UNIT 4 BOUNDARY VALUE TESTING, EQUIVALENCE CLASS TESTING, DECISION TABLE TESTING

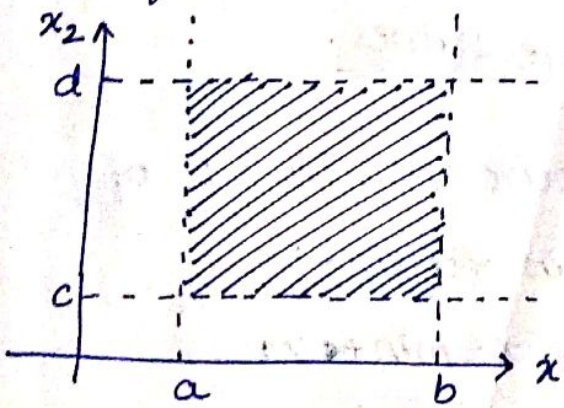
BOUNDARY VALUE TESTING (BVA)

Boundary value Analysis is a technique of Black Box testing. This form of testing mainly focuses on i/p domain & to identify testcases for program or software.

- When a function F of two variables x_1 and x_2 is implemented as a prog. the i/p variables x_1 and x_2 will have some boundaries:

$$a \leq x_1 \leq b$$
$$c \leq x_2 \leq d$$

- The intervals [a,b] and [c,d] are referred as ranges of x_1 and x_2. The input space (domain) of our function F is as shown. Any point within the shaded rectangle is a legitimate input to the program or function F.



BVA focuses on boundary of i/p to identify test cases.

The main idea is that error tend to occur near extreme values of i/p variables.

BVA is a s/w testing technique in which tests are designed to include representatives of boundary values in a range. Given that we have a set of test vectors to test the system, a topology can be defined on the set.

- The basic idea of BVA is to use i/p variable values at their "T" commercially available testing tool "T" refer to these preferable values

a) minimum (min)
b) just above the minimum (min +)
c) a nominal value (nom)
d) just below the maximum (max -)
e) maximum (max)

Ex: Given the range 1 to 10

min = 1 min + = 2 nom = 5
max - = 9 max = 10

- The next part of BVA is based on a "Critical assumption (single fault assumption) in reliability theory.
- It means that failures are only the result of the simultaneous occurrence of two (or more) faults. Therefore, the BV test cases are obtained by holding the values of all but one variable at their nominal values & letting that variable assume its extreme values.
- The BVA test cases for our function F of two variables as shown in figure:

Test cases An+1

{< x1 nom, x2 min >, < x1 nom, x2 min+ >,
< x1 nom, x2 nom >, < x1 nom, x2 max- >,
< x1 nom, x2 max >, < x1 min, x2 nom >,
< x1 max-, x2 nom >, < x1 min+, x2 nom >,
< x1 max, x2 nom > }

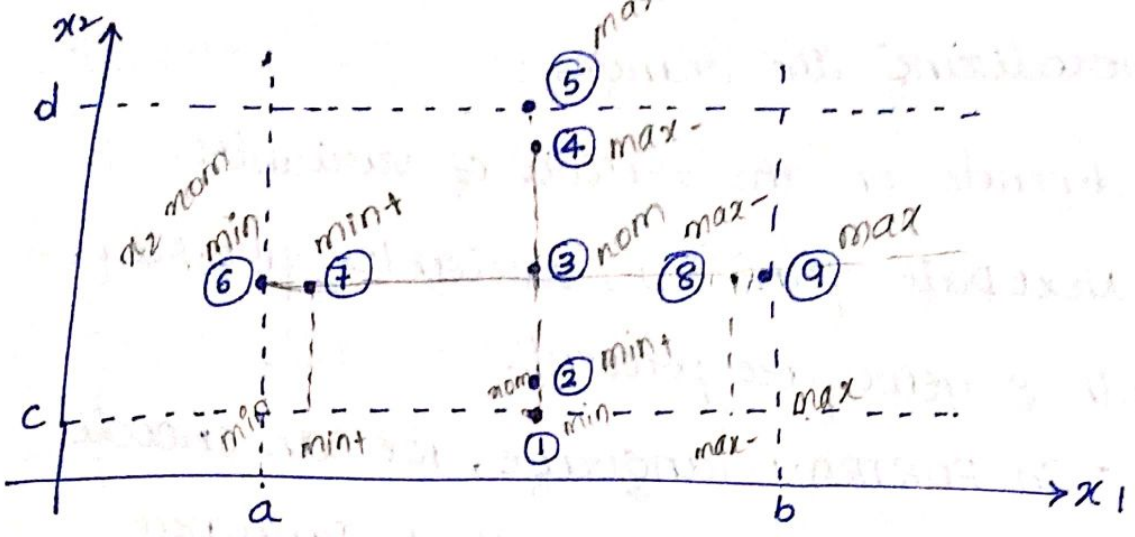


Fig: Boundary value test cases.

GENERALIZING BOUNDARY VALUE ANALYSIS

1) Generalizing the no. of variables:

- Generalizing the no. of variables is easy. Say, function 'F' has 'n' variables, we can hold all but one at the nominal values & let other variables assume min, min+, max, max-, nom for fun F of 'n' var.

BVA yeilds $4n+1$ unique test cases.

Ex: Refer fig. above, using two var. x_1 & x_2,

$\therefore$ No. of variables, $n = 2$ $\quad 4(n)+1 = 4(2)+1 = 9$

We get 9 test cases in the I/P space.

In the formula $4n+1$, (4) refers to the four values min, min+, max, max-.

'n' = no. of variables, x_1 & x_2.

'1' is the variable value in nominal position.

2. Generalizing the ranges :

- It depends on the nature of variables.

(i) In Next Date function, variables for Day, Month & year are present.

- In FORTRAN language, we can encode the value of month, so that January corresponds to 1, February to 2 & so on.

- In ADA or Pascal, which supports user-defined types, var. can be defined in enumerated type as { Jan, Feb, ..., Dec }

∴ The values of min, min+, max-, max are clearly known.

(ii) When a variable has discrete bounded values, as the variables in the commission problem, again nom, min, min+, max-, max are easily determined.

(iii) When there are no explicit bounds present, as in a A/c problem, artificial bounds have to be created. The lower bound (min) of side length is 1.

- For the upper bound we can have largest integer by using MAXINT or restrict upper limit to 200 or 2000, since it is

arbitary or our convenient.

(iv) BVA does not affect for Boolean variables. Th extreme values (min, max) are True & False. The values are not defined for the remainin 3 values of (nom, min+, max-).

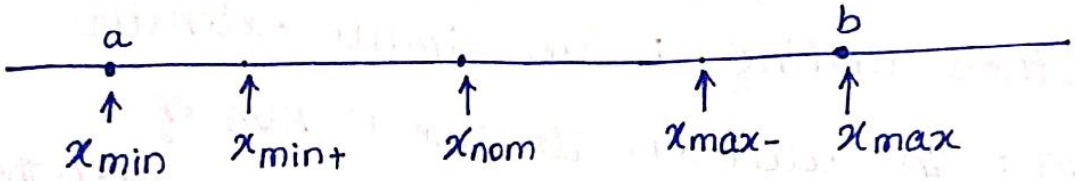


Fig: Inbound BV Testing

LIMITATIONS FOR BVA

1. BVA works well only when the prog./fun to be tested has several independent variables which rep. bounded physical quantities.

Physical quantities: when a variable refers to phy. quantity such as temp., pressure, air speed, angle of attack, load & so on, physical boundaries are important.

Ex: International airport at Bengaluru had to close of June 26, 2012, because of air temperature equal to 122°F.
Pilots were unable to set the instruments before take-off. Because Inst. accepts max. 120°F.

2. Considering the BVA test cases for NextDate, very little stress appears on Feb & on leap years.

- The real problem is dependencies exists among the day, month & year variables. BVA tries to define the var. which are totally independent.

ROBUSTNESS TESTING

- Robustness testing is an simple extension of BVA : In addition to the 5 BVA of a variable, we record what happens when the extreme values (min and max) are exceeded with a value slightly greater than the maximum (max+) & a value slightly less than the minimum (min −).
- The importance of robust testing lies with the expected outputs, & not inputs.
- when a phy. quantity exceeds its maximum :
  - Temperature = Flights are cancelled.
  - Capacity of an elevator = Nothing will happen.
  - Date : May 32 = Error msg is printed [max = 31, max+ = 32]
- The main values of RT is that it always concentrates on exception handling.

- In strongly typed lang., RT is not easy.

Ex: In Pascal, if a variable is defined within a range, values outside that range results in runtime errors which stops normal execution.

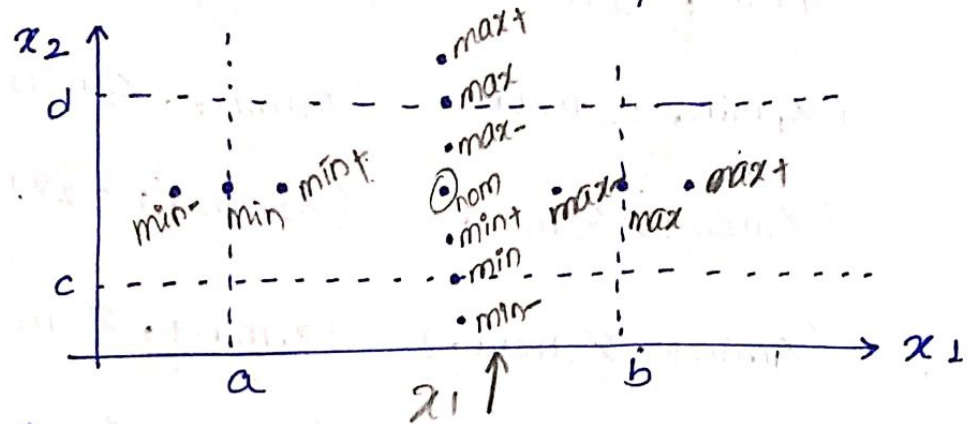


Fig: RT for a fun^n F of two variables x1 and x2.

WORST CASE TESTING

- Rejecting the critical assumption (single fault assumption) means that we are interested in what happens when more than one variable has an extreme values.
- In electronic circuit analysis, this is called "Worst case analysis", used in order to generate worst-case test cases.
- For each variable we start with 5 elements set that contains min, min+, nom; max-, max values The cartesian product of the sets are taken to generate test cases.
- Worst-case testing for a fun^n of 'n' variables generates $\boxed{5^n}$ test cases.

$x_1 = \{ min, min+, nom, max-, max \}$

$x_2 = \{ min, min+, nom, max-, max \}$

$x_1 * x_2 = \{ (x_1 min, x_2 min)$ ① $(x_1 min, x_2 min+)$ ②

$n = 2$ (x_1 and x_2)

$5^n = 5^2 = 25$

$(x_1 min, x_2 nom)$ ③ $(x_1 min, x_2 max-)$ ④

$(x_1 min, x_2 max)$ ⑤ $(x_1 min+, x_2 min)$ ⑥

$(x_1 min+, x_2 min+)$ ⑦ $(x_2 min+, x_2 nom)$ ⑧

$(x_1 min+, x_2 max-)$ ⑨ $(x_1 min+, x_2 max)$ ⑩

$(x_1 nom, x_2 min)$ ⑪ $(x_1 nom, x_2 min+)$ ⑫

$(x_1 nom, x_2 nom)$ ⑬ $(x_1 nom, x_2 max-)$ ⑭

$(x_1 nom, x_2 max)$ ⑮ $(x_1 max-, x_2 min)$ ⑯

$(x_1 max-, x_2 min+)$ ⑰ $(x_1 max-, x_2 nom)$ ⑱

$(x_1 max-, x_2 max-)$ ⑲ $(x_1 max-, x_2 max)$ ⑳

$(x_1 max, x_2 min)$ ㉑ $(x_1 max, x_2 min+)$ ㉒

$(x_1 max, x_2 nom)$ ㉓ $(x_1 max, x_2 max-)$ ㉔

$(x_1 max, x_2 max) \}$ ㉕

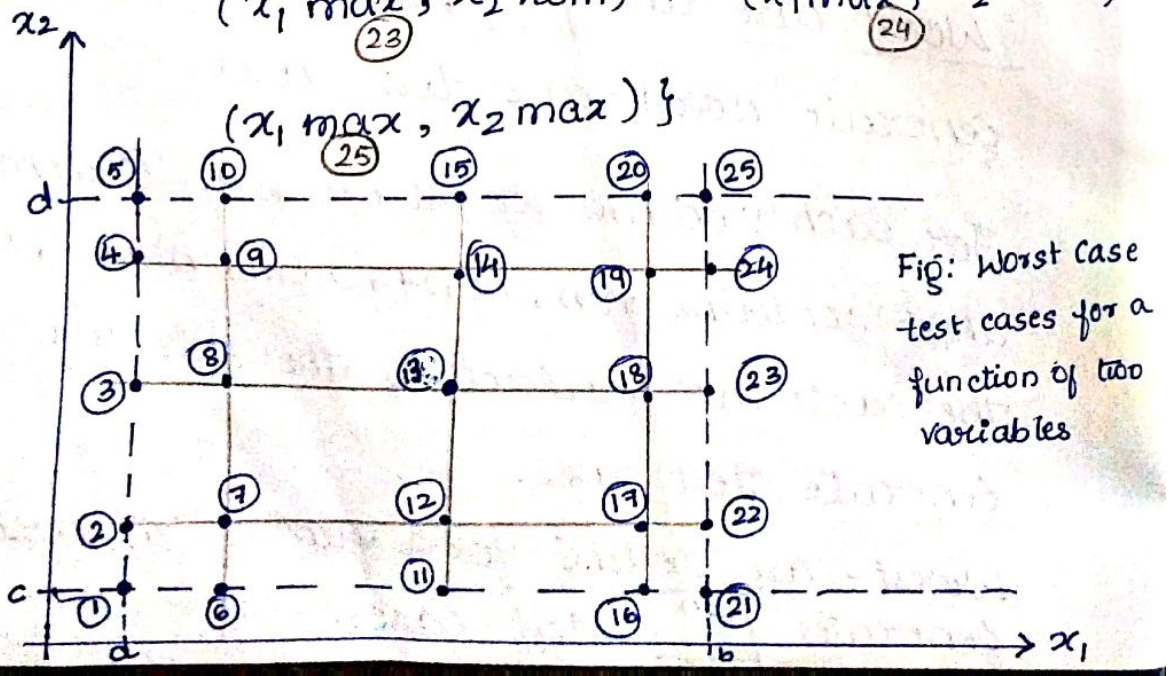
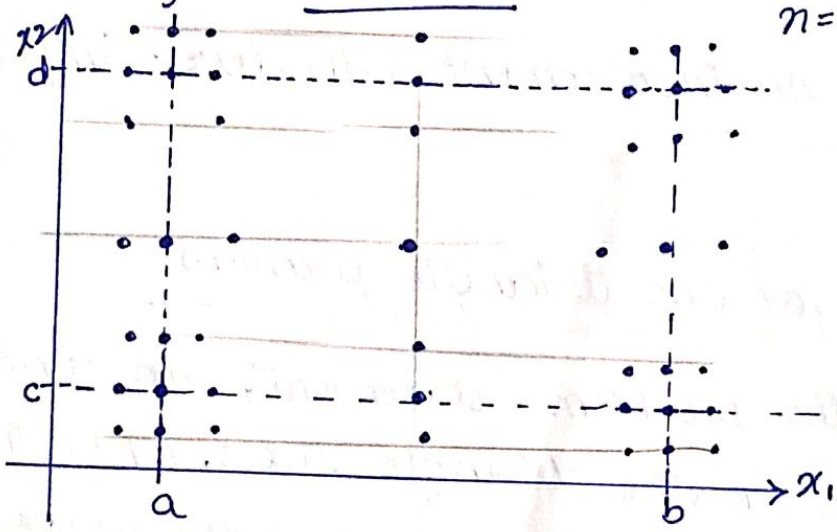


Fig: Worst case test cases for a function of two variables

- Generalization of worst case testing is similar to the generalization of BVA. Even the limitations are same
- WCT is more suitable for physical quantity variables.
- RT is more suitable for really paranoid testing. It involves the cartesian product of 7 element set resulting in 7^n test cases. $7^n = 7^2 = 49$ $n = 2 = x_1 \& x_2$



SPECIAL VALUE TESTING

- Well practiced form of functional testing.
- SVT occurs when a tester uses domain (area) knowledge, experience with similar prog. & inform" about "soft" spots" to devise test cases.
- SVT is also referred as "ad hoc testing" or "seat-of-the-pants testing" or "seat-of-the-skivet testing".
- It is described as 'Best engineering judgement' as it depends on experienced professionals.

Advantages of SUT :

1. Errors can be identified in less time.
2. Requires less effort.
3. Cost effective.

Disadvantages :

1. Requires experienced professionals with appropriate domain knowledge.
2. There is no involvement with developing team.

EXAMPLES

1. Test cases for the triangle problem

In the problem statement, no. conditions are specified on the triangle sides, other than being integers. The lower bound is 1 & the upper bound is 10. Upper bound is arbitary (as convinient). The table contains BV test cases using these ranges.

[NOTE: Test case no. 3, 8, 13 are identical in which two test cases can be ignored]

Boundary = 1 to 10

$\therefore$ min = 1

min+ = 2

nom = 5

max- = 9

max = 10

$4n+1 = 4(3)+1 = 13$

$n = 3$ (3 sides of Δ^{le})

$\therefore$ Only maximum 13 possible test cases can be generated.

Case	a	b	c	Expected output
1	5	5	1	Isoceles
2	5	5	2	Isoceles
3	5	5	5	Equilateral
4	5	5	9	Isoceles
5	5	5	10	Not a triangle
6	1	5	5	Isoceles
7	2	5	5	Isoceles
8	5	5	5	Equilateral
9	9	5	5	Isoceles
10	10	5	5	Not a Δ^{le}
11	5	1	5	Isoceles
12	5	2	5	Isoceles
13	5	5	5	Equilateral
14	5	9	5	Isoceles
15	5	10	5	Not a Δ^{le}

2) In case of Δ^{le} problem, the worst case analysis yeilds $5^n = 5^3 = 125$ test cases. Among 125 test cases 25 test cases are shown.

- This tables refers to just "one corner" of the input space cube.

Case	a	b	c	Expected Output
1	1	1	1	Equilateral
2	1	1	2	Not a Δ^{le}
3	1	1	5	Not a Δ^{le}
4	1	1	9	Not a Δ^{le}
5	1	1	10	Not a Δ^{le}
6	1	2	1	Not a Δ^{le}
7	1	2	2	Isoceles
8	1	2	5	Not a Δ^{le}
9	1	2	9	Not a Δ^{le}
10	1	2	10	Not a Δ^{le}
11	1	5	1	Not a Δ^{le}
12	1	5	2	Not a Δ^{le}
13	1	5	5	Isoceles
14	1	5	9	Not a Δ^{le}
15	1	5	10	Not a Δ^{le}
16	1	9	1	Not a Δ^{le}
17	1	9	2	Not a Δ^{le}
18	1	9	5	Not a Δ^{le}
19	1	9	9	Isoceles
20	1	9	10	Not a Δ^{le}
21	1	10	1	Not a Δ^{le}
22	1	10	2	Not a Δ^{le}
23	1	10	5	Not a Δ^{le}
24	1	10	9	Not a Δ^{le}
25	1	10	10	Isoceles

RANDOM TESTING

- Rather than always choosing the min, min+, nom, max-, max values of a bounded value variable, we can use a random number generator to pick test cases values. This avoids the bias (partiality) in testing.
- The no. of randomly generated test cases are derived from a VB application that picks values for a bounded variable $a \leq x \leq b$ as follows:

$$x = Int((b-a+1) * Rnd + a)$$

where the function Int returns the integer part of a floating point number & the fun° Rnd generates random numbers in the interval [0, 1]

- In the tables, the last line shows what % of the random test cases was generated for each column.

Random test cases for Triangle problem

Test Cases	Non Δ^{le}	Scalene	Isoceles	Equilateral
1289	663	593	32	1
15436	7696	7372	367	1
17091	8556	8164	367	1
2603	1284	1252	66	1
6475	3197	3122	155	1
5978	2998	2850	129	1
9008	4447	4353	207	1
Percentage	49.83%	47.87%	2.29%	0.01%

EQUIVALENCE CLASSES

- Equivalence classes form a partition of a set, where partition refers to a collection of mutually disjoint selbsets and the union of which is the entire set.
- 2 important implications are :
  a) The entire set provides a form of completeness.
  b) The disjointedness ensures a form of non-redundancy. Because the subsets are determined by an equivalence relation "the elements of a subset have something in common".
- The idea of equivalence class testing is to identify the test cases by using one element from each equivalence class. If the equivalence classes chosen carefully, this reduces the redundancy among test cases.

  Ex : In Δ^{le} problem (5, 5, 5) (6, 6, 6) (10, 10, 10) all these cases indicate equilateral.
- It depends on the choice of the equivalence relation that determines the classes, ie, first differentiate weak & strong equivalence class testing. Later this is compared with traditional form equivalence class testing.

- When a fun^n F with 2 variables x_1 and x_2 is implted as a program, the i/p variables x_1 & x_2 will have the foll. boundaries & intervals within boundaries:

$a \leq x_1 \leq d$ with intervals $[a,b)$, $[b,c)$, $[c,d]$

$e \leq x_2 \leq g$ with intervals $[e,f)$, $[f,g]$

- [ ] denotes closed interval endpoints
- ( ) denotes open interval endpoints.

1) WEAK NORMAL EQUIVALENCE CLASS TESTING:

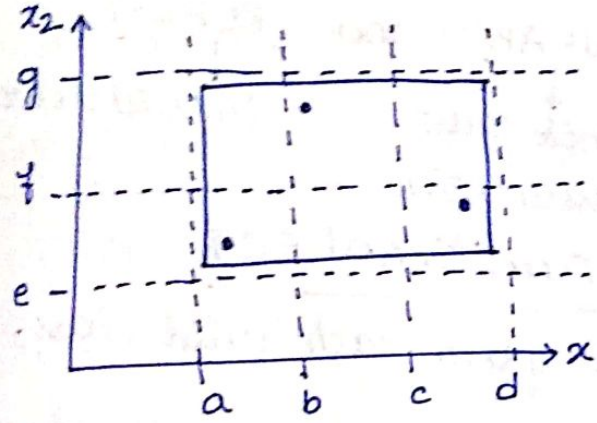


FIG: WNECT test cases:
$[a,b)$ $[b,c)$ $[c,d]$ $[e,f)$ $[f,g]$

- Take few valid i/p values.
- WNECT is done by using one variable from each equivalence class (interval) in a test case, as shown in test cases above.
- single fault assumption
- The no. of WECT cases are equal to classes in the partition with the largest no. of subsets.

2) STRONG NORMAL EQUIVALENCE CLASS TESTING

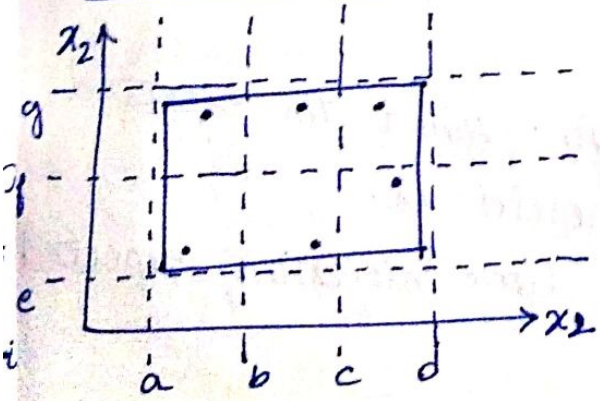
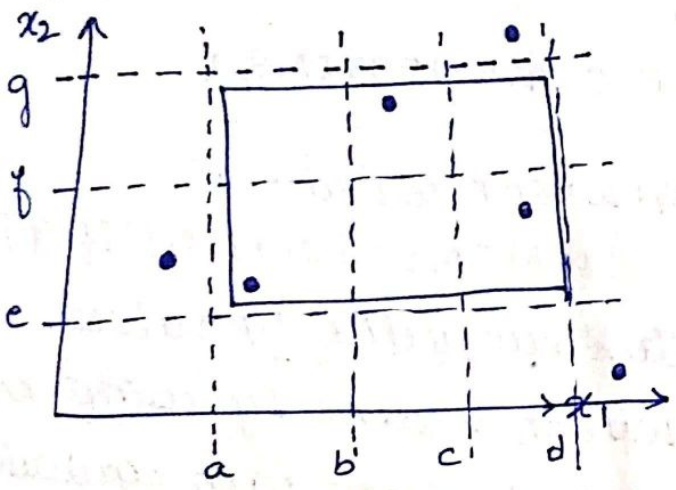


FIG: SNECT test cases: {(a,b)(e,f)} {(a,b)(f,g)} {(b,c)(e,f)} {(b,c)(f,g)} {(c,d)(e,f)} {(c,d)(f,g)}

- Take all valid input values.
- SNECT is based on multiple fault assumption, ∴ test cases from each element of the cartesian product of the equivalence classes as shown above.

- The cartesian product guarantees that we have a notion of completeness in 2 senses:
  a) ~~we cover~~ all equivalence classes are covered.
  b) Atleast one of each possible combination of inputs.

3) WEAK ROBUST EQUIVALENCE CLASS TESTING



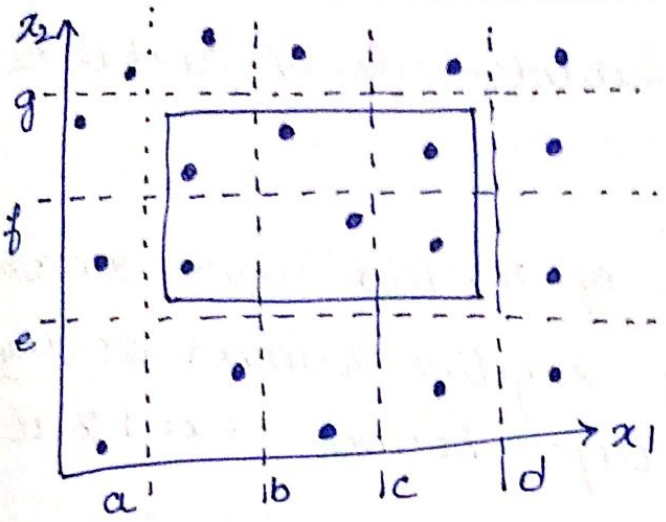
- Take few valid & few invalid input values.
- WEAK and ROBUST
  - WEAK → Single fault assumption
  - ROBUST → invalid values

⇒ Traditional ECT.

1) For valid i/p, use one value from each valid class, ie WNECT.
2) For invalid i/p, a test case will have one invalid value & the remaining values will all be valid. Thus a single failure should cause the test case to fail.

- Two problems:
  1) The specification does not define what the expected o/p for an invalid input should be.
     ∴ Testers spend a lot of time defining expect o/p for these cases.
  2) Strongly typed lang. eliminate the need of invalid inputs.

4> STRONG ROBUST EQUIVALENCE CLASS TESTING



- Take all invalid and all valid input values.
- strong part refers to multiple fault assumption.
- Robust part - invalid values.

- The test cases are generated from each element of the cartesian product of all equivalence classes.

EQUIVALENCE TEST CASES FOR TRIANGLE PROBLEM

In Δ^{le} problem, 4 possible o/p are as follows:

- Not a Δ^{le}
- Isoceles
- Equilateral
- Scalene

- These o/p can be used to identify o/p (range) EC as:

R1 = {<a,b,c> : The triangle with sides a, b, & c is equilateral}

R2 = {<a,b,c> : ——" —— " —— is isoceles}

R3 = {<a,b,c> : ——" —— " —— is scalene}

R4 = {<a,b,c> : ——" —— " —— do not form Δ^{le}}

1> The four WNEC test cases, chosen arbitarily from each class are:

TEST CASE	a	b	c	Expected o/p
WN1	5	5	5	Equilateral
WN2	2	2	3	Isoceles
WN3	3	4	5	Scalene
WN4	4	1	2	Not a Δ^{le}

2) SNEC test cases are identical to WNEC test cases. Because no invalid subintervals of variables a, b, c exist.

3) WREC test cases consists of invalid values for a, b, c which can be zero, any negetive number or any number greater than upper bound. Ex: 1 to 10

Test case	a	b	c	Expected Output
WR1	-1	5	5	Value of a not in range
WR2	5	-1	5	—"— b —"—
WR3	5	5	-1	—"— c —"—
WR4	11	5	5	value of a not in range
WR5	5	12	5	—"— b —"—
WR6	5	5	11	—"— c —"—

4) SREC test cases are as follows:

Test case	a	b	c	Expected output
SR1	-1	5	5	value of a not in range
SR2	5	-1	5	—"— b —"—
SR3	5	5	-1	—"— c —"—
SR4	-1	-1	5	value of a, b not in range
SR5	5	-1	-1	—"— b, c —"—
SR6	-1	5	-1	—"— a, c —"—
SR7	-1	-1	-1	value of a, b, c not in range

- If we place equivalence classes on i/p domain, no. of test cases are increased.

Ex: for Δ^{le} program, 3 integers a, b, c are:

- All of them can be equal
- Only 2 sides can be equal (3 possible ways)
- None are equal.

$D1 = \{ \langle a, b, c \rangle : a = b = c \}$

$D2 = \{ \langle a, b, c \rangle : a = b, a \neq c \}$

$D3 = \{ \langle a, b, c \rangle : a = c, a \neq b \}$

$D4 = \{ \langle a, b, c \rangle : b = c, b \neq a \}$

$D5 = \{ \langle a, b, c \rangle : a \neq b, b \neq c, a \neq c \}$

- The property of Δ^{le} can be applied on 3 integer values a, b, c to check whether they form a Δ^{le}.

Ex: $\langle 1, 4, 1 \rangle$ has 2 equal sides, but it cannot form a Δ^{le}.

$D6 = \{ \langle a, b, c \rangle : a \geq b + c \}$

$D7 = \{ \langle a, b, c \rangle : b \geq a + c \}$

$D8 = \{ \langle a, b, c \rangle : c \geq a + b \}$

- The above stmt can be even more precise by:

$D6' = \{ \langle a, b, c \rangle : a = b + c \}$

$D6'' = \{ \langle a, b, c \rangle : a > b + c \}$

$D7' = \{ \langle a, b, c \rangle : b = a + c \}$

$D7'' = \{ \langle a, b, c \rangle : b > a + c \}$

$D8' = \{ \langle a, b, c \rangle : a = b + c \}$

$D8'' = \{ \langle a, b, c \rangle : a > b + c \}$

EQUIVALENCE CLASS TEST CASES FOR THE COMMISSION PROBLEM

- The i/p domain of the commission problem is partitioned by the limits on locks, stocks & barrels.
- These equivalence classes are exactly identical to those which are identified by weak robust (traditional) equivalence class testing.
- The 1st class is the valid i/p & the other two are invalid.
- The valid classes of input variables are:

  $L1 = \{ locks : 1 \leq locks \leq 70 \}$

  $L2 = \{ locks = -1 \}$ (It is used to ctrl iteration, occurs as if (locks = = -1)

  $S1 = \{ Stocks : 1 \leq Stocks \leq 80 \}$

  $B1 = \{ Barrels : 1 \leq Barrels \leq 90 \}$

- The corresponding <u>invalid classes</u> of i/p variables are

  $L3 = \{ locks : locks = 0 \text{ or } locks < -1 \}$

  $L4 = \{ locks : locks > 70 \}$

  $S2 = \{ stocks : stocks < 1 \}$

  $S3 = \{ stocks : Stocks > 80 \}$

  $B2 = \{ barrels : barrels < 1 \}$

  $B3 = \{ barrels : barrels > 90 \}$

- There is exactly 1 WNECT and 1 SNECT. Because there are no invalid intervals of variables locks, stocks & barrels.
- The case locks = -1 terminates the iteration.

Case ID	Locks	Stocks	Barrels	Expected o/p
WN1 SN1	10	20	30	220

$10 \times 45 = 450$
$20 \times 30 = 600$
$30 \times 25 = 750$
Sales = 1800

$\therefore$ Commission = 1000 + 800
$\Rightarrow$ (10%) (15%)
= 100 + 120
= 220

We have 8 WRECT:

Case ID	Locks	Stocks	Barrels	Expected Output
WR1	10	10	10	100
WR2	-1	40	45	Program terminates
WR3	-2	40	45	Locks is not in range
WR4	71	40	45	" " "
WR5	35	-1	45	Stocks is not in range
WR6	35	81	45	" " "
WR7	35	40	-1	Barrels is not in range
WR8	35	40	91	" " "

2) Strong Robust ECT :

Case ID	Locks	Stocks	Barrels	Expected Output
SR1	-2	40	45	Value of locks not in range
SR2	35	-1	45	Value of stocks —"—
SR3	35	40	-2	Value of barrels —"—
SR4	-2	-1	45	Value of locks is not in range Value of stocks —"—
SR5	-2	40	-1	Value of locks not in range —"— barrels —"—
SR6	35	-1	-1	Value of stocks not in range Value of barrels —"—
SR7	-2	-1	-1	Value of stocks not in range Value of locks —"— Value of barrels —"—

Output Range Equivalence Class Test :

- Value of the variable 'Sales' depends on the no. of locks, stocks and barrels sold :

  Sales = 45 x locks + 30 x stocks + 25 x barrels

- Equivalence classes of 3 variables for commission prob are shown below & those equivalence classes depen on commission ranges :

  S1 = { < locks, stocks, barrels > : sales ≤ 1000 }

  S2 = { < locks, stocks, barrels > : 1000 < sales ≤ 1800 }

  S3 = { locks, stocks, barrels > : sales > 1800 }

Test case	Locks	Stocks	Barrels	Sales	Commission
OR1	5	5	5	500	50
OR2	15	15	15	1500	175
OR3	25	25	25	2500	360

DECISION TABLES

Decision tables are mainly used for:

- Representation of complex logical relationships.
- Analysis of ——— " ——— " ———
- Description of situations in which some actions are taken for some particular condition.
- Decision table is divided into 4 parts:

a) Stub: left most column
   → condition stub
   → Action stub

b) Entry: right ~~most~~ columns
   → Condition entries
   → Action entries

c) Condition 'C'

d) Action 'a'

- A column in the entry portion is a Rule. Rules indicate which actions are taken for the values (T or F) indicated in the condition portion of the rule.

Stub	Rule 1	Rule 2	Rule 3,4	Rule 5	Rule 6	Rule 7,8
c1	T	T	T	F	F	F
c2	T	T	F	T	T	F
c3	T	F	-	T	F	-
a1	X	X		X		
a2	X				X	
a3		X		X		
a4			X			X

- In this DT, when conditions c1, c2, c3 are all true, actions a1, a2 occur. When c1, c2 are true and c3 is false, action a1, a3 occur.
- The entry for c3 in the rule 3, where c1 is true & c2 is false is called as "don't care" entry.
- Don't care ← the condition is irrelevant (or) the condition does not apply (N/A)
- In the binary conditions (T/F, 0/1, Yes/No), the condition portion is a truth table rotated 90° This structures guarantee that every possible combination of condition values are considered.
- DT in which all the conditions are binary are called "Limited entry decision tables". If the conditions have several values (>2 values T or F) the resulting tables are called "Extended entry decision tables".

DECISION TABLE APPROACH FOR TRIANGLE PROGRAM

- To identify test cases, conditions are equal to inputs & actions as outputs.
- Sometimes; conditions = Equivalence classes of i/p
  Actions = Functional processing portions
  Rules = Test cases.
- When a rule is logically impossible, an action named "impossible" is added.

C1: a,b,c form a Δ^{le} ?	F	T	T	T	T	T	T	T	T
C2: a=b?	-	T	T	T	T	F	F	F	F
C3: b=c?	-	T	T	F	F	T	T	F	F
C4: c=a?	-	T	F	T	F	T	F	T	F
a1: Not a Δ^{le}	X								
a2: Scalene									X
a3: Isoceles					X		X	X	
a4: Equilateral		X							
a5: Impossible			X	X		X			

Fig: DT for Δ^{le} program.

- First column refers to that if the first condition C1 is false, then rest of the conditions can be directly ignored.

In rules 3,4,6, if 2 pairs of int are equal, by transitivity, the 3rd pair must be equal. ∴ these entries make Rules impossible.

c1: a<b+c ?	F	T	T	T	T	T	T	T	T	T	T	T	T	T
c2: b<a+c ?	-	F	T	T	T	T	T	T	T	T	T	T	T	T
c3: c<a+b ?	-	-	F	T	T	T	T	T	T	T	T	T	T	T
c4: a=b ?	-	-	-	T	T	T	T	T	T	F	F	F	F	F
c5: b=c ?	-	-	-	-	T	T	F	F	T	T	T	F	F	F
c6: c=a ?	-	-	-	-	T	F	T	F	T	F	T	F	T	F
Rule Count	2^5 = 32	2^4 = 16	2^3 = 8	1	1	1	1	1	1	1	1	1	1	1
a1: Not a Δ^{le}	X	X	X											
a2: Scalene														X
a3: Isocelese									X		X	X		
a4: Equilateral										X				
a5: Impossible											X	X		

Fig: Expanded DT for Δ^{le} prog. with rule counts.

- The condition c1: a, b, c form a Δ^{le}? is expanded to more detailed view of 3 inequalities of the Δ^{le} property. If any 1 of these fails, then 3 in do not form a Δ^{le}.

- For the limited entry decision tables, the no. of rules is 2^n, where n = no. of conditions.
- When don't care entries indicate condition is irrelevant, Rule Count is developed as follows:
  a) Each "don't care" entry in a rule, doubles the count of that rule.
  b) If there is no "don't care" entry then the rule count is 1.

← Sum of rule count = 64.

TEST CASES FOR TRIANGLE PROGRAM

← Referring to the DT, 11 test cases are generated.

Case ID	a	b	c	Expected Output
DT 1	4	1	2	Not a triangle
DT 2	1	4	2	Not a triangle
DT 3	1	2	4	Not a triangle
DT 4	5	5	5	Equilateral
DT 5	?	?	?	Impossible
DT 6	?	?	?	Impossible
DT 7	2	2	3	Isoceles
DT 8	?	?	?	Impossible
DT 9	2	3	2	Isoceles
DT 10	3	2	2	Isoceles
DT 11	3	4	5	Scalene

Bindushree S.C.
School of CIT, RU

SOFTWARE TESTING

A PERSPECTIVE ON TESTING

- **TESTING** - Testing is the process which is concerned with errors, faults and failures.

- **SOFTWARE TESTING** - S/w testing is the process to check whether the s/w meets the user requirements or not.

(or)

Software testing is the process of identifying errors in the s/w product.

- **REASONS FOR TESTING**

1. To check the quality.
2. To discover problems.

- **Basic definitions**

1. **Error** = Mistake, usually occurred by the interaction of humans.
2. **Bug** : When people make mistakes while coding is called BUG.
3. **Fault** = Defect. It is the result of an error or representation of an error, where representation refers to DFD, source code, narrative texts, hierarchy charts etc.

- Two types of faults are available :

a) Fault Commission occurs when we enter something into a representation that is incorrect.

Ex : Passwords

b) Fault Ommission occurs when we fail to enter correct information. It is more difficult to detect & resolve.

Ex : Mandatory fields.

4. Failure occurs when a fault executes.

a) Failures only occur in an executable representation usually source code or loaded object code.

b) This definition relates failures only to faults of commission.

5. Incident : when a failure occurs it may or may not be readily apparent to user.

- An incident is a symptom associated with a failure that alerts the user to the occurrance of failure.

6. Test : Testing is concerned with identifying errors, faults and failures.

- It is an act of exercising software with test cases.

- Two goals are :

a. To find failures.

b. Demonstrate correct execution.

7. Test cases : It is an identity associated with a program behavior. It has set of inputs and expected outputs.

- The main goal is to determine a set of test cases for the s/w to be tested.

- Test cases contains :

TEST CASE ID
PURPOSE
PRECONDITIONS
INPUTS
EXPECTED OUTPUTS
POST CONDITIONS
EXECUTION HISTORY
DATE RESULT VERSION RUN BY

a) Inputs

- Preconditions - circumstances that hold prior to test case execution.
- Actual Input - identified by some testing method.

b) Expected outputs

- Postcondition - conditions to be satisfied after getting output.
- Actual o/p - o/p to be obtained after executing s/w or program.

- The act of testing consists of establishing the necessary preconditions, providing the test case inputs, observing the outputs, comparing these with the expected o/p & then ensuring that the expected post conditions exist to determine whether the test possesed.

- TEST CASE ID - unique name for test case
- PURPOSE - what is need of testing (or) which functionality is being tested.
- PRECONDITIONS
- INPUTS
- EXPECTED OUTPUTS
- POST CONDITIONS
- EXECUTION HISTORY - contains information regarding who has executed the prog. & when. This is to be filled after executing the prog.
  - ↳ DATE - when s/w was executed using test case.
  - ↳ RESULT - Pass / Fail
  - ↳ VERSION - unique name given to s/w or prog
  - ↳ RUN BY - Person name who has tested

8) TESTING LIFE CYCLE

- First 3 stages (Req., Design, coding) where the occurrance of errors leads to fault is refferred as PUTTING BUGS IN.

- Testing phase refers to FINDING BUGS.
- The last 3 phases refers to PUTTING BUGS OUT.

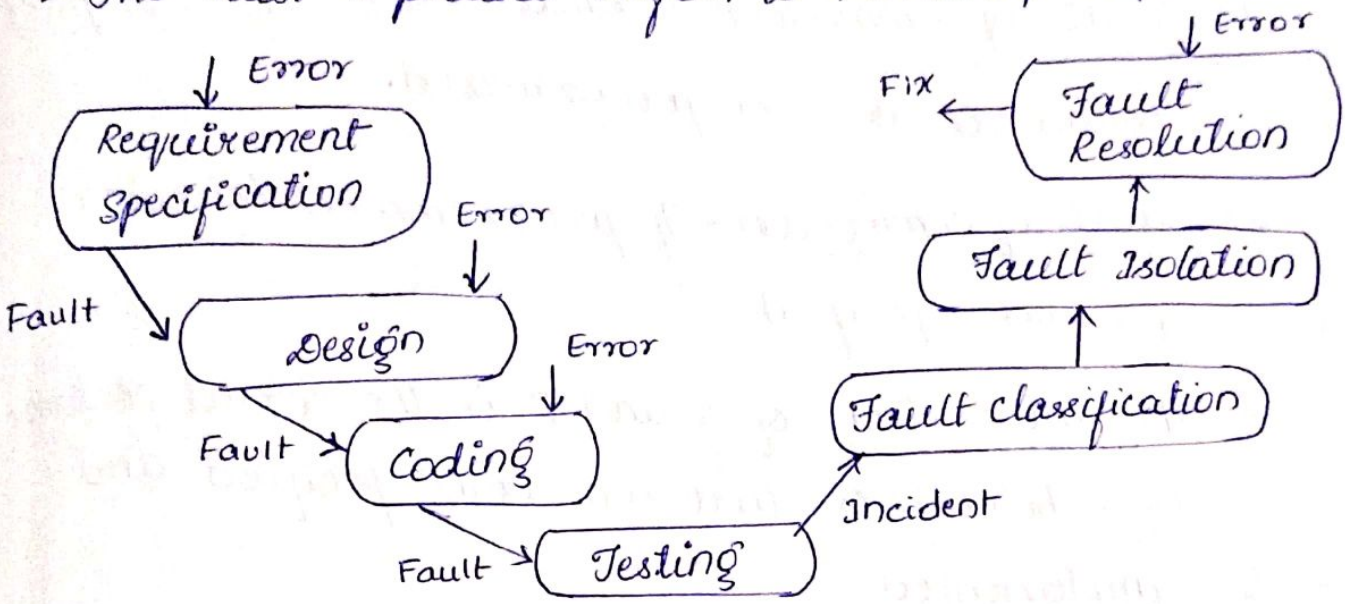
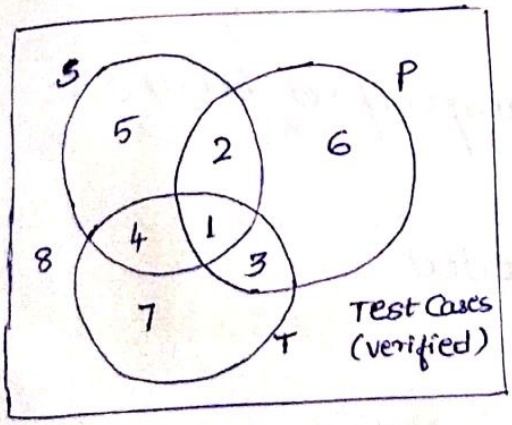


Fig: TESTING LIFE CYCLE

TESTING WITH THE VENN DIAGRAM

- Structural view - focuses on what it is. (White Box)
- Behavioral view - considers what it does. (Black Box)



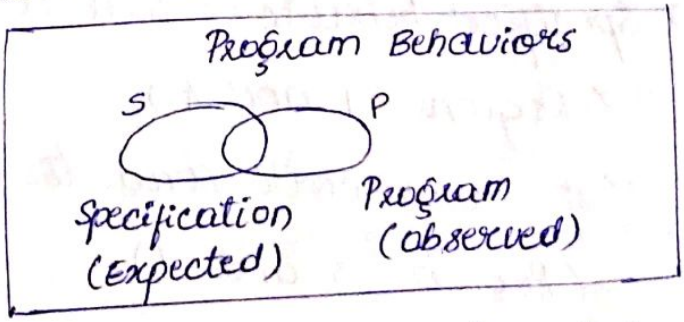


FIG: Specification & implemented program Behavior

- Consider a universe of prog. behaviors:
  S - set of specified behavior
  P - set of programmed behavior
- The venn diagram shows relationships among specified & programmed behaviors. Of all possible prog. the specified ones are in the circle labelled S & all actually prog. are in P.

- Two problems are identified :

1. Faults of ommission - what will happen if specified behaviors are not programmed.

2. Faults of commission - If programmed behavior are not specified.

- The intersection of S and P is the correct portion, ie., behaviors that are both specified and implemented.

- Relationships b/w S, P and T :

a) Specified behaviors that are not tested
(Regions 2 and 5)

b) Specified behaviors that are tested
(Regions 1 and 4)

c) Test cases corresponds to unspecified behavior
(Region 3 and 7)

d) Prog. behavior that are tested
(Region 1 and 3)

e) Prog. behavior that are not tested
(Region 2 and 6)

f) Test cases corresponding to unprog. behavior
(Region 4 and 7)

IDENTIFYING TEST CASES

- Two fundamental approaches are:

1. **Functional Testing**

- It is based on view, to check behavior of the prog. by giving i/p & evaluating o/p. I/p are taken from i/p domain (space which has all possible inputs).
- In this the tester mainly concentrates on behavior and the functionality is understood completely in terms of its inputs and outputs.
  ∴ It is called **Black Box** Testing.
- Implementation of a ~~bla~~ prog is not known.

- **Advantages**

1. They are independent of how software is implemented. If the implementation changes, the test cases are still useful.
2. Test case development can occur in parallel with the implementation, reducing the overall project development interval.

- **Disadvantages**

1. Significant redundancies may exist among test cases.
2. It cannot identify unspecified behaviores.

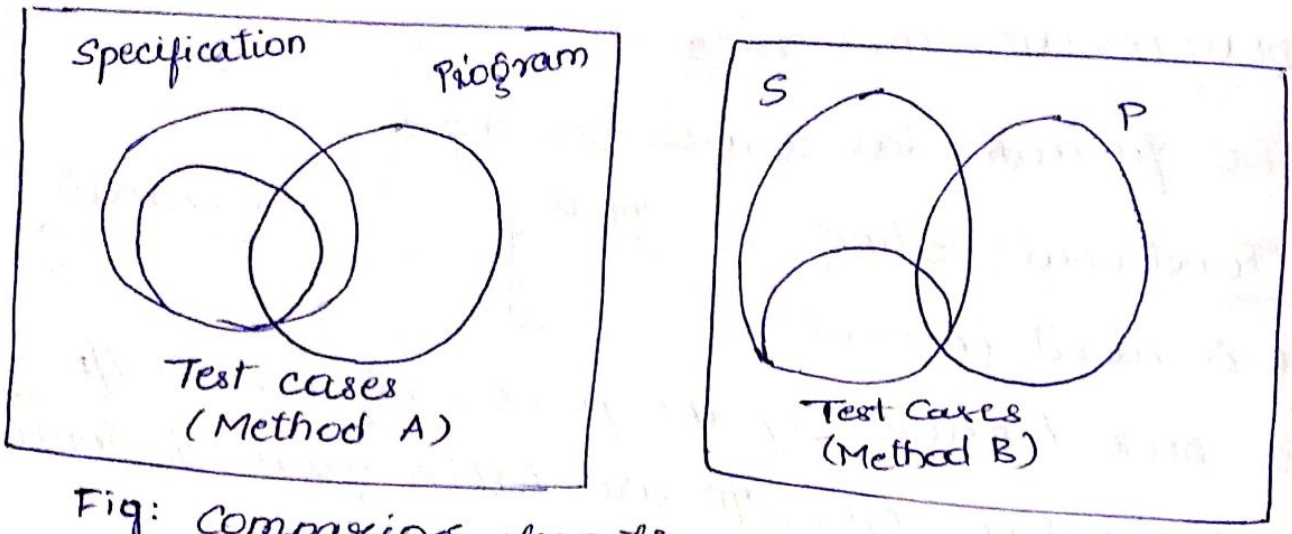


Fig: Comparing functional test case identification methods.

- Method A identifies a larger set of test cases than that of method B.
- In both methods, the set of test cases is completely residing within the set of specified behavior.

Note: In functional testing, the complete set of test cases are within set of specified behavior.

∴ The name **Behavioral testing**.

2. Structural Testing (Glass Box Testing)
   = White Box Testing
   = Clear Box Testing

- It refers to the implementation of the program and based on that the test cases are designed.
- The ability to "see inside" the black box allows the tester to identify test cases

based on how the s/m is actually implemented.

- It gives a proper meaning to testing management because of its strong theoretical basis.

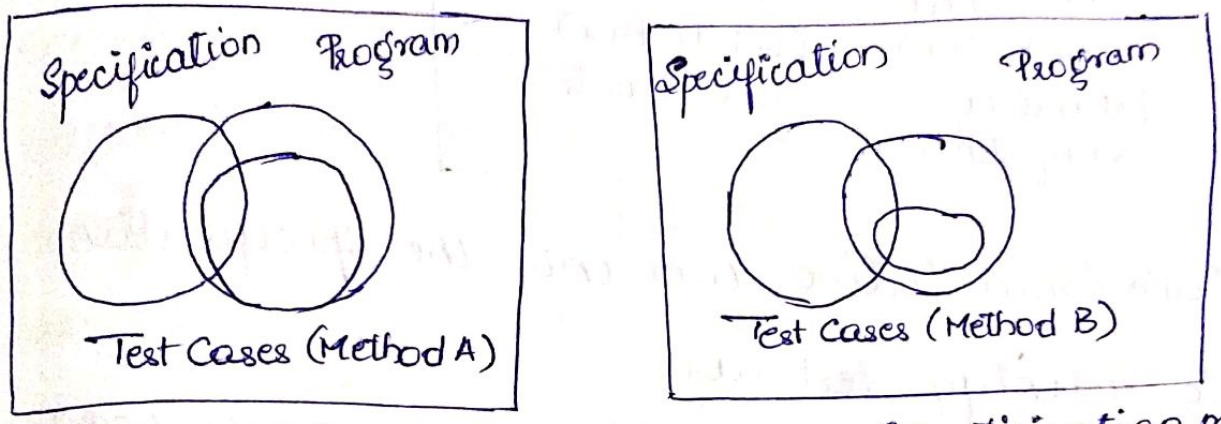


Fig: Comparing structured test case identification methods.

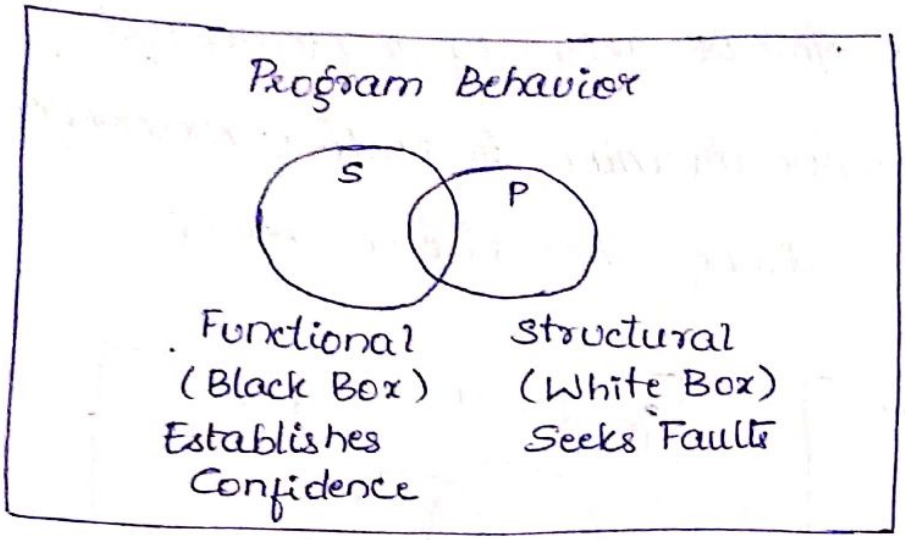
- Out of 2 methods Method A identifies a larger set of test cases than 'B', which lies in programmed behavior because this test is based on program.

→ For testing of any s/w both functional and structural testing are required, because

1) If all the specified behavior are not implemented, then structural test cases will never recognise this.

2) If Program implement unspecified behavior, then functional test case will not reveal any errors.

- When functional test cases are executed with structural, all problems can be recognized and resolved.



- Functional testing uses only the specification to identify test cases
- Structural testing uses program source code (implementation) for identification of test cases.
- Functional testing suffers from
  a) Redundancies
  b) Gaps
- When functional test cases are executed along with structural test coverage metrics, the problems are recognized & resolved.

ERROR AND FAULT TAXONOMIES

- Process refers to how we do something.
- Product is end result of process.
- S/W Quality Assurance (SQA) refers to improve process for developing product.

- SQA is more concerned with development process, while testing is concerned with identifying faults.

MILD	- Misspelled word
MODERATE	- Misleading or redundant information
ANNOYING	- Truncated names Ex: Bill = $0.00
DISTURBING	- Some transactions not processed
SERIOUS	- Lose a transaction
VERY SERIOUS	- Incorrect transaction execution
EXTREME	: Frequent "very serious" errors
INTOLERABLE	- Database corruption
CATASTROPHIC	- System shutdown
INFECTIOUS	- Shutdown that spreads to others

FIG: Faults classified based on severity

Input / output Faults

TYPE	INSTANCES
INPUT	Correct i/p not accepted
	Incorrect i/p accepted
	Description wrong or missing
	Parameters wrong or missing
Output	Wrong format
	wrong result
	Correct result at wrong time (too early/too late)
	Incomplete or missing result
	Spelling / Grammer
	spurious result

LOGIC FAULTS

Missing case(s)
Duplicate case(s)
Extreme condition neglected
Misinterpretation
Missing condition
Extraneous condition
Test of wrong variable
Incorrect loop iteration
Wrong operator

COMPUTATION FAULTS

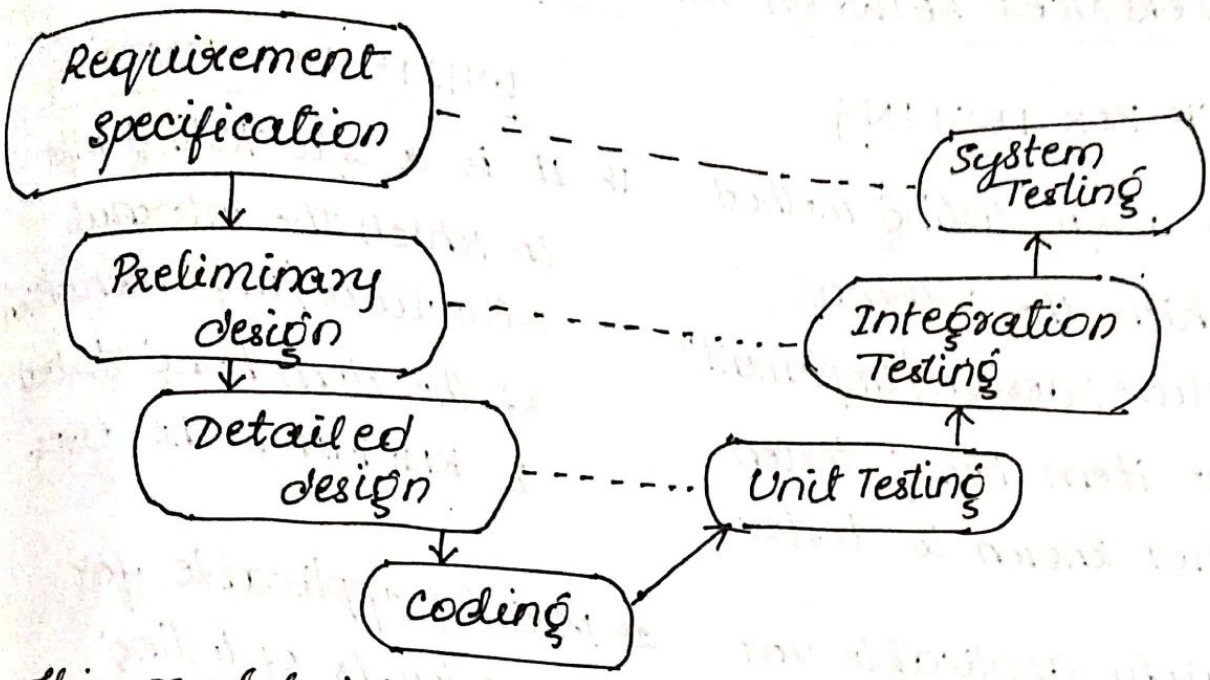
Incorrect algorithm
Missing computation
Incorrect operand
Incorrect operation
Parenthesis error
Wrong built in fun^n
Insufficient precision (round off, truncation)

DATA FAULTS

Incorrect initialization
Incorrect storage/access
Wrong flag/index value
Incorrect packing/unpacking
Wrong variable used
Wrong data reference
Scaling or units error
Incorrect data dimension
Incorrect subscript
Incorrect type
Incorrect data scope.
Sensor data out of limits
off by one
Inconsistent data

INTERFACE FAULTS

I/O Timing
Call to wrong procedure
Call to non existent procedure
Incompatiable types
Incorrect interrupt handling
Parameter mismatch (type, number)
Superfluous inclusion



- This model is useful for testing as a means of identifying distinct levels of testing & for clarifying the objectives that pertain to each level.
- The diagram emphasises on testing & design levels.
- In terms of functional testing the 3 levels of definition (specification, preliminary design & detailed design) correspond directly to 3 levels of testing (unit, integration & system).
- Structural testing suits at unit level and functional testing suits at system level.

DIFFERENCES BETWEEN BLACK BOX & WHITE BOX

BLACK BOX TESTING	WHITE BOX TESTING
1. It is a s/w testing method in which the internal structure / design / implement^n of the item being tested is NOT known to tester.	1. It is a s/w testing m/d in which the internal structure / implementation of the item being tested is KNOWN to the users.
2. Mainly applicable for higher levels of testing s/m and Acceptance test.	2. Mainly applicable for lower levels of testing Unit & Integration test.
3. This type of test is carried out by testers.	3. Test is carried out by s/w developers.
4. Programming knowledge is not required to carry out Black Box testing.	4. Prog. language is required to carry out White Box testing.
5. Implementation knowledge is NOT required to carry out this test.	5. Implementation knowledge is required.
6. Black Box test is referred to as functional test or external testing.	6. White box is referred to as Structural test or internal testing.

7. This testing primarily concentrates on functionality of s/m under test.	7. The primary concern is the source program code of the s/m under test.
8. Main aim is to check what functionality is being performed by s/m under test.	8. Main aim is to check how s/m is performing.
9. Black-Box testing can be started based on requirement specification documents.	9. This test can be started based on detailed design documents.

SOFTWARE TESTING

- It is a process of executing a program or an application with the intent of finding the s/w bugs.
- It is the process of validating & verifying that a s/w prog. or application or product.
- It helps in judging quality of product and also discover problems.

SOFTWARE DEVELOPMENT LIFE CYCLES (SDLC)

- Testing in SDLC helps to prove that all the s/w requirements are always implemented correctly or not.
- Testing helps in identifying defects & ensuring that testing are addressed before s/w development.

- If any defects is discovered & fixed after development, the correction cost is higher than the cost of fixing at earlier stages of development.
- Testing in SDLC demonstrates that s/w always appears to be specification & performance requirements are met.
- Whenever several s/m are developed in different components, different levels of testing help to verify proper integration or interaction of all components to rest of all the s/m.
- Testing always improves the quality of product and project by discovering bugs early in the s/w.

GENERALISED PSEUDOCODE

- It provides a platform to express prog. source code in a "language neutral" way.
- It has 2 levels: Unit & program components.
- Following example represents for unit testing:

Language Element	Generalized Pseudocode Construct
Comment	<text>
DS declaration	Type <type name> <list of field descriptions> End <type name>
Data declaration	Dim <variable> As <type>
Assignment stmt	<variable> = <expression>
Input	Input (<variable list>)
Output	Output (<variable list>)
Condition	<expression> <relational operator> <expression>
Compound condition	<condition> <logical connective> <condition>
Sequence	Stmts in sequential order
Simple selection	If <condition> Then <thenclause> End IF
Selection	If <condition>
Multiple selection	Case <variable> of case 1 : <predicate> <case clause> .... Case n : <predicate> <case clause> End case
Counter-Controlled repitition	For <counter> = <start> to <end>
Pretest repetition	While <condition> ... End while
Posttest repetition	Do .. until <condition>

Procedure definition (similarly for functions & O-O m/d)	<procedure name> (Input: <list of variables>; Output: <list of variables>)
Inter unit communication	call <procedure name> (<list of variables>; <list of var>)
class/object definition	<name> (<attribute list>) <method list>, <body> End <name>
object creation	Instantiate <class name> <object name> (list of attribute values)
object destruction	Delete <class name> <obj name>
Program	Program <program name>

THE TRIANGLE PROBLEM

PROBLEM STATEMENT

Simple version : The triangle prog. accepts 3 integers a, b and c as input which refers to the sides of the triangle. The o/p is expected to be determined as which type of Δ^{le} based on three sides: Equilateral, Isoceles, Scalene.

Improved version : The Δ^{le} prog. accepts 3 int a, b, c as i/p. These represents the sides of Δ^{le}. The 3 integers a, b, c must satisfy following conditions :

C1 : $1 \leq a \leq 200$

C2 : $1 \leq b \leq 200$

C3 : $1 \leq c \leq 200$

C4 : $a < b + c$

C5 : $b < a + c$

C6 : $c < a + b$

- The o/p of the prog. is the type of Δ^{le} determined by 3 sides : Equilateral, Isoceles, scalene or Not a Δ^{le}.
- If an i/p value fails any of conditions C1, C2, C3 the prog. returns an o/p msg. Ex: "value of b is not in the range of permitted values".
- If values of a, b, c satisfy C1, C2, C3 conditions one of the 4 mutually exclusive o/p are given:

1) $a = b = c$, Prog. o/p = equilateral

2) $a = b$ or $b = c$ or $a = c$, it is isoceles (exactly 1 pair equal)

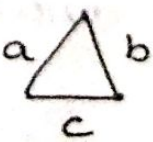
3) $a \neq b \neq c$ (no pair are equal), it is scalene.

4) If C4, C5, C6 fails, then it is not a triangle.

- The sum of any pair of sides must be strictly greater than the third side.

TRADITIONAL IMPLEMENTATION OF TRIANGLE PROBLEM

```
Program Triangle
Dim a,b,c matth as integer
Output ("Enter 3 integers which are sides of a Δle")
Input (a, b, c)
Output ("Side A is ", a)
```



```
Output ("Side B is ", b)
Output ("Side C is ", c)
Match = 0
if a = b
Then match = match + 1
end if
if a = c
Then match = match + 2
endif
if b = c
then match = match + 3
endif
If match = 0
Then if (a + b) <= c
Then output ("Not a triangle")
else if (b + c) <= a
Then output ("Not a triangle")
else if (a + c) <= b
Then output ("Not a triangle")
else output ("Scalene")
end if
endif
endif
```

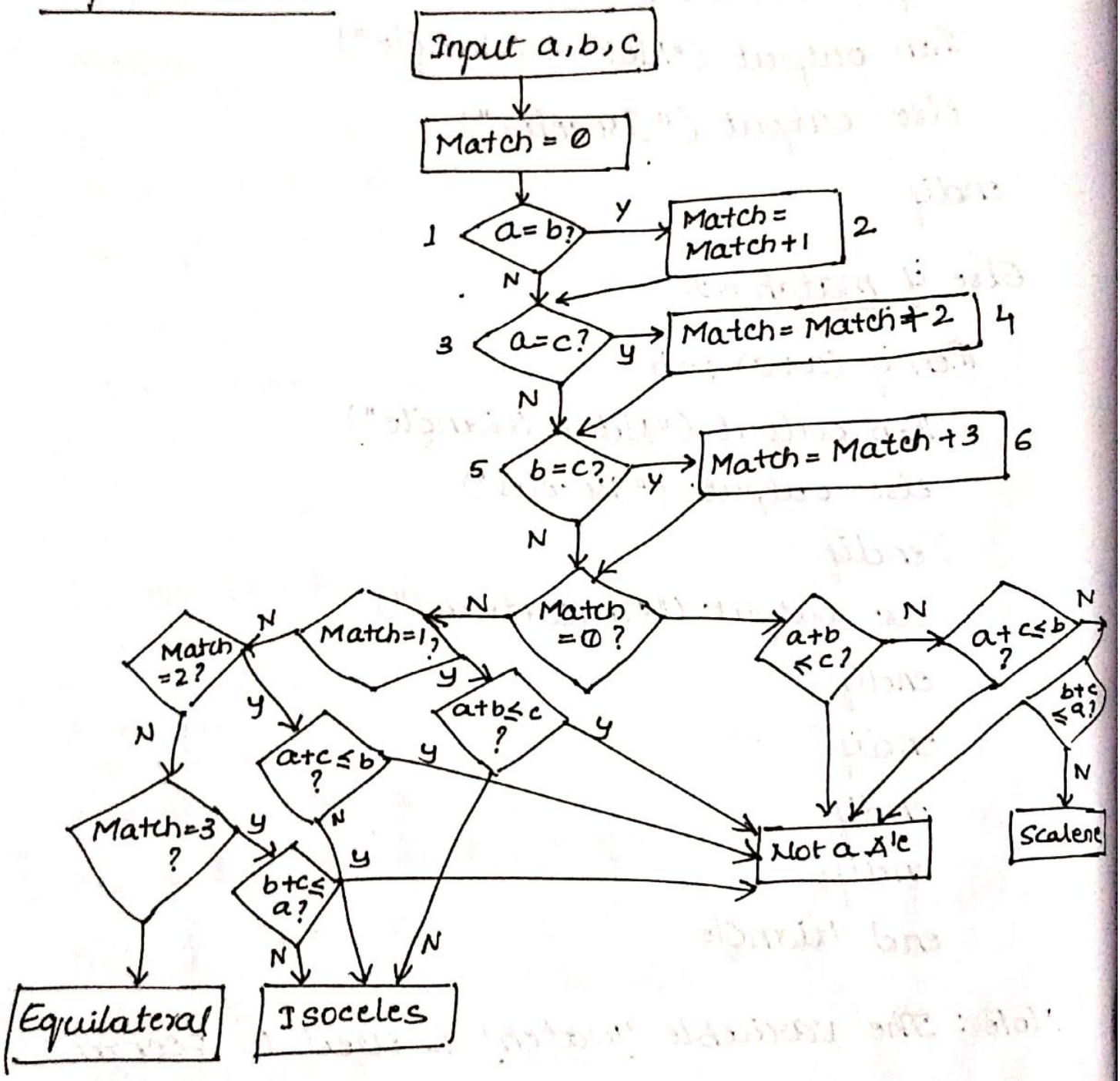
```
Else if match = 1
Then if (a+c) <= b
  Then output ("Not a triangle")
  else output ("Isoceles")
endif
Else if match = 2
  Then if (a+c) <= b
    Then output ("Not a triangle")
    else output ("Isoceles")
    endif
    else output ("Equilateral")
    endif
    endif
    endif
    endif
    end triangle
```

Note: The variable 'match' is used to record equality among pairs of sides.

Dim = declares and allocates storage space for one or more variables

[<attributelist>] [accessmodifier] [[shared] [shadows] | [Static]] [ReadOnly] Dim [withEvents] variable list

FLOWCHART for the traditional triangle problem implementation :



STRUCTURAL PROGRAMMING VERSION OF SIMPLER SPECIFICATION

Dim :a, b, c as integer

Dim is a triangle as Boolean

Step 1 : Get Input

Output : ("Enter 3 integers which are sides of a Δle")

Input (a, b, c)

Output ("Side A is ", a) Output ("Side B is ", b)

Output ("Side C is ", c)

Step 2 : Is a triangle ?

If (a < b+c) AND (b< a+c) AND (c < a+b)

Then Is A triangle = True

Else Is A triangle = False

End If

Step 3 : Determine the triangle type

If Is a triangle

Then If (a=b) AND (b=c)

Then output ("Equilateral")

Else If (a ≠ b) AND (a ≠ c) AND (b ≠ c)

Then output ("Scalene")

Else output ("Isoceles")

End If

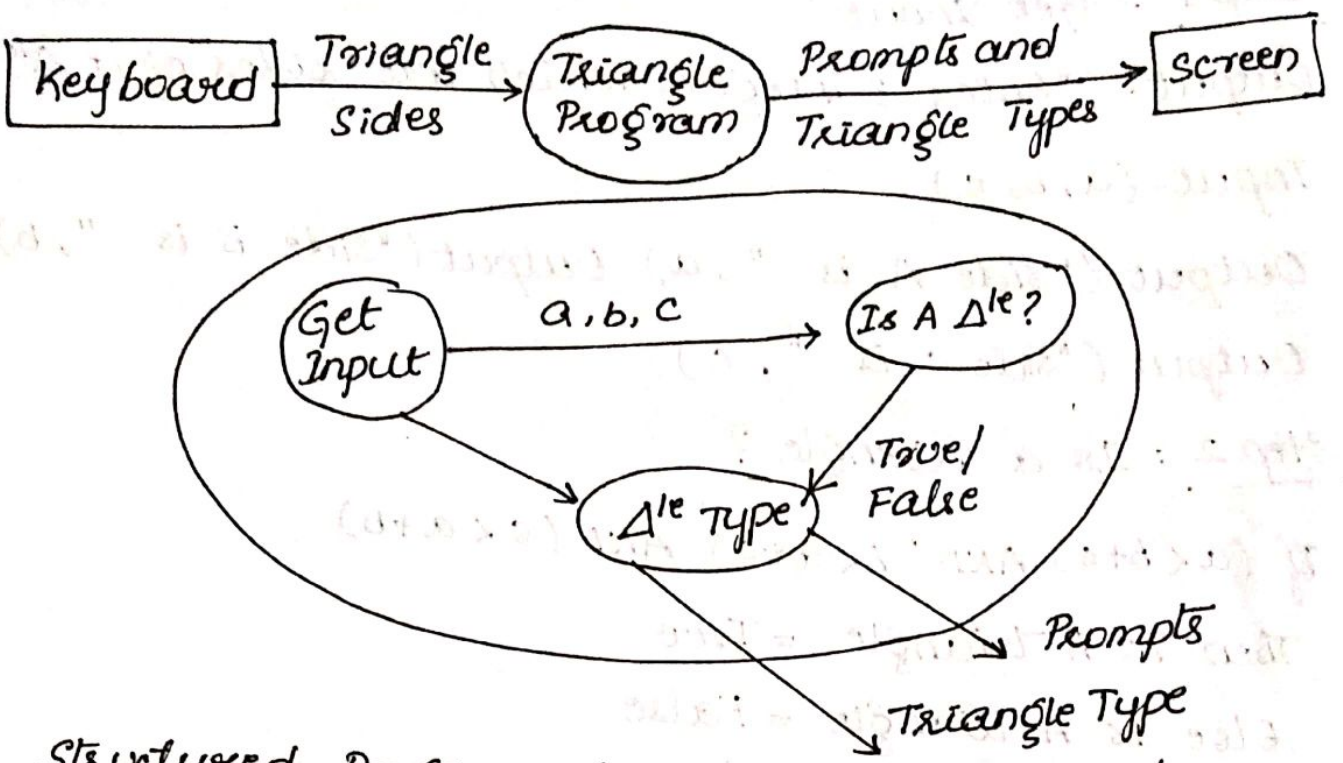
End If

Else output ("Not a triangle")

End If

Dataflow diagram for a structural triangle

Program implementation :



Structured Programming version of simpler specification

Dim a, b, c As Integer

Dim Is A Triangle as Boolean

Step 1 : Get Input

Output ("Enter 3 Integers which are sides of a Δle")

Input (a, b, c)

Output ("Side A is ", a)

Output ("Side B is ", b)

Output ("Side C is ", c)

Step 2 : Is A Triangle

If (a < b+c) AND (b < a+c) AND (c < a+b)

Then Is A Triangle = True

Else Is A Triangle = False

End If

Step 3: Determine Triangle Type

```
If Is a Triangle
Then If (a = b) AND (b = c)
Then output ("Equilateral")
Else If (a ≠ b) AND (a ≠ c) AND (b ≠ c)
Then output ("Scalene")
Else output ("Isoceles")
 End If
 End If
Else output ("Not a triangle")

End If
```

Structured Programming version of Improved Specification

```
Dim a, b, c as Integer
Dim c1, c2, c3 Is a Triangle as Boolean
```

Step 1 : Input

```
Do
Output ("Enter 3 integers for the sides of Δle")
Input (a, b, c)
C1 = (1 <= a) AND (a <= 200)
C2 = (1 <= b) AND (b <= 200)
C3 = (1 <= c) AND (c <= 200)
```

```
If NOT (c1)
Then output ("Value of a is not in range")
End If

If NOT (c2)
Then output ("Value of b is not in range")
End If

If NOT (c3)
Then output ("Value of c is not in range")
End If

Until C1 AND C2 AND C3
Output ("Side A is ", a)
Output ("Side B is ", b)
Output ("Side C is ", c)
```

Step 2 : Is A Triangle ?

```
If (a < (b+c)) AND (b < (a+c)) AND (c < (a+b))
Then Is A Triangle = True
Else Is A Triangle = False
End If
```

Step 3 : Determine Triangle Type

```
If Is A Triangle
Then If (a = b) AND (b = c)
  Then output ("Equilateral")
  Else if (a ≠ b) AND (a ≠ c) AND (b ≠ c)
   Then output ("Scalene")
   Else output ("Isoceles")
  EndIf
 End If
 Else output ("Not a Triangle")
 End If
```

COMMISSION PROBLEM

PROBLEM STATEMENT

A rifle salesperson sold rifle locks, stocks, and barrels made by a gunsmith

Price of a lock is $45

Price of a stock is $ 30

Price of a barrels is $ 25

The salesperson have to sell at least 1 complete rifle per month and product limits are 70 locks, 80 stocks and 90 barrels.

After each town visit, the salesperson sends a msg to gunsmith with the no. of stocks, locks and barrels sold in that town.

At the end of the month, the salesperson sends a very short telegram showing

-1 locks sold.

Then the gunsmith will come to know that the sales for the month is complete and will compute the salesperson's commission as follows:

10% on sales upto $ 1000
15% on the next $ 800
20% on any sales in excess of $ 1800

The commission program is designed to produce a monthly sales report which refers to total no. of locks, stocks & barrels sold, total sale and finally the commission.

1. The i/p data
2. Sales calculation
3. Commission calculation

Implementation

```
Program commission (input, output)
Dim Locks, Stocks, Barrels As Integer
Dim Lockprice, Stockprice, Barrelprice As Real
Dim TotalLocks, TotalStocks, TotalBarrels As Integer
Dim LockSales, StockSales, BarrelSales As Real
Dim Sales, Commission As Real

Lockprice = 45.0
Stockprice = 30.0
Barrelprice = 25.0
TotalLocks = 0
TotalStocks = 0
TotalBarrels = 0
```

```
Input (Locks)
While Not (Locks = -1)
/* Input device uses -1 to mark end of data */
Input (Stocks, Barrels)
TotalLocks = TotalLocks + Locks
TotalStocks = TotalStocks + Stocks
TotalBarrels = TotalBarrels + Barrels
Input (Locks)
End while
Output ("Locks sold: ", TotalLocks)
Output ("Stocks sold: ", TotalStocks)
Output ("Barrels sold: ", TotalBarrels)
LockSales = Lockprice * TotalLocks
StockSales = stockprice * TotalStocks
BarrelSales = Barrelprice * TotalBarrels
Sales = LockSales + StockSales + BarrelSales
If (sales > 1800.0)
Then comm = 0.10 * 1000.0
     comm = comm + 0.15 * 800
     comm = comm + 0.20 * (Sales - 1800)
Else If (sales > 1000.0)
     comm = 0.10 * 1000
```

```
comm = comm + 0.15 * (sales - 1000.0)
Else
comm = 0.10 * sales

End If
End If
Output ("Commission is $ ", commission)
End Commission
```