Software Testing
CS II: Data Structures & Abstraction

Jonathan I. Maletic
Kent State University
Spring 2013

Software Testing

- The objective of software testing is to cause failures to find faults/errors in the system

- A *Failure* is an incorrect output for a given input
- Failures are caused by faults/errors in your program
- A *fault/error* is an incorrect piece of code/document (bug)
  - Missing condition
  - Incorrect stopping condition
  - Wrong equation
  - etc.

Finding Faults

- Test cases are developed to exercise your program with the goal of uncovering the faults
  - Given a failure, *debugging* is done to identify the fault/error(s) in the code
- The **best** test case is one that has a high probability to cause a failure
- Start by testing each method (unit tests)
- Then each class in full (module tests)
- Then the whole program (system tests)

Information Needed to do Testing

- A method/function is defined by an input/output specification
  - Termed the I/O spec
  - The pre and post conditions of a function describe the I/O spec

- A method/function is also defined by its implementation details
  - For-loop vs while loop vs recursive

Assertions, Pre/Post Conditions, Invariants

- Assertion (ASSERT): A statement that is true at a specific point in the program
- Pre-condition (REQUIRES): A statement that must be true before a method/function is called/executed
- Post-condition (ENSURES): A statement that is true after a method/function is executed
- Invariants – statement that are always true
  - LOOP INV: Statement that is always true within the scope of a loop
  - CLASS INV: Statement that is always true for object of a give type/class
  - GLOBAL INV: Statement that is always true globally

Example

```
//REQUIRES:  assigned(key && tbl[0,..size-1]) && (size >= 0)
//ENSURES:   (RETVAL == i where tbl[i] == key) ||
//           (RETVAL == -1 where i == size)
int search(const int tbl[], int key, int size) {
    //ASSERT (size >= 0) && (sizeof(tbl) == size)
    int i = 0;
    //ASSERT (size >= 0) && (sizeof(tbl) == size) && i == 0
    while (i < size){ //LOOP INV: (key != tbl[0,..,i-1]) && (i <= size)
        if (tbl[i] == key) {
            //ASSERT (tbl[i] == key) && (i < size)
            return i;
        }
        //ASSERT (key != tbl[0,..,i]) && (i < size)
        ++i;
        //ASSERT (key != tbl[0,..,i-1]) && (i <= size)
    }
    //ASSERT key != tbl[0,..,size-1]
    return -1;
}
```

Black-box vs Glass-box Testing

- Black-box testing uses only the I/O spec to develop test cases
  - Ignore the implementation and test to specification
  - Also called specification testing
- Glass (white) box uses only the implementation details to develop test cases
  - Ignore the specification and test the code
  - Test every path, make sure every line is executed
- Both types of information are necessary to develop a good set of test cases for a method/function

Number of Possible 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? Thankfully, no!
- Again, the best test case is one that has a high probability in uncovering a fault

Pairing Down Test Cases

- 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 (a topic in Discrete Structures)
  - 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 boundaries of these sets:
  - -20, -1, 0, 1, 20

Example

```
//REQUIRES: (n>=0) && tbl[0,.., n-1]
//ENSURES:   RETVAL == i where key == tbl[i]
//         or RETVAL == -1 if key != tbl[0,.., n-1]
int search(int key, int tbl[], int n) {
    int i = 0;
    while (i < n) {
        if (key == tbl[i]) return i;
        ++i;
    }
    return -1;
}
```

Testing to the Specification

- Problem: Search an array tbl of size n for a key, Return the location of first occurrence (or -1)

- Equivalence Partitions:
  - n: {0, …, maxint}
  - key: {-maxint, ..., maxint}
  - tbl: contains key, does not contain key, contains multiple keys
  - tbl: key at 0 ... n-1

Black-box Test Cases

- One from each equivalence class and the boundaries

- n: 0, 1, 25, 500
- key: -10, 0, 10
- tbl: key at 0, 1, n/2, n-2, n-1
- tbl: key not in the array
- tbl: Multiple keys in the array

Testing to the Code

- Make sure each line is executed

```
int search(int key, int tbl[], int n) {
    int i = 0;
    while (i < n) {
        if (key == tbl[i])
            return i;
        ++i;
    }
    return -1;
}
```

Testing to the Code

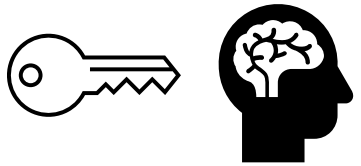
- Make sure each line is executed

```
int search(int key, int tbl[], int n) {
    int i = 0;                          //Any n
    while (i < n) {                     //Any n
        if (key == tbl[i])              //n>0
            return i;                   //n>0, key in tbl
        ++i;                            //n>0, key != tbl[0]
    }
    return -1;                          //Any n, key not in tbl
}
```

Glass-box Test Cases

- Some n>0 and key in/not in

- n = 10, key in tbl
- n = 10, key not in tbl

Test-Driven Development (TDD) 

- Testing is an integral and critical part of development
- Without testing, you do not have a functioning application (bugs)

- 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 until all tests pass

Unit Testing

- Build a program for unit testing
  - A main with all the tests
  - Called a *test driver*
- One test driver for each method
- Test simplest methods first, more complex later
- First test constructors, I/O, simple accessors
- Then more complex operations
- Build confidence that simple operators are correct so you can localize errors when testing more complex operations in the class

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 that cause another fault
- Re-running your test cases will help to uncover these problems (if they happen)

Example Test 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 << std:: endl;
    std::cout << “All Tests Completed“ << std:: endl;
    return 0;
}
```

TDD Mantra (Agile Development Process)

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