Testing and Debugging

- **Validation: is a process designed to increase our confidence that a program works as intended.**

- **Debugging: is a process of ascertaining why a program is not functioning properly**

- **Defensive programming: is the practice of writing programs in a way to ease the process of validation and debugging.**

Validation

- **Can be done through Verification Or Testing**

- **Verification: is a formal or informal argument that a program works on all possible inputs**

- **Testing: is the process of running a program on a set of test cases and comparing the actual results with expected results.**

Testing

- **Purpose: reveal the existence of errors**
- **examine relationship between inputs and outputs**
- **exhaustive testing is impossible for many programs.**
- **choosing the proper test data.**
- **Goal: find a reasonably small set of test cases that will allow us to approx. the info we get thru exhaustive testing.**

Black-box Testing

- **Generate test cases from specifications alone**
- **Advantages:**
  - **testing is not adversely influenced by the compt. being tested.**
  - **robust with respect to changes in implementation**
  - **results can be interpreted by people unfamiliar with the internals of the program**

Testing Paths through the Specification

- Explore alternate paths through the spec.

- Paths can be thru both REQUIRES and EFFECTS clauses

Testing Paths through the Specification (Example: Path thru Requires clause)

```
static float sqrt(float x, float epsilon)
  //REQUIRES: x >=0 and .00001 < epsilon < .001
  //EFFECTS: Returns sq s.t.
  // x – epsilon <= sq*sq <= x + epsilon
```

Test data:

1. x = 0 and .00001 < epsilon < .001
2. x > 0 and .00001 < epsilon < .001

Testing Paths through the Specification (Example: Path thru Effects clause)

```
static boolean int x)
  //EFFECTS: If x is a prime returns true
  //          else returns false
```

Test data:

1. x prime
2. x not prime

Testing Paths through the Specification (Another Example)

```
static boolean isOdd(int x)
  // Effects: If x is odd, returns false
  // else returns true
```

Test data:

1. x odd
2. x not odd

Testing Paths through Effects Clause

```
static int search(int[] a, int x) throws
    NotFoundException, NullPointerException
 //Effects: If a is null, throws
 //NullPointerException else if x is in a,
 //returns i s.t. a[i] = x, else throws
 //NotFoundException
```

Test data:

1. x is in a
2. x not in a
3. a is null

Test data should also test the cases where exception must be thrown.

Testing Boundary Conditions

- **Always test with “typical” input values**
- **also test the boundary conditions.**
- **e.g. `sqrt`, include cases for epsilon very close to .001 and .0001**
- **e.g. for strings, test with empty string, one character string**
- **for arrays, test with empty array, one element array**

Aliasing errors

- **A kind of boundary condition occurs when two different references refer to the same object.**

- **Test data should include input which are aliases of each other.**

Aliasing errors

```
static void appendVector( Vector v1, Vector v2)
  throws NullPointerException
  //Modifies: v1 and v2
  // Effects: If v1 or v2 is null, throws
  // NullPointerException else removes all
  // elements of v2 and appends them in reverse
  // order to the end of v1
```

Test : what if v1 == v2, i.e. v1 and v2 refer to the same object

Black-box testing

- **test all paths through the specification (not implementation)**
- **test boundary conditions**
- **check for aliasing errors**

Glass-box Testing

- **Code of the program being tested is taken into account**
- **should supplement Black-Box testing**
- **Exercise each path through the program**

- **Goal: test set s.t. each path is exercised by at least one test case. Such test set is called *path-complete*. *Path-complete* tests also need to take exceptions into account.**

Glass-box Testing (Example)

```
static int maxOfThree(int x, int y, int z) {
  if( x > y )
    if(x > z) return x;
    else return z;
  if(y > z) return y;
  else return z;
}
```

n^3 inputs, only 4 paths

Test data into 4 classes:

3, 2, 1 (x > y and z)

3, 2, 4 (x > y but x < z)

1, 2, 1

1, 2, 3

Glass-box Testing (contd.)

Path-completeness is not sufficient to catch all errors.

```
static int maxOfThree(int x, int y, int z) {
  return x;
}
```

Path-complete test set:

2, 1, 1

Glass-box Testing (contd.)

Problems of Path-completeness:

- **not likely to reveal the existence of missing paths**
- **too many different paths thru a program**

Specification must be taken into account

Glass-box Testing (contd.)

Approximate Path-complete testing for loops and recursion:

- **For loops with a fixed amount of iteration, tests should include one and two iterations of the loop. For loops with a variable amount of iteration, tests should include zero, one and two iterations of the loop. Tests to terminate the loop.**

- **For recursion, tests should include no recursion and one recursive call.**

Path-complete tests should also take exceptions into account

Testing Procedures

Consider both specification and implementation

```
static boolean palin(String s) throws NullPointerException
{
 //Effects: If s is null throws NullPointerException, else
 // returns true if s reads the same forward and backward;
 // else returns false
 int low = 0; int high = s.length() – 1;
 while(high > low) {
  if(s.charAt(low) != s.charAt(high)) return false;
  low++; high--; return true;
 }
}
```

Testing Procedures: look at spec

Test for

- **null argument**
- **case where true is returned**
- **case where false is returned**
- **boundary conditions: empty string, one character string**

```
"", "d", "deed", "ceed"
```

Testing Procedures: look at code

Test for `""`, `"d"`, `"deed"`, `"ceed"`, `"aaba"`, `"aca"`

- **NullPointerException raised by calling length()**
- **Not executing the loop**
- **returning false in first iteration**
- **returning true after the first iteration**
- **returning false in second iteration**
- **returning true after second iteration**

Testing Data Abstractions

Consider both specification and implementation of each method.

- **test the methods in a group**

- **call repOk() after each call to a method.It must return true, otheriwse found a bug.**

Unit Testing and Integration Testing

Testing occurs in two phases:

- **Unit Testing: each individual module is tested in isolation**

- **Integration testing: the entire program is tested with all the modules**