Software Testing

CS II: Data Structures & Abstraction

Michael John Decker
Adapted with permission from
Jonathan I. Maletic
Kent State University

What is Software Testing?

- The objective is to find faults
- A fault is incorrect output for a given input
- Faults are caused by errors in your program
- An error (bug) is an incorrect peace of code
  - Missing condition
  - Incorrect stopping condition

How to find the Faults?

- Test cases are developed to exercise your program with the goal of finding errors
  - This will lead you to fault(s) in the code
- The best test case is one that has a high probability to cause an error
- Start by testing each method (unit tests)
- Then each class in full (module tests)
- Then the whole program (system tests)

Information Needed

- A method/function is defined by an input/output specification (I/O spec).
  - The pre and post conditions describe the I/O spec
- A method/function is also defined by its implementation details
  - For-loop vs while loop vs recursive

Black Box vs Glass Box

- Black box testing uses only the I/O spec to develop test cases
- Glass (white) box uses only the implementation details to develop test cases
- Both types of information are necessary to develop a good set of test cases for a method/function

Number of Test Cases?

- Most functions have a very large (i.e., infinite) number of possible inputs and outputs
- Do you need to test all of these to be satisfied your function behaves correctly? NO!
- Again, the best test case is one that has a high probability in uncovering a fault

Pairing Down Test Cases

- Can take advantage of symmetries, equivalencies, and interdependencies in the data to reduce the number of test cases.
  - Equivalence Testing
  - Boundary Value Analysis
- Determine the ranges of input & output
- Develop equivalence classes of input/output
- Examine the boundaries of these classes carefully

Equivalence Partitioning

- Input data and output results often fall into sets of related data called equivalence partitions
  - Given the range -20, … 20
  - One partition: {-20, … , -1}, {0}, {1, … 20}
- Test cases should be chosen from each of the different partition
  - -10, 0, 10

Boundary Value Analysis

- Given the equivalence partitions:
  - {-20, … , -1}, {0}, {1, … 20}
- Choose test cases at the boundary of these
  sets:
  - -20, -1, 0, 1, 20

Example

- Problem: Search an array Tbl of size N for a key K, Return the location of first occurrence
- Equivalence Partitions:
- N: {0, …, MAX}
- K: {-maxint, …, maxint}
- Tbl: contains K, does not contain K
- Tbl: K at 0 … N
- Tbl: contains multiple K

Test Cases

- One from each equivalence and the boundaries
- N: 0, 1, MAX/2, MAX-1, MAX
- K: -10, 0, 10
- Tbl: K at 0, 1, MAX/2, MAX-1, MAX
- Tbl: K not in the array
- Tbl: Multiple K in the array

Test-Driven Development

- Testing is an integral part of development
- To write a method/function:

1. Determine the I/O spec
2. Develop test cases
3. Implement the method
4. Run the method against the test cases
5. Fix any faults (debugging)
6. Go to 4 (or if serious problems 1)

Unit Testing

- Build a program (called a driver) for unit testing
- One test driver (main) for each method
- Test simplest methods first, more complex later
- Test constructors, I/O, simple accessors, then more complex operations

Regression Testing

- Each time you add a new method to your class or fix a fault, run ALL your test cases
- Adding something new or fixing a problem may have side effects
- Re-running your test cases will help to uncover these problems (if they happen)

Example Driver

```
#include <cassert>
int main() {
    Set a;
    assert(a.card() == 0);

    Set b(1, 4);
    assert(b.card() == 2);
    assert(b == set(1, 4));
    assert(b != a);

    std::cout << “{1, 4} == ‟ << b << endl;
    std::cout << “All Tests Completed‟ << endl;
    return 0;
}
```

Mantra (TDD)

- Develop Test Cases before you code
- Test each time you add code
- Run all test cases