Testing : References

- *Software Engineering* : Roger Pressman
  - Chapter 16 *Software Testing Techniques*
- *Software Engineering* : Ian Sommerville
  - Chapter 22 *Defect Testing*

Testing

Deutsch : The development of software systems involves a series of production activities where opportunities for injection of human fallibilities is enormous …

Bezier : There is a myth that if we were really good at programming, there would be no bugs to catch …

Testing cannot show the absence of defects, only the presence of errors …

Myers : Testing Objectives

- Testing is a process of executing a program with the intent of finding an error
- A good test case is one that has a high probability of finding an as-yet undiscovered error
- A successful test is one that uncovers an as-yet undiscovered error

Testing Principles

- All tests should relate to customer requirement
- Tests should be planned long before testing begins
- Exhaustive testing is not possible
- To be most effective, testing should be conducted by an independent third party
- Pareto principle applies to testing
  - 80% of errors found in 20% of modules

Software Testability

- quality of software that makes it easy to test
  - observability
  - *what you see is what you test*
    - distinct output for each input
    - current and past system states and variables are visible or queriable during execution
    - incorrect output (i.e. an error) is easily identifiable

Software Testability

- controllability
- *the better we can control the software, the more the testing can be automated …*
  - all possible outputs can be generated
  - all code is executable
  - input and output formats are consistent and structured
  - states and variables can be directly controlled

Test Strategy

- decide on an approach to testing
- design test data sets
- white-box
  - from structure of code
- black-box
  - based on function definition i.e. from pre and post-conditions

White Box

- structural testing
- can see the way the function is implemented
- use program control structures to derive test cases
  - exercise all independent paths
  - exercise all logical decisions
  - execute all loops at boundaries and within limits
  - exercise all internal data structures

White Box

- if-else
  - true and false branches
- while, do, for
  - skip loop
  - one pass, two passes
  - m passes where m < n
  - n-1, n, n+1 passes
- switch
  - all cases

White Box

- internal structures
- e.g. array
  - empty, full, partial
  - even/odd number of elements
  - first element, last element, middle element
  - before first, after last

Black Box

- functional testing
- can't see the code
- can only go by outputs derived from given inputs
- use functional specification (VDM) to derive test cases
- equivalence partitioning
  - classes of input values with common properties
- boundary value analysis
  - select input values at *edge* of class

Test Case Results

- name of FID function being tested
- tabulated results
  - state value(s) before test
  - input value(s)
  - expected state value(s) after test
  - actual state value(s) after test
  - expected output value(s)
  - actual output value(s)
  - test result [passed/failed]

White Box Example

```
int s = new int[10];
int length = 10;
boolean isMember (element e) {
int j;
 for (j = 0; j < length; j++) {
  if ( e == s[j] ) { return true;}
 }
 return false;}
```


White Box Example

- if
  - condition true, e in s
  - condition false, e not in s
- array
  - length = 0, s = []
  - length = 10, s = [1,2,3,4,5,6,7,8,9,10]
  - length = 1, s =[1]
  - length = 5, s = [1,2,3,4,5]
- gives 2 * 4 = 8 tests (actually 7, e never in empty s)

White Box Example

- Test 1 isMember

  state before s =[], length = 0;
  input e = 2;
  expected state after [no change] s =[], length=0;
  actual state after s =[], length=0;
  expected output false
  actual output false
  test result passed

White Box Example

- Test 2 isMember

  state before s =[1,2,3,4,5], length = 5;
  input e = 2;
  expected state after [no change] s =[1,2,3,4,5], length=5;
  actual state after s =[1,3,4,5], length=4;
  expected output true
  actual output true
  test result failed

Black Box Example

- use pre-condition

  setFlavour (f : String) ok:B
  ext wr flavours:Flavours

  pre
  f ε dom flavours

Black Box Example

- use pre-condition

  set value(s) in flavours to test
  cases for f
  ε dom flavours
  ¬ ε dom flavours

  error in pre-condition
  flavours unchanged

Black Box Example

- use post-condition

post

(<u>flavours</u>(f) $\wedge$ flavours = <u>flavours</u> $\wedge$ ok = true)

$\vee$

($\neg$ <u>flavours</u>(f) $\wedge$ flavours = <u>flavours</u> † {f $\rightarrow$ true} $\wedge$ ok = true)

error

flavours = <u>flavours</u> $\wedge$ ok = false

Black Box Example

- use post-condition

set value(s) in flavours to test

f cases

in flavours and not added

not in flavours and added

Black Box Example

- Test 1 setFlavour

state before flavours = {Vanilla}

input f = Chocolate

expected state after flavours = {Vanilla, Chocolate}

actual state after flavours = {Vanilla, Chocolate}

expected output true

actual output true

test result passed

Black Box Example

- Test 2 setFlavour

state before flavours = {Vanilla}

input f = Vanilla

expected state after flavours = {Vanilla}

actual state after flavours = { }

expected output true

actual output true

test result failed

Black Box Example

- Test 3 setFlavour

state before flavours = {Vanilla}

input f = Apricot

expected state after flavours = {Vanilla}

actual state after flavours = {Vanilla}

expected output false

actual output false

test result passed

Test Harness Structure

one menu item per FID function

one switch/case entry per FID statement

set value(s) in state

accept function input argument(s)

call function

display function output argument(s) and status

display aspects of state that may have changed

add extra menu item(s) to view state

Test Harness Class

Create a test harness and run it

```
public class TestHarness
{  public static void main(String args[]) throws
   Exception {
  TestHarness th = new TestHarness();
  try { th.run(); }
  catch (Exception e) { System.out.println(e); }
  th.quit(); }
```


Test Harness Constructor

initialise a system state by creating a FIDo instance,
open a data stream for argument input,
create local variables

```
private FIDo f;
public TestHarness () {
f = new FIDo();
in = Text.open(System.in);
nFlavours = new WrappedInt();
```


Test Harness Menu

```
System.out.println("  (1) setSize");
System.out.println("  (2) setHowServed");
…
System.out.println(" (19) print");
System.out.println(" (20) quit Test Harness");
Text.prompt("Operation:");
inputCode = Text.readInt(in);
switch(inputCode) {
```


Test Harness - Operation

```
case 1:
Text.prompt(" Large size [true/false]?");
s = f.setSize( readBoolean() );
break;

case 19:
s = f.print();
break;
```


Test Harness Usage

- input from keyboard, output to screen
  - demonstrating
- input from keyboard, output to file
  - results saved for report
- input from file, output to file
  - bulk testing

Project Files

- *.java per BE class
- *.java for state (container class)
- Wrapped*.java
- FIDo.java
- TestHarness.java
- Backend.html
- *.dat (to hold the data)