CS 211

Lab 10

Test Cases and Unit Testing

Test Cases

- Does your program give the proper results?

- Test Cases just verify if program works.
- This is NOT fixing a non-working program. That is DEBUGGING!
- Testing and Debugging normally go hand-in-hand.

- However, This Lab is only about Testing.

Test Cases

- Does your program give the proper results?

- Three Parts of a Test Case:
  - Description of Program Part/Feature being Tested
  - Set of input to use when running the code
  - Output that should be created
    - Actual Output is Best
    - Sometimes an Output Description is Acceptable
- What input values should be used?

Test Cases

- What input values should be used?

- We want to verify that all “code paths” execute correctly.
  - Each program feature will have a unique code path.
  - Each error check will have a unique code path.
  - A code path may execute multiple input sets.

Test Cases – Equivalence Class

- We want to verify that all “code paths” execute correctly.
- Each code path must be tested.
- An “Equivalence Class” is the set of all input value groups that will execute a single code path.
- Successful execution of one input value group from an Equivalence Class should result in all input value groups from that Equivalence Class to be successful

Test Cases

- We want to verify that all “code paths” function correctly.
- We need to test (at least) one input value group from each Equivalence Class.
- However, testing too many input value groups from a single Equivalence Class is a waste of time.

- Consider the following Program Specification:

Program Specification for a Simple Tax Calculation

Create a web form that will allow the user to calculate the amount of taxes owed to the government. The user should enter the income amount in an input field, then press/click a button and the amount of taxes owed is to be displayed. If the user enters a non-numeric income amount or a negative income amount, display an error message telling the user that a positive numeric value must be entered. The amount of taxes owed is determined by:

- If the income amount is $5,000 or less, the tax amount is 10% of the income amount.
- If the income amount is more than $5,000 and is $50,000 or less, the tax amount is 15% of the income amount.
- If the income amount is more than$50,000, the tax amount is 20% of the income amount.

Equivalence Classes

- The Program Specification has 5 Equivalence Classes:

  EC1: values from $0 up to $5000

  EC2: values greater than $5000 up to $50000

  EC3: values greater than $50000

  EC4: values less than $0 (negative input)

  EC5: non-numeric input

Equivalence Classes

- Each Equivalence Classes should produce 1 test case
- The input value group should be near the middle/median of the range of possible inputs.
- Equivalence classes with an infinite range (…values greater than $50000…), should select input from the middle of the “expected range of inputs”

Boundary Case Testing

- Additional test cases should be included to verify the input values at the boundary between two Equivalence Classes.
- Each boundary should generate multiple test cases:
  - The boundary value itself
  - The values on either side of the boundary value

Boundary Case Testing

- The tax program specification has 3 Boundary Values

- the value of $0 (between EC1 and EC4)
- the value of $5000 (between EC1 and EC2)
- the value of $50000 (between EC2 and EC3)

Unit Testing Frameworks

- This type of testing can create a huge amount of tests.
- For the short specification above a minimum of 14 tests can easily be needed
- We need automated tests to make out lives easier and to not “wimp out” on doing enough testing

- There exists a number of Unit Testing Framework to help automate these tests.

TinyTest

- A testing framework similar to the JUnit framework for Java
- Available on GitHub at

  https://github.com/joewalnes/tinytest

- Using ASSERT_EQUALS( ) statements to verify EXPECTED output matches ACTUAL output of a function/method call

  ASSERT_EQUALS (5.0, sqrt(25.0));

TinyTest – main()

- Uses RUN ( ) macro to call a testing function.

  RUN (testDistanceFrom);

- RUN ( ) stores information to be used by TEST_REPORT ()

- TEST_REPORT () is called to display results of all the test.

  return TEST_REPORT();

TinyTest – testing functions

- Functions with no parameter and a void return called by RUN

```
RUN ( testMathLibrary );
```

- Sets up calls to the actual function/method being tested.
- Verifies if the EXPECTED result matches the ACTUAL result.

```
void testMathLibrary ( )  {
   ASSERT_EQUALS ( 5.0, sqrt ( 25.0 ) );
}
```

TinyTest – testing functions

- Testing functions can contain multiple tests and other code to help set-up for the test(s).

```
void testDistanceFrom ( )  {
        Point2d p1 ( 0, 0 );
        Point2d p2 ( 8, 0 );
        ASSERT_EQUALS ( 8.0, p1.distanceFrom(p2) );
        ASSERT_EQUALS ( 8.0, p2.distanceFrom(p1) );
}
```

TinyTest – testing functions

- When a Testing Function finds its first error, it returns.

```
void testDistanceFrom ( )  {
        Point2d p1 ( 0, 0 );
        Point2d p2 ( 8, 0 );
        ASSERT_EQUALS ( 75.0, p1.distanceFrom(p2) );
        // since above test fails,
        //      the following code is not executed
        ASSERT_EQUALS ( 8.0, p2.distanceFrom(p1) );
}
```

TinyTest – TEST_REPORT

- When all testing functions successfully complete, TEST_REPORT() gives a message similar to:

PASSED [Point2dTest.cpp] (total:4)

- Name of testing program given in square brackets
- Number of test function executed given in parenthesis

TinyTest – TEST_REPORT

- When a testing function fails an ASSERT_EQUALS ( ), TEST_REPORT() gives a message similar to:

```
failure: Point2dTest.cpp:60: In test testDistanceFrom():
   p1.disanceFrom(p2) ((75.0) == (p1.disanceFrom(p2)))
FAILED [Point2dTest.cpp] (passed:3, failed:1, total:4)
```

- First line gives specifics on failed ASSERT_EQUALS statement
- Last line total on all testing functions that passed and failed

TinyTest – Multiple Source Code Files

- Multiple Source Code files is EXTREMELY helpful with TinyTest
- The code to be tested is in one source code file
- The production main() that uses the code in a second file
- The main() TinyTest driver to test the code in a third file
- A makefile could compile either the production version or the testing version depending on the target given!