tests using the CUnit Basic interface */
    CU_basic_set_mode(CU_BRM_VERBOSE);
    CU_basic_run_tests();
    CU_cleanup_registry();
    return CU_get_error();
}
void testRunner(int x, int y, int expected) {
    int actual = eval(x,y);
    printf("  Actual: eval(%d,%d) => %d.  RESULT:",x,y,actual);
    CU_ASSERT_EQUAL(expected, actual);
}

void test00(void) { testRunner(10,5,15); }
void test01(void) { testRunner(-5,5,0); }
void test02(void) { testRunner(0,10,10); }
void test03(void) { testRunner(10,0,10); }
In class we saw CUnit output like this:

Suite: Tests of eval(x,y) function
  Test: Expected: eval(10,5) => 15 ...FAILED
  1. tests.c:11  - CU_ASSERT_EQUAL(expected,actual)
  Test: Expected: eval(-5,5) => 0 ...FAILED
  1. tests.c:11  - CU_ASSERT_EQUAL(expected,actual)
  Test: Expected: eval(0,10) => 10 ...passed
  Test: Expected: eval(10,0) => 10 ...FAILED
  1. tests.c:11  - CU_ASSERT_EQUAL(expected,actual)

Run Summary:    Type  Total    Ran  Passed  Failed  Inactive
               suites   1      1    n/a      0        0
               tests    4      4      1      3        0
               asserts  4      4      1      3      n/a

This output isn't as helpful as it could be, because it doesn't show the actual value computed in the case of a failing test.

I promised to look into how we can most easily provide that feedback in our output.
I did not find a self-evident way to get CUnit to provide this (maybe it's there and I am not seeing it, or maybe it's not).

One way to get that information embedded in our testing output is to explicitly print it from our tests.

Adding one printf statement to our testRunner function works in this example:

```c
void testRunner(int x, int y, int expected) {
    int actual = eval(x, y);
    printf(" Actual: eval(%d,%d) => %d.  RESULT: ", x, y, actual);
    CU_ASSERT_EQUAL(expected, actual);
}
```
Question from class

With this change the output becomes as shown below. I have highlighted the added text in blue.

Suite: Tests of eval(x,y) function
Test: Expected: eval(10,5) => 15 ... Actual: eval(10,5) => 5. RESULT: FAILED
  1. tests.c:11 - CU_ASSERT_EQUAL(expected,actual)
Test: Expected: eval(-5,5) => 0 ... Actual: eval(-5,5) => 5. RESULT: FAILED
  1. tests.c:11 - CU_ASSERT_EQUAL(expected,actual)
Test: Expected: eval(0,10) => 10 ... Actual: eval(0,10) => 10. RESULT: passed
Test: Expected: eval(10,0) => 10 ... Actual: eval(10,0) => 0. RESULT: FAILED
  1. tests.c:11 - CU_ASSERT_EQUAL(expected,actual)

Run Summary: | Type     | Total | Ran | Passed | Failed | Inactive |
-------------|--------|------|-------|--------|---------|
 suites      | 1      | 1    | n/a   | 0      | 0       |
 tests       | 4      | 4    | 1     | 3      | 0       |
 asserts     | 4      | 4    | 1     | 3      | n/a     |
#include "CUnit.h"
#include "Basic.h"
#include "eval.h"

void testRunner(int x, int y, int expected) {
    int actual = eval(x, y);
    printf(" Actual: eval(%d,%d) => %d.  RESULT: ", x, y, actual);
    CU_ASSERT_EQUAL(expected, actual);
}

/* A test is a function void -> void function that
1) states input for the function under test
2) states the expected (correct) result for those inputs
3) computes the actual value produced for the inputs
4) determines whether expected and actual are the same, by
   calling a CUnit ASSERT function
*/
void test00(void) { testRunner(10, 5, 15); }
void test01(void) { testRunner(-5, 5, 0); }
void test02(void) { testRunner(0, 10, 10); }
void test03(void) { testRunner(10, 0, 10); }

/* The main() function for setting up and running the tests.
* Returns a CUE_SUCCESS on successful running, another
* CUnit error code on failure.
*/
int main() {
    CU_pSuite SimpleAdditionTestSuite = NULL;
    /* initialize the CUnit test registry */
    if (CUE_SUCCESS != CU_initialize_registry()) { return CU_get_error(); }
    /* add a suite to the registry */
    SimpleAdditionTestSuite = CU_add_suite("Tests of eval(x,y) function", NULL, NULL);
    if (NULL == SimpleAdditionTestSuite) {
        CU_cleanup_registry();
        return CU_get_error();
    }
    /* add the tests to SimpleAdditionTestSuite */
    /* For each test, call CU_add_test to add to a given suite, check for NULL, || results to check for error */
    if (false) {
        (NULL == CU_add_test(SimpleAdditionTestSuite, "Expected: eval(10,5) => 15 ", test00))
            || (NULL == CU_add_test(SimpleAdditionTestSuite, "Expected: eval(-5,5) => 0 ", test01))
            || (NULL == CU_add_test(SimpleAdditionTestSuite, "Expected: eval(0,10) => 10 ", test02))
            || (NULL == CU_add_test(SimpleAdditionTestSuite, "Expected: eval(10,0) => 10 ", test03))
    }
    CU_cleanup_registry();
    return CU_get_error();
}

/* Run all tests using the CUnit Basic interface */
CU_basic_set_mode(CU_BRM_VERBOSE);
CU_basic_run_tests();
CU_cleanup_registry();
return CU_get_error();
}