UNIT – 3

Course (Unit-3) Objectives:

- Make the student familiar with white box testing and different methods in it
- Get the student acquainted with Basis path testing and its methods
- Application of WBT by using graph matrices
- Understand the process of loop testing and data flow testing
- Make the student recognize and use mutation testing, both in theory and lab
- Make the student to understand and apply all methods of static testing

1. White box testing (WBT): Also known as glass box testing, this method studies the entire design, structure and code of the software. It is also known as structural or development testing.
2. Need of WBT: The reasons for the need (requirement) of WBT are given below:
   (a) WBT is used for testing a program at its initial stage. BBT is the second stage testing. Note that though BBT test cases can be designed earlier than WBT test cases, they can be executed only after WBT. Hence WBT is not an alternative but essential stage.
   (b) WBT is supportive to BBT; there exist some bugs that can be revealed only through WBT but not BBT.
   (c) We MUST execute WBT test cases to reveal bugs that have been transported from earlier phases of SDLC.
   (d) Some logical paths that are not used frequently can also be tested by WBT.
   (e) WBT can also find typographical errors which go unobserved in BBT.
3. Logic Coverage Criteria: WBT considers the program code and the test cases are designed based on the logic of the program. Every part of the logic is covered in WBT test cases.
4. Statement Coverage: It is assumed that if all the statements are executed at least once, every bug will be notified.

   Ex: scanf("%d", &x); scanf ("%d", &y);

```
 while(x!=y)
{
 if (x>y)
     x=x-y;
 else
     y=y-x;
 }
printf("x=%d",x);printf("y=%d",y);
```

   To cover all these statements, the test cases are:
   T1: x=y=n, where n is any number
   T2: x=n,y=m where n and m are different numbers

   T1 skips the 'while' loop. T2 executes the loop but every statement inside the loop is not executed. For this, two more test cases are added.

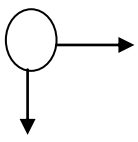
T3: x>y T4: x<y

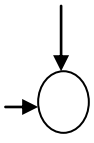
Though all statements are now covered, the logic of the program is not yet under testing. It can be concluded that statement coverage is necessary but not sufficient.

5. Decision or Branch Coverage: Branch coverage states that each decision considers all possible outcomes at least once. Considering the previous example, the test cases here must consider the outcomes of both 'while' and 'if' statements.
   T1: x=y; T2: x!=y; T3: x<y; T4: x>y
6. Condition Coverage: This states that each condition in a decision takes on all possible outcomes at least once.
   Ex: while((i≤5) && (j>COUNT)). Two conditions exist here. The test cases can be:
   T1: i≤5, j>count T2: i<5, j>count
7. Condition coverage in a decision doesn't mean that the decision also has been covered. If the decision "if(A&&B)" is being tested, the condition coverage would allow to write two test cases.
   T1: A is true, B is false; T2: A is false, B is true
   Note that these test cases have no effect on the 'then' clause of the 'if' statement. So this requires sufficient test cases such that each condition in a decision takes on all possible outcomes at least once and each point of entry is invoked (called) at least once.
8. Multiple Condition Coverage: It requires that sufficient test cases should be written so that all possible combinations of condition outcomes in each decision and all points of entry are invoked at least once. The following types of test cases can exist:

	A	B
TC1	T	T
TC2	F	F
TC3	T	F
TC4	F	T

9. Basis Path Testing: This technique is based on the control structure of the program. It is the method of selecting the paths that provide a basis set of execution paths through the program. BPT is used to identify and utilize those paths that cover maximum logic – not all the paths.
10. Guidelines for effectiveness of path testing:
    (1) Path testing (PT) is based on control structure of the program and a flow graph is prepared to depict the same.
    (2) PT requires complete knowledge of the program structure.
    (3) PT is mainly used by the developer.
    (4) The effectiveness of PT is reduced with the increase in the size of the software.
    (5) Choose enough paths such that maximum logic of the program is covered.
11. Control Flow Graph (CFG): It is a graphical representation of the control structure of a program.
    V-> Vertices; E-> Edges/links; N-> nodes

Decision node: 

Junction Node: 

Region -> Area bounded by edges and nodes

When counting the regions, the area outside the graph is also considered as a region.

12. CFG Notations:

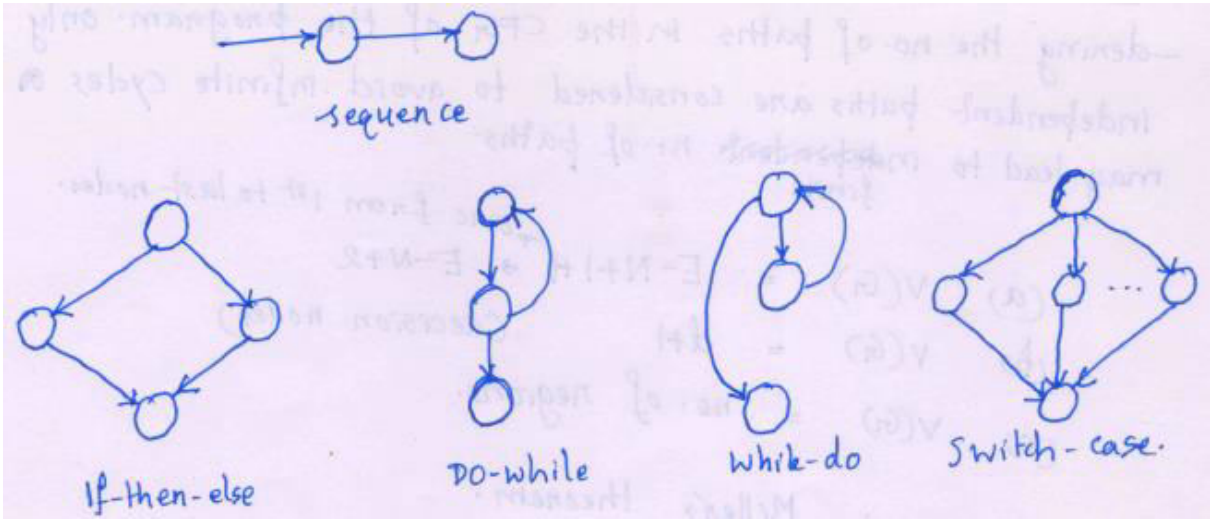


Fig. 3.1: CFG Notations

13. Path Testing Terminology:
    - Path: It is a sequence of instructions/statements starts at an entry, junction or decision and at another junction, decision or exit.
    - Segment: A path consists of segments where the smallest segment is a single link.
    - Path Segment: It is succession of consecutive links that belong to the same path.
    - Length of a path: It is measured by the number of links (or sometimes by no. of nodes) in it; not by the no. of statements executed.
    - Independent Path: It is any path through the graph that introduces at least one new set of processing statements or new conditions. It must move along at least one edge that has not been traversed before the path was defined.

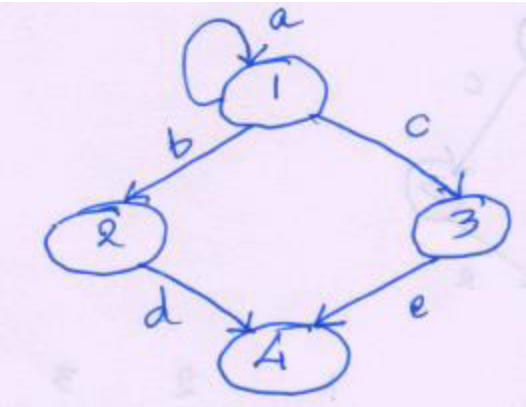
14. Cyclomatic Complexity (CC): Complexity is measured by considering the no. of paths in the CFG of the program. Only independent paths are considered to avoid infinite cycles.
    (a) V(G) = E-N+2
    (b) V(G) = R
    (c) V(G) = P+1 (predicates) [Miller's Theorem]

15. Applications of Path Testing:
    (a) Through Testing/More Coverage: It gives us the no. of test cases that are considered as important. CC also provides more coverage of paths.
    (b) Unit Testing: Path testing is mainly used by in structural testing of a module. In unit testing, it is more useful since each outcome of a decision is considered.

(c) Integration Testing: Since one module may call one or more other modules, interface errors are likely to surface here. Path testing analyzes all the possible paths on the interfaces and explores all the errors.
(d) Maintenance Testing: PT is also necessary for the modified version(s) of the software and to re-test the modules or interfaces.
(e) Testing effort is proportional to the complexity of the software. PT takes care of this, and obtains the no. of paths to be tested by CC.
(f) Basis Path testing concentrates more on error-prone software. The CC (or the no. of paths) signifies that the testing effort is only on the error-prone part of the software, thus minimizing the testing effort.

16. Graph Matrices: Graph matrix is a square matrix whose rows and columns are equal to the no. of nodes in the flow graph. Each row and column identifies a particular node and the matrix entries are the connections between the nodes. Each cell is a link between the two concerned nodes.
    NOTE: If there is a link between node ‘a’ and node ‘b’, it doesn’t mean that the opposite is also true.

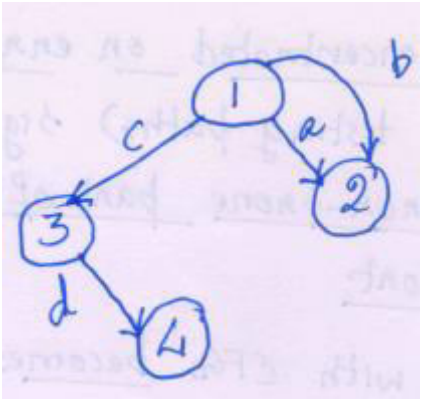
17. Ex:



Graph Matrix:

	1	2	3	4
1	a	b	c	
2				d
3				e
4				

18. Ex:



Graph Matrix:

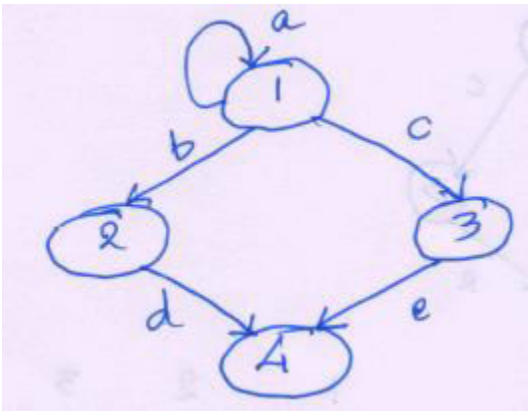
	1	2	3	4
1		a+b	c	
2				
3				d
4				

19. Connection Matrix: The matrix representation used above is only a tabular representation of the graph with no other extra information.

    If link weights are added to each cell entry, the graph matrix can be used as a powerful tool in testing. The links between two nodes are assigned link weight that becomes an entry in the matrix cells.

    *Link weight provides direct information about control flow. If a link exists, link weight is 1; else 0 or empty cell.

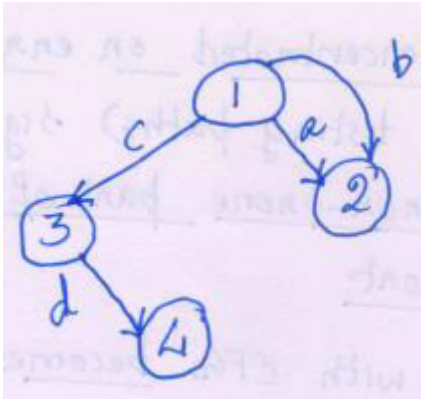
    Ex:



Connection Matrix:

	1	2	3	4
1	1	1	1	
2				1
3				1
4				

Note that even if more than one link is present between two nodes we show it only as 1.



Connection Matrix:

	1	2	3	4
1		1	1	
2				
3				1
4				

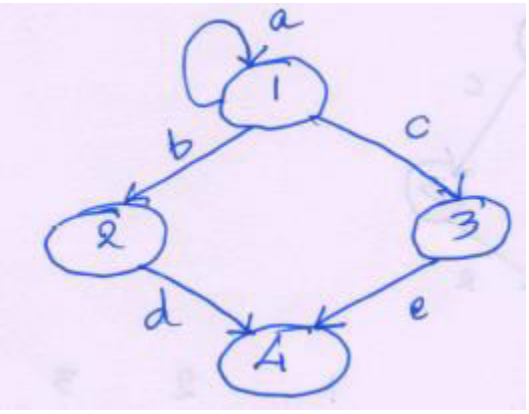
2-1=1

1-1=0

Total = 1+1=2 (Cyclomatic number)

20. Method:
    (1) For each row, count the no. of 1s and write it in front of the row.
    (2) Subtract 1 from that number. Ignore the blank rows.
    (3) Add the final count of all rows.
    (4) Add 1 to the final count to obtain the final sum.
    (5) The number obtained is known as Cyclomatic number of the graph.

21. Ex:



	1	2	3	4
1	1	1	1	
2				1
3				1
4				

3-1=2
1-1=0
1-1=0

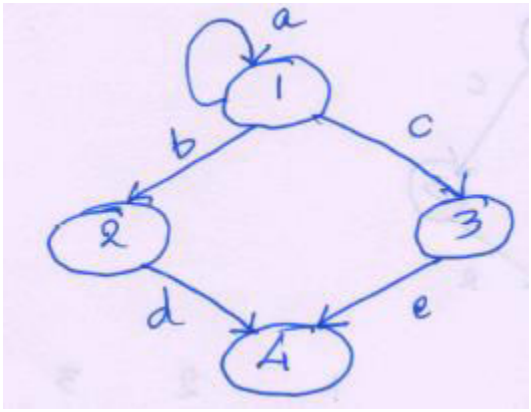
Total = 2+1 = 3

22. Use of graph matrix for finding set of all paths: Producing a set of paths is important to trace out the k-link paths.
    Ex: How many 2-link paths exist from node 1 to node 2?

    Aim: Using matrices to obtain the set of all paths between all nodes.

    The power operation on a matrix expresses the relation between each pair of nodes via intermediate nodes under the assumption that the relation is transitive. (a->b, b->c => a->c).

23. Ex:



	1	2	3	4
1	a	b	c	
2				d
3				e
4				

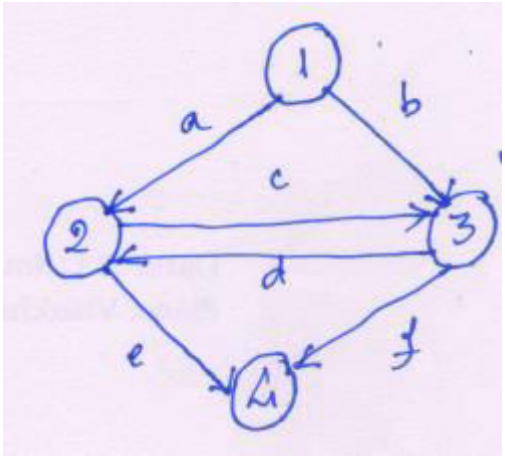
Find 2-link paths for each node.

For finding 2-link paths, we should square the matrix.

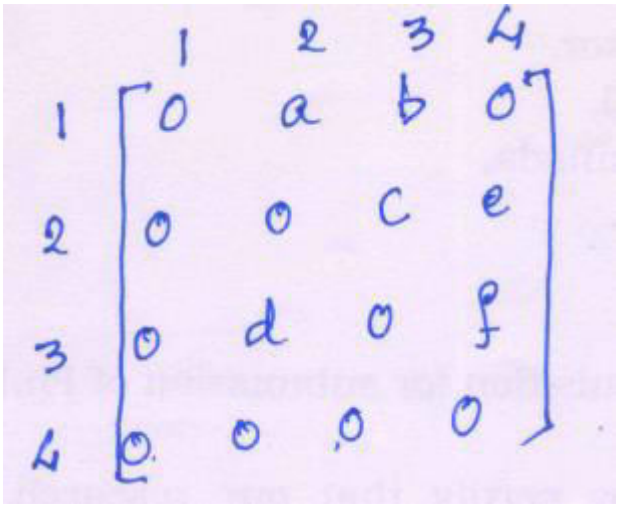
a b c 0		a b c 0		a^2 ab ac bd+ce
0 0 0 d		0 0 0 d		0 0 0 0
0 0 0 e	X	0 0 0 e	=	0 0 0 0
0 0 0 0		0 0 0 0		0 0 0 0

The resulting matrix shows all the 2-link paths from one node to another.
Ex: From node 1 to node 2, there exists a link ‘ab’

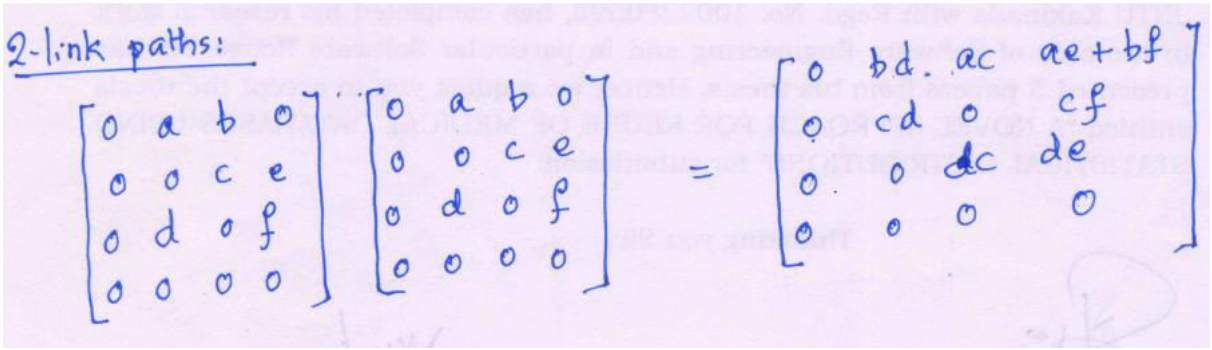
24. Ex2:



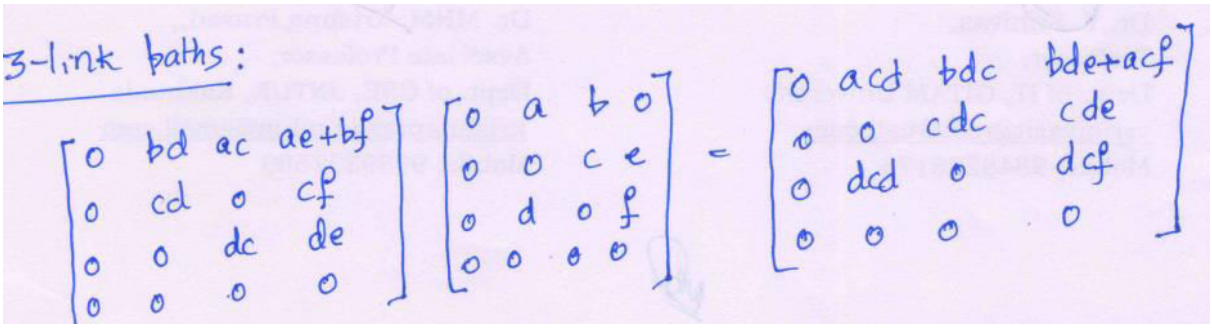
Graph Matrix:



2-link Paths:



3-link Paths:

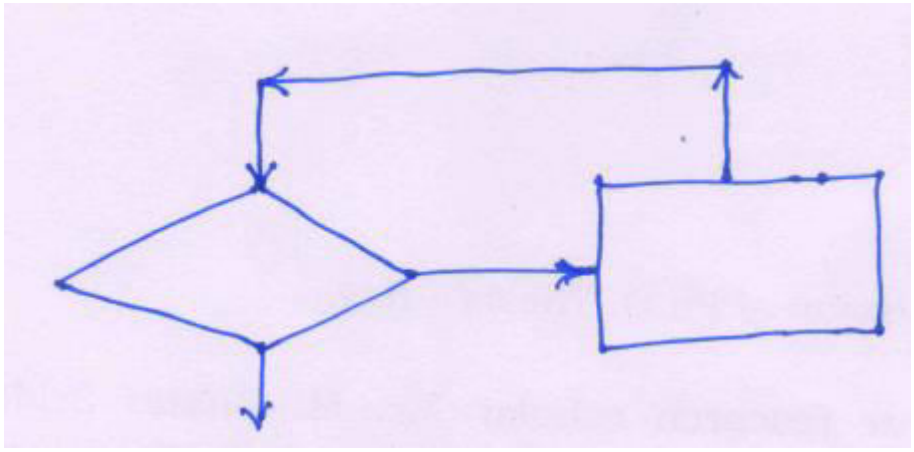


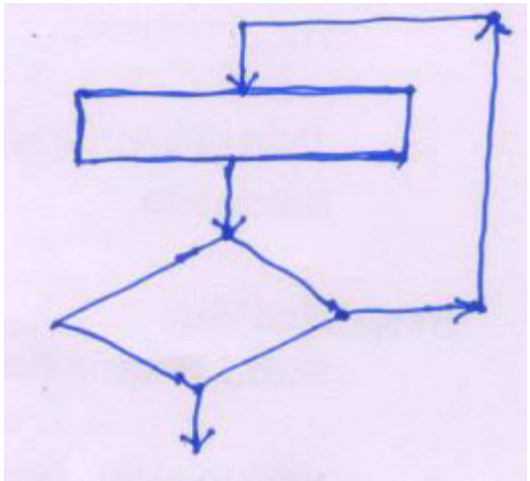
25. □ For n number of nodes, we can get the set of all paths of (n-1) links length with the use of matrix operations.

    These operations can be programmed and used as a software testing tool.

26. Loop Testing: It is an extension to branch coverage. If loops are not tested properly, bugs may be undetected. Loop testing can be performed effectively while performing development testing on a module.

    (a) Simple loops: Single loop exists in the control flow.



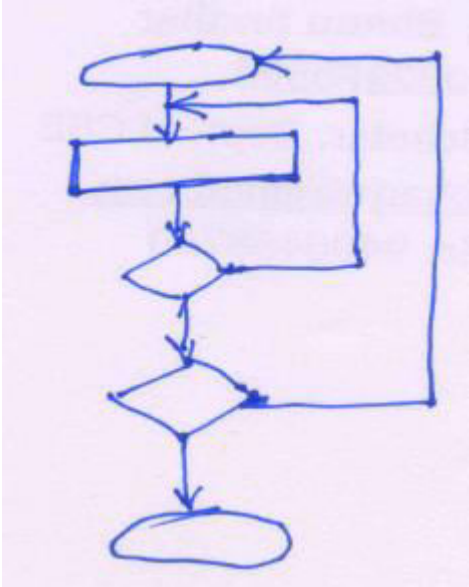


    Test cases to be considered here:

    - Check whether you can bypass the loop or not.

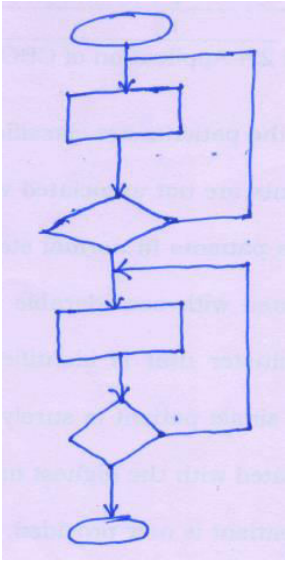
- Check whether the loop control variable is negative or not.
- Write one TC that executes the statements inside the loop.
- Write TCs for a typical no. of iterations through the loop.
- Write TCs for checking the boundary values of the maximum and minimum no. of iterations. (min-1, min, min+1 .....)

(b) Nested Loops: When two or more loops are embedded, it becomes a nested loops.



In this case, we should start with the innermost loop while holding the outermost loops to their minimum values.

(c) Concatenated Loops: Two loops are concatenated if it is possible one after exiting the other, while still on a path from entry to exit.



(d) Unstructured Loops: These loops cannot be tested directly and must be converted into nested/concatenated loops or the program must be resigned.

27. Data Flow Testing: It is a WBT technique that can be used to detect improper use of data values due to coding errors.
    Ex: Not initialized before usage or not utilized at all after declaration/initialization.

DFT looks out for inappropriate data definition, its usage in predicates, computations and terminations. Mostly used for detect out of scope data usage (double for int).

For this purpose, A CFG is used.

28. States of a Data Object:
    (a) Defined (d): A data object (DO) is called defined (d) when it is initialized i.e., on the LHS of an assignment statement.
    Other examples: File has been opened, an object has been allocated, data pushed into a stack, a record written etc.
    (b) Killed (k): When an object has been reinitialized, scope of a loop variable is finished, file closed etc.
29. Usage (u): DO is on the RHS of an assignment statement, or used as a loop control variable, as a pointer etc. Notes that there are two types of usage: 'c' for computation and 'p' for predicate.
30. DF Anomalies: These are usage patterns which may lead to incorrect code execution.

Anomaly	**Acronym**	**Effect of Anomaly**
'du'	Define-use	Normal case. Allowed.
*'dk'	Define-kill	Potential bug. Data killed without use.
'ud'	Use-define	Data is used and redefined. Normal.
'uk'	Use-kill	Allowed.
*'ku'	Kill-use	Serious bug.
'kd'	Kill-define	Allowed.
*'dd'	Define-define	Redefining without any usage. Potential bug.
'uu'	Use-use	Normal.
*'kk'	Kill-kill	Harmless bug but not allowed.

NOTE: Potential Bug => having a capacity to become a bug in the future.

31. Single Character Anomalies:
    -x: All prior actions are of no interest to x.
    x-: All post actions are of no interest to x.

Anomaly	**Explanation**
-d	Normal.
*-u	Used without definition. Potential Bug.
*-k	Data (might be) killed before definition. Bug.
k-	OK
*d-	[define last]. Potential bug.
u-	OK

32. Terminology in DFT: Let
    P --> program
    G(p) --> graph
    V--> variables
    Assumption is that G(p) has single entry and exit nodes.
    (1) Definition Node: Defining a variable means assigning a value to a variable for the first time in the program.

Ex: Input statements, assignment statements, loop control statements, procedure calls etc.

(2) Usage Node: It means the variable has been used in some statement of the program. Node 'n' that belongs to G(p) is a usage node of variable 'v' if the value of v is used at the statement corresponding to node n.
Ex: Output statements, assignment statements (RHS) etc. [c, p]

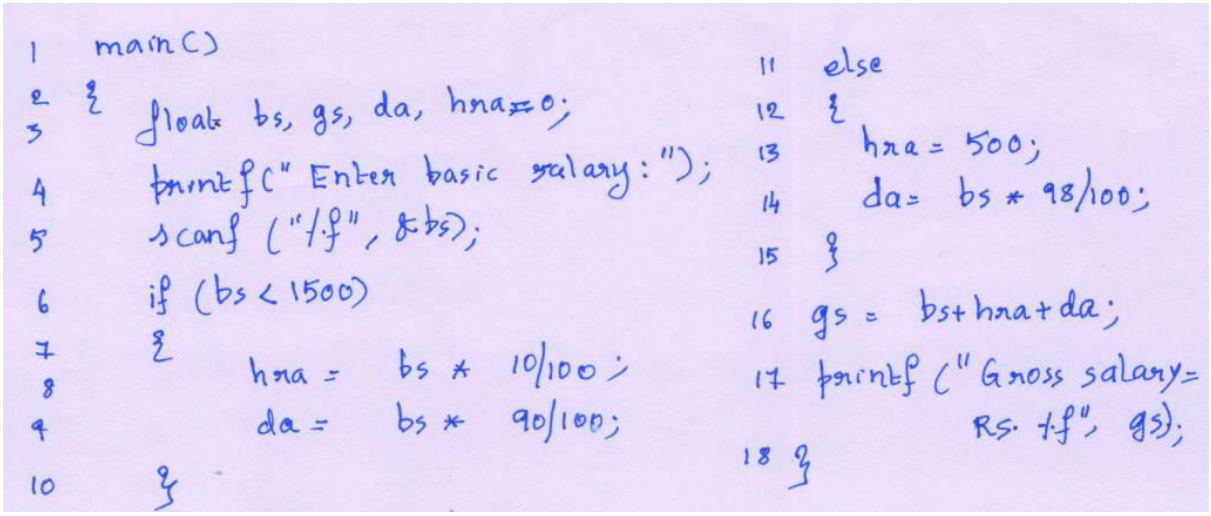
(3) Loop-free Path Segment: It is a path segment for which every node is visited once at most.

(4) Simple Path Segment: One node is visited twice. It may be loop free or if a loop does exist, only one node is involved.

(5) 'du' path: Path between definition node and usage node of a variable.

(6) Definition-clear Path (dc path): Path between definition node and usage node of a variable such that no other node in the path can redefine the variable v.

33. Static Data Flow Testing: Source code is analysed without execution of the program.
Ex: If basic < 1500, HRA = 10% of basic & DA = 90% of basic. If basic >= 1500, HRA = 500 & DA = 98% of basic. Calculate the gross salary.



Find out d-u-k patterns for all variables in the source code.
For 'bs':

Pattern	Line No.	Explanation
-d	3	Normal
du	3-5	Normal
uu	4-6, 6-7, 7-12, 12-14	Normal
uk	14-16	Normal
k-	16	Normal

For 'da':

Pattern	Line No.	Explanation
-d	7	Normal
du	7-4	Normal
uk	14-16	Normal
k-	16	Normal

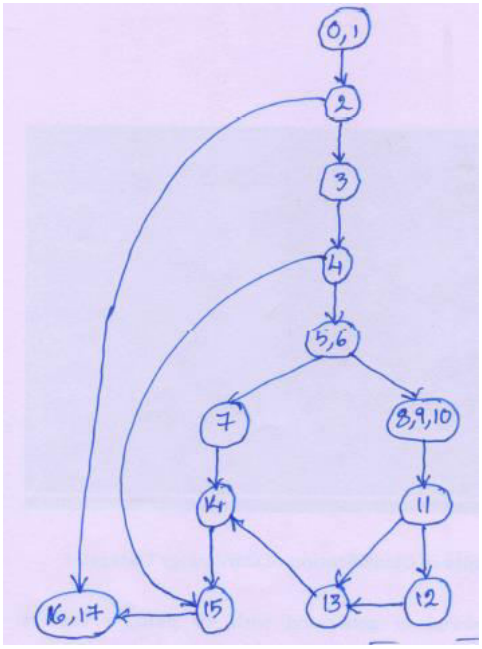
For ‘hra’:

Pattern	Line No.	Explanation
-d	1	Normal
dd	1-6 or 1-11	Harmless bug.
du	6-14 or 11-14	Normal
uk	14-16	Normal
k-	16	

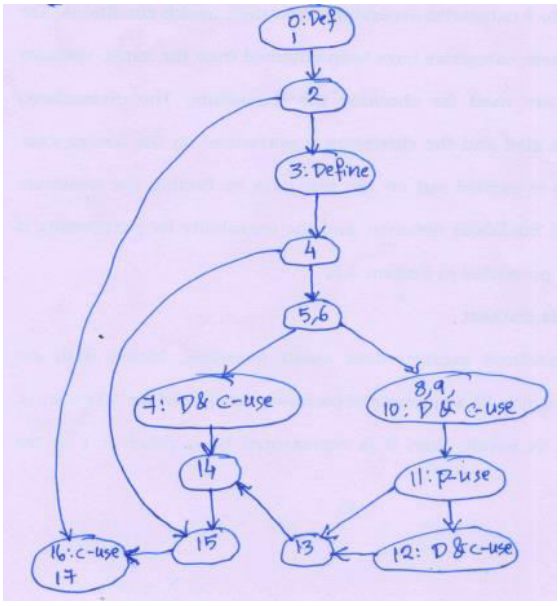
34. Disadvantages of Static Analysis:
    (a) It is not possible to determine the state of a data variable by only static analysis. This applies when the variable is an array or pointer and so on.
    (b) If the index is generated dynamically, static analysis is of no use.
    (c) The static methodology might detect an anomaly that is used rarely but not all of them.
35. Dynamic DFT: This is done to detect bugs in data usage during code execution. The TCs here try to detect every d and u of a data variable. The strategies are given below:
    (a) All du paths (ADUP): Every du-path of every variable should be used under some test. This can be quoted as the strongest strategy that uses maximum number of paths for testing. It is also the superset of all other strategies.
    (b) All uses (AU): For every usage of a variable, there is a path from the definition to the use.
    (c) APU+C (All predicates uses + some computational uses): For every variable at least one dc path should exist to all of its predicate uses. If predicate uses are less, we can add some c uses.
    (d) ACU+P (All computational uses + some predicate uses): One dc path to every computational uses and some predicate uses also.
    (e) APU
    (f) ACU
    (g) AD: All definitions are covered.
36. Example: A C program for calculating work and concerned payment is given below:

```
    int work;
0.  double payment = 0;
1.  scanf ("%d", work);
2.  if ( work >0) {
3.      payment =40;
4.  if (work>20)
5.  { if (work ≤ 30)
6.
7.      payment = payment + (work -25 ) * 0.5;
8.  else
9.  {
10. →   payment = payment + 50 + (work-30) * 0.1;
11.     if (payment ≥ 3000)
12.         payment = payment * 0.9;
13.  }
14. }
15. }
16. printf ("Final payment", payment);
17. }
```

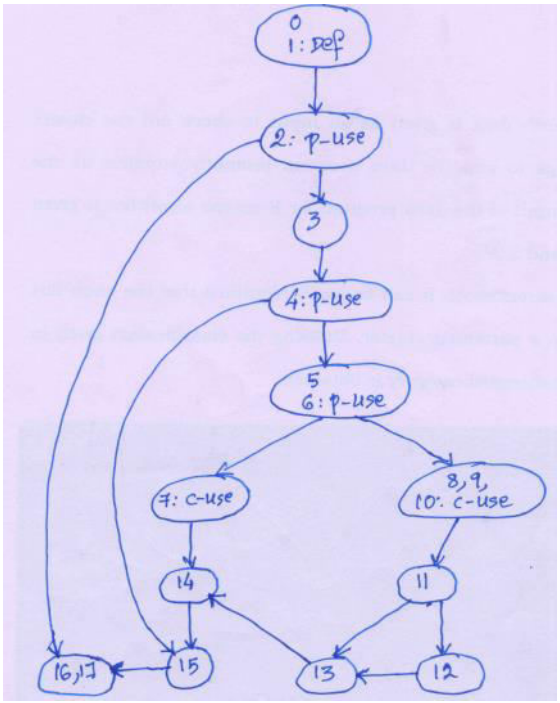
Decision to Decision Path Graph (DD Path Graph) for this program is given below:



DFG for ‘payment’ variable:



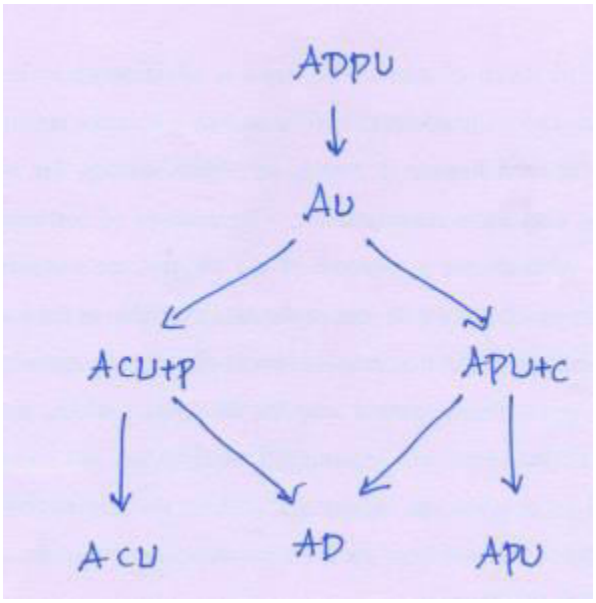
DFG for ‘work’ variable:



List:

Variable	Defined at	Used at
Payment	0,3,7,10,12	7,10,11,12,16
Work	1	2,4,6,7,10

37. Order of DFT Strategies: Note that the strongest strategy is at the root and weakest at the last level.



38. Mutation Testing: It is the process of mutating (changing) some segment of code (placing some error) and testing this mutated code with some test data.

    If the test data can detect some errors, it is acceptable; else the quality of test data must be improved so that they can detect at least some mutants.

    Mutation helps the user to create high quality test data.

    Faults are introduced/induced into the program in different ways – each program becomes a new version. The goal here is to make the program collapse in some way.

    A mutant is *killed* if a test case causes it to fail. This is done since the fault has been noticed and rectified – now there is no use of the mutant.

    The main objective here is to select efficient test data with high error-detection power.

39. Primary Mutants:

    Ex: .....

```
if (a>b)
  x+=y;
else
  x=y;
printf("%d",x);
.........
```

    Mutants that can be considered here are:

    M1: x=x-y;

    M2: x=x/y;

    M3: x=x+1;

M4: printf("%d",y);

It is to be understood from the above example that primary mutants are single modifications of the initial program (using some operators). Mutation operators are dependent on programming languages that are to be worked upon.

Results:

Test Data	X	Y	Initial Program Result	Mutant Result
TD1	2	2	4	0 (M1)
TD2 (x & y ≠ 0)	4	3	7	1.4 (M2)
TD3 (y≠1)	3	3	6	4 (M3)
TD4 (y≠0))	4	2	6	2 (M4)

40. Secondary Mutants:

Ex: if (a<b)
c=a;

Mutants can be:

M1: if (a<=b-1) c=a;
M2: if (a+1<=b) c=a;
M3: if (a==b) c=a+1;

These are known as secondary mutants since multiple levels of mutation are applied on the initial program.

Ex1: Original Program (P)

```
r=1;
for(i=2;i<=3;++i)
{
        if(a[i] > a[r])
                r=i;
}
```

The mutants for P are:

M1:
```
r=1;
 for(i=1;i<=3;++i)
{
    if(a[i] > a[r])
            r=i;
 }
```

M2:
```
r=1; for(i=2;i<=3;++i) {
if (i>a[r])
  r=i;  }
```

M3:
```
r=1; for(i=2;i<=3;++i) {
if(a[i] >= a[r])
r=i;  }
```

M4:
```
r=1; for(i=2;i<=3;++i) {
if(a[r] > a[r])
r=i;  }
```

Test Data Selection:

	A[1]	A[2]	A[3]
TD1	1	2	3
TD2	1	2	1
TD3	3	1	2

Apply the above data to mutants M1, M2, M3 and M4.

	P	M1	M2	M3	M4	Killed Mutants
TD1	3	3	3	3	1	M4
TD2	2	2	3	2	1	M2,M4
TD3	1	1	1	1	1	NONE

M1 and M3 are not killed.
Hence test data is incomplete and new test data is needed.

41. Mutation Testing Process:
    - Construct the mutants (induce faults).
    - Add TCs to the mutation system and check the outputs.
    - If the output is incorrect, that means a fault has been found, program must be modified and the process must be restarted.
    - If the output is correct, execute that TC with every mutant.
    - If the output of a mutant differs from the original program on the same test case (TC), kill the mutant.
    - After each TC is executed, the remaining mutant(s) fall into two categories:
      (a) One: Mutant is functionally equivalent to the original program.
      (b) Two: The mutant can be killed, but a new set of TCs must also be created and tested for the mutant before killing it.
    - Mutant score for a set of test data = The % of non-equivalent mutants killed by that data.

NOTE: If mutation score = 100%, then the test data is called mutation adequate. (satisfactory)

42. Drawbacks of Dynamic Testing:
    (a) Dynamic testing uncovers bugs at later stages of SDLC; so it becomes costly to debug.
    (b) Dynamic testing is expensive and time consuming (create, run, validate and maintain TCs).
    (c) Efficiency decreases as the size of software increases.
    (d) Dynamic testing provides information about bugs but debugging is not easy since the exact cause and place of failure are difficult to locate.
    (e) Dynamic testing can't detect all potential bugs. Ex: dd and kk are potential bugs (i.e., warnings in the present but may become bugs in the future).

43. Static testing techniques are now to be applied (in deep) in WBT to obtain higher quality of software. Static testing is also called human/non-computer testing.

    Static testing can be applied for most of the verification activities, at every stage of SDLC. It checks the software and confirms that all standards are met. Ex: Static testing can be applied for requirement design, test plans, source code, user manuals, maintenance etc.

    *Nearly 60% of the errors can be reported by static testing:

    Advantages:
    (a) Quality of product increases since bugs can be found out and fixed at early stages.
    (b) Overall cost when compared to dynamic testing is low.
    (c) By the usage of static testing, dynamic testing also improves.
    (d) Productivity increases.

44. Types of static testing:
    (a) Software Inspections
    (b) Walkthroughs
    (c) Technical reviews
45. Inspections:
    - Introduced by IBM in 1970s.
    - Detects and removes errors after each phase of SDLC and improves the quality of the software.
    - It is a manual examination of the software.
    - It can be applied to any product at any phase of SDLC.
    - Doesn't need executable code or test cases. Bugs here are found on less executed paths.
    - Inspection is machine independent and can be used even before the software is ready (on algorithms).
    - The documents that can be inspected are: SRS, SDD (s/w design and development), code and test plan.
    - Inspection involves inspection steps, role of participants, item being inspected.

- Entry and exit criteria are used to determine whether an item is ready to be inspected.

46. Inspection Team Members:
    - Author: Programmer who has written the code and who has to fix the bugs.
    - Inspector: Finds the errors. (Tester)
    - Moderator: Manages this whole (inspection) process.
    - Recorder: Files all the details of an inspection meeting.
47. Inspection Process: Also called Fagan's original Process.

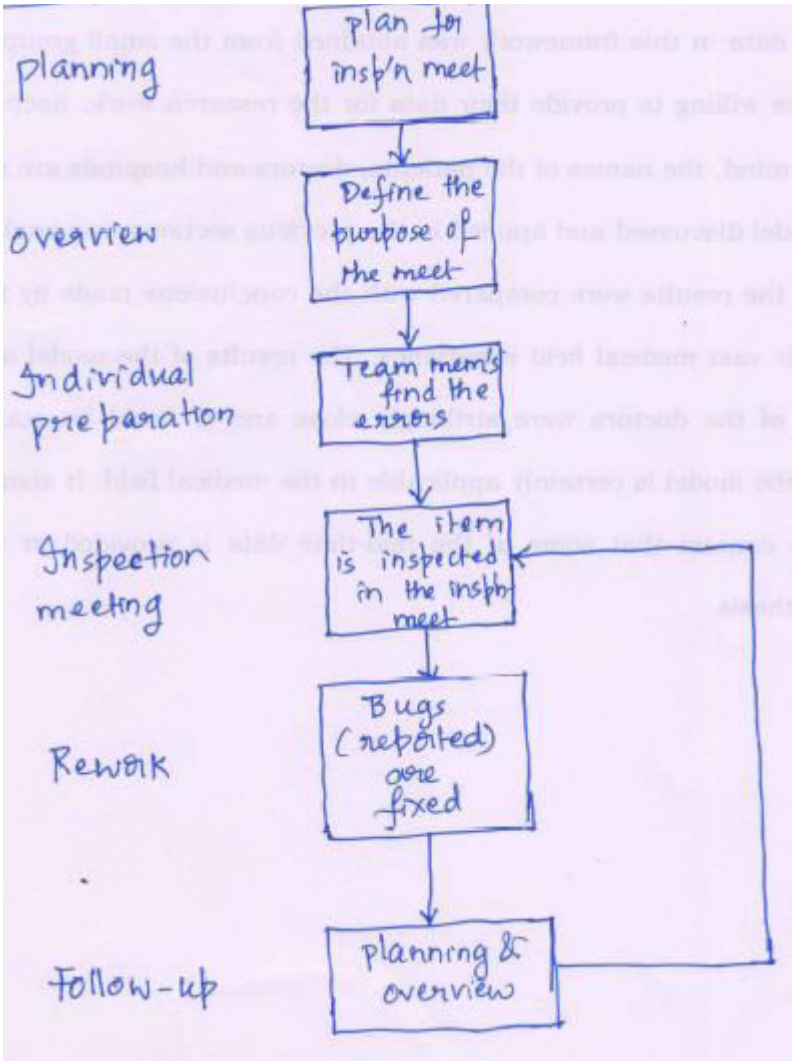


Fig 3.7: Inspection Process

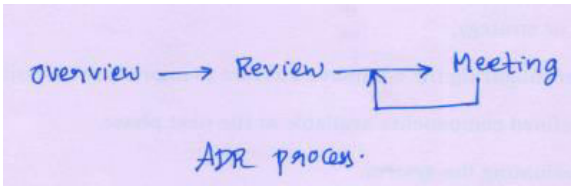
48. Planning (Inspection): The product to be inspected is identified, a moderator is assigned, and the goal is stated. The moderator checks if the product is ready for inspection, selects the team, gives the tasks, schedules time, and distributes the inspection material.
49. Overview: Here the inspection team is provided with the needed information for inspection.
50. Individual Preparation: A team member now utilizes the information provided in overview and prepares for the inspection process.
    Potential Errors are traced out, filed and reported to the moderator.
    Checklists are used during this stage for guidance on bug identification. Logs are prepared.
    Moderator checks the efficiency of the log and submits a final report to the author.

51. Inspection Meeting: It is the stage of actual inspection where the author checks each issue raised. All the members arrive at a final conclusion deciding the issues that have to be fixed, how and by whom. Summary of the meeting is produced by the moderator.
52. Rework: The author fixed the bugs and reports back to the moderator.
53. Follow-up: Moderator checks if all the reported bugs have been fixed. If not, another inspection meet is called.
54. Advantages of Inspection Process:
    - Bug Reduction: the no. of bugs per 1000 lines can be reduced by two-thirds.
    - Bug Prevention: Experiences of previous inspections can be used for future bug prevention.
    - Productivity: Cost of inspection is less since no execution is needed, thereby increasing the productivity. (General: 23%)
    - Real-time feedback to developers
    - Reduction in resource usage: Since bugs are identified and fixed close to their origin we get less cost and more time.
    - Quality Improvement: Better logic, more testing, etc.
    - Checking coupling and cohesion: Note that coupling is the level of independence between two software modules and cohesion is the degree to which elements of module belong to each other. Hence high cohesion => loose coupling.
    - Learning through inspection: This improves the programming capability of the team members.
    - Process Improvement: Learning from the results
      Ex: Finding most error-prone modules and identifying the error types.
55. Effectiveness of Inspection Process: Results of an analysis show that 52% of the errors can be found by inspection. In the remaining 48%, only 17% are detected by WBT.

    Rate of Inspection => How much evaluation of an item has been done by the system
    Rate is very fast => Coverage is high, but less no. of errors are detected
    Rate is too slow => Coverage is low, more no. of errors are found. But cost increases.

    □ Error Detection Efficiency = Error found by an inspection
    ---------- * 100
    Total no. of errors before inspection

56. Cost of Inspection Process: Let 4 members be part of an inspection team. 100 lines of code are considered. Then, cost of inspection = one person per a day of effort. Note that testing costs depend upon the no. of faults in the software.
57. Variants (diff. types) of Inspection Process:
    (a) Active Design Reviews (ADR): These are used for inspecting the design phase of SDLC. It covers all the sections of design based on a set of small reviews instead of a single large set. (single review)

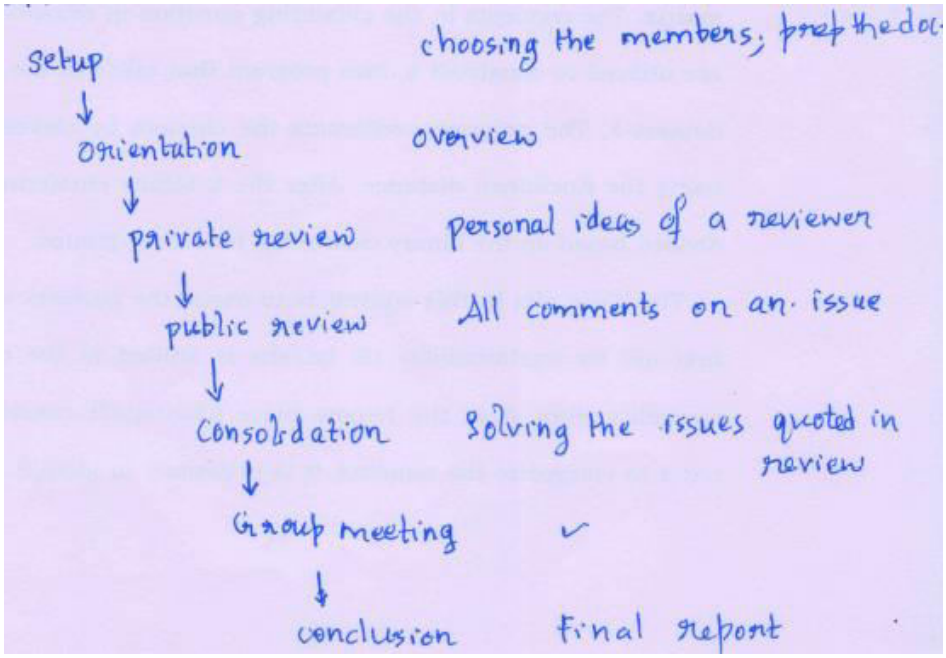


Overview: Explains the structure of the concerned module

Review: Reviewers are given questionnaire and time frame to answer, raise any doubts etc.

Meeting: Designers read the responses and resolve the queries, problems etc. This meeting(s) goes on till all the issues are resolved.

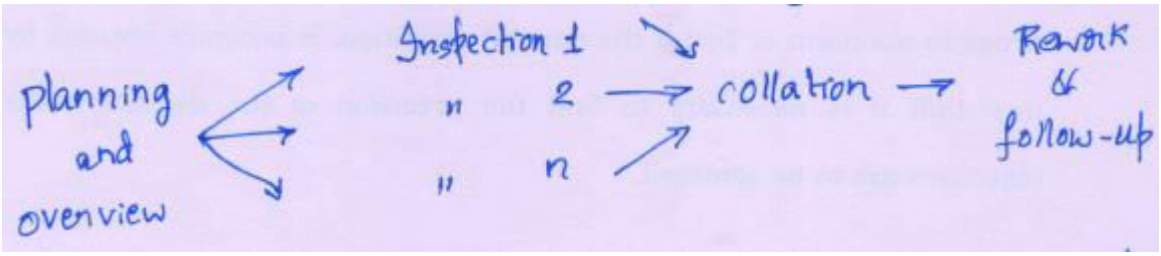
(b) Formal technical Asynchronous Review Method (FTArm):



(c) Gilb Inspection: (Gilb & Graham)

(d) Humphrey's Inspection:

(e) N-fold Inspection: The effectiveness of the inspection process can be increased by replicating it.



(f) Reading Techniques: These are steps guiding an inspector to understand the product.

1. Adhoc Method
2. Checklists
3. Scenario based testing
   - Perspective based
   - Usage based
   - Abstract driven
   - Task driven
   - Function point based scenarios

58. Structured Walkthroughs: Less former and less rigorous technique. No rules and no methods exist. Members are: coordinator, presenter/developer, recorder, tester, maintenance oracle, standards bearer, user agent.
59. Tech Reviews: This is intended to evaluate the software in the light of development standards, guidelines, and specifications. Same as walkthrough but includes management.

****