Chapter 7

Testing

1. Introduction
2. Module testing
   2.1 White box testing (Basis path test, loop test)
   2.2 Black box test (Equivalence partitions. Boundary value Analysis)
3. Integration testing
4. Validation testing
5. System testing

1. Intro
   Testing is a process of executing program with the intent of finding an error. A good test cast is one that has high probability of finding an as-yet undiscovered error. (Note: testing cannot show the absence of defect, it can only show the errors are present.)

How to design a test case?

Testing Principles
1) We cannot have exhaustive test cases.
   Ex: Procedure xyz (i: integer);  -∞ — +∞
   ⇒ We need to design test cases

2) Pareto principle: (80-20%)
   80% of defect can be traced to 20% of code (modules)
   ⇒ if we find an error in a module, we should test that module extensively.

4 steps of test

```
MT  System
    testing
MT  Tested
     integration
VT  Validated
     SW1
     SW2
     SW3

Step 1 Code
Step 2 AD (design)
Step 3 REQ/SPEC
Step 4 Whole system → Real environment

MT - Module test,  IT - Integration test,  VT - Validation test
```
2. Module test

Black box (look at the module as a whole, don’t care what is inside)
Look inside m
White box (glass box)
(Open-up box) test individual unit

2.1 White box test (Glass box)
(A) Basis path test
~ is a white box test technique to test the “logical complexity” of a module, based on the basis set and execution paths.

3 concepts:
a) flow graph of a module. Representation of the code in graph.

3 elements in flow graph.

i. Sequence:

\[ T_1 \rightarrow T_2 \] if

ii. Selection:

iii. Iteration:
Ex:

Procedure Process-record;
    Begin
        While Recordsremain
            Read RECORD;
            If Record-field = 0 then
                process-RECORD
                store in Buffer;
                increment counter;
            else if record-field = 2 then
                reset the counter
            else process record
                store in-file
            end if
        end while
    end

⇒ Flow graph  
A node = multiple sequence statement

R4
b. Independent paths:
Any path through the program that introduces at least one new set of statement is called ~. In terms of flow graph, an I.P. must move along a path with a least.

One edge that has NOT been traversed

Path 1: 
Path 2: 
Path 3: 
Path 4: 

Some examples:

- Path 1: 1 – 9
- Path 2: 1 – 2 – 3 – 7 – 8 – 9
- Path 3: 1 – 2 – 4 – 5 – 7 – 8 – 1
- Path 4: 1 – 2 – 4 – 6 – 7 – 8 – 1

c. Cyclomatic complexity (C.C.)
Is a software metric that provides a quantitative measure of the logical complexity of a module (program)

It defines,

1. # of independent path
2. upperbound for the basis path test

The C.C. is computed as follow:

1. # of regions in flow graph
2. \( E - N + 2 \)  
   \( E \): # of edges  
   \( N \): # of nodes
3. \( P + 1; \)  
   \( P \): # of predicate nodes

1. # of regions:
   region is an area bounded by edges & nodes
   Ex: 4  Ri  \( i = 1, 2, 3, 4 \)

2. \( E - N + 2 \)
   Ex: \( E - N + 2 = 11 - 9 + 2 = 4 \)

3. \( P + 1 \)
   Ex: \( P + 1 = 4 \)

Basis Path test steps
Step 1: find the F.G. of the module
Step 2: Determine the C.C.
Step 3: Determine the LIP
Step 4: Prepare test cases such that each test case will force the execution of each LIP
Ex: Procedure AVERAGE

counter = 1;
total input = total valid = sum = 0;

while value[counter] <> e of and
  total input ≤ 100 do
      Increment total input by 1;
      If value[counter] ≥ minimum and
      value[counter] ≤ maximum then
          begin
              Increment total valid by 1;
              sum: = sum + value[counter];
          end
  end if
      Increment counter by 1;
end while

if totalvalid > 0 then
    average = sum / total valid
else average = nil;
end if
Step 1. Create Flow Graph

Step 2. Find C.C

1. # of regions = 6
2. $E - N + 2 = 17 - 13 + 2 = 6$
3. $P + 1 = 5 + 1 = 6$

Step 3. LIP

1. 1 2 10 11 13
2. 1 2 10 12 13
Step 4. Prepare test cases:

Path 1: 1 2 10 11 13

Value \[i\] = \text{eof} where \ 2 \leq \ i \leq 100
Value \[k\] = valid input where \( k = i - 1 \)
Expected: Average = \text{sum} / \text{total valid}

Observe value: correct average?

Path 2. 1 – 2 – 10 – 12 – 13

Test case:  

Expected value: Average = \text{Nil} “-999”
Observe value: Average = \text{Nil}?
result: pass / fail

Path 3. 1 – 2 – 3 – 10 – 11 – 13

Test case: 

Prepare \geq 100 \text{ values at least one valid}
Expected: average = \text{sum} / \text{total valid}
Observe value: average correct?
result: pass / fail
White-box testing

(B) Loop test

White box test technique that focuses exclusively on the validity of loop constructs.
⇒ prepare certain # of test cases
How many depends on the type of the loop
4 types of loop:
1) Simple loop
2) Concatenated loop
3) Nested loop
4) Unstructured loop

i. Test for a simple loop

- 6 Test cases
- Boundary tests (upper bound, lower bound)
- General case 1

(Test case 1) 0 times through the loop
2) 1 time through the loop
3) 1 < # < n-1 (general case)
4) n-1 times
5) n times
6) n+1 times

Ex.
Data: Array [1..10] of integer
Left = start = 1
While Data[left] < startvalue DO
    Left = left + 1

Test case 1. Data[left] ≥ startvalue
    Expected value: left = start = 1
    Observed value: left: 1 start:1
    Result: Pass / NO pass

Test case 2. - Data[2] ≥ startvalue
    Data[1] < startvalue
    - expected value left = 2
    - observed ~ pass / no pass
Test case 3. 

| < | < | < | ≥ |

Expected: left = 6  
Observed: —— Pass / No pass, …

ii. nested loop

**Step 1** start at the inner most loop set all the outer loops at the minimum value.

**Step 2** work outwards by conducting test for next outer loop etc etc. – keep all the other loops at the min. values: - inner loops at the “typical” value for the counters.

Ex:  
For i = 1 to 20 do
    :  
    x = …
L3  
    For j = 1 to 50 do
        :  
        y = …
L2  
    For k = 1 to 100 do
        :  
        z= …
L1  
    Value [i, j, k] = x + y + z;
    :

**Step 1** test for L1  
Set i = 1, j = 1, prepare test cases for L1  
Where we prepare 6 test cases.

**Step 2** test for L2  
Set i = 1, k = “typical value” = 50  
Prepare 6 test cases for j (L2)

**Step 3** test for L3  
Set j = 25, k = 50  
Prepare 6 test cases for i (L1)

Total 18 test case

iii. concatenated loop

Two simple loops

L1
L2
Not independent from each other
Step 1  start with the first loop. Set the 2nd loop counter to minimum value

Step 2  Go to the next loop by setting the 1st loop to “typical Value” for the counter.

Ex.

For i = 1 to 10 do
  x = i + 1;
end for

For j = 1 to 20 do
  y = f (x)
end for

Step 1  Test for L1 prepare test cases for L1 by changing i
Set j = 1

Step 2  Set I = “typical value” = 5
Prepare 6 test cases for j

iv. unstructured loop

Use of “goto”
⇒ structure first → test

2.2 Blackbox test
This method focuses on the functional requirement of a module (program) i.e. black box test enables the tester to derive set of input condition that will fully exercises all functional requirements. ⇒ derive test cases.

(A)  Equivalence Partitioning (E.P.)
This method divides the input domain into classes of data from which test cases can be designed. E.P. strives to define test cases that uncover class of errors.

Ex 1: Integer (value) we divide the input into EC (equivalence classes). below class < value class < above class

Ex 2: range below class < within range class < above class 3 test cases
Ex 3: set in / not in 2 test cases
Ex of E.P.: a simple example of a module that tests valid area code of a TEL #.

```
range Y / N
Between 201 & 905 → Y else N
```

Prepare 3 test cases
1 From the range
1 From below the range
1 From above the range

(B) BVA (Boundary value Analysis) Similar to E.P.
Except we are choosing boundary values rather than any value in the class.

Ex: as previous (area code example)
Class: a, b. 2 cases below a, b, 6 test cases above a, b.

3. Integration Testing

(A) Top-down
(i) main control is the test driver. Build stubs for all subordinate modules
(ii) replace the stubs with real module one at a time.
(iii) Conduct the test as each module is introduced.
No need to build another new test driver module.

(B) Bottom-up
(i) Few modules are combined into builds (clusters) that perform a specific function.
(ii) Write a driver to test this function.
(iii) The drivers are removed and clusters are combined upward along in the program structure.
2 calls 5, 6 for some purposes, not testing them. So need test module (driver)

4. Validation test: Test of the software against the REQ using black box method
   - Alpha-Beta test (Alpha: on the developer’s side) (Beta: on the user’s side)

5. System Test ↔ not testing functional requirement
   Series of test where the primary purpose is to fully exercise the entire software in real condition.
   (i) Recovery test: create test cases where system fails ⇒ correct recovery
   (ii) Security test: create test cases to test security
   (iii) Stress test: test cases demand abnormal request for resources
   (iv) Performance test: run time performance of the integrated system.