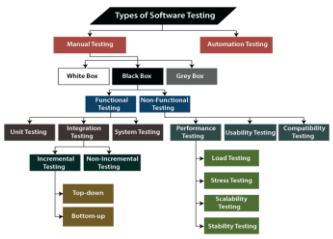
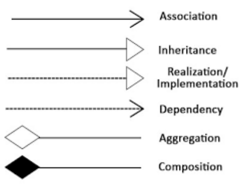
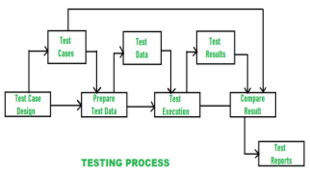
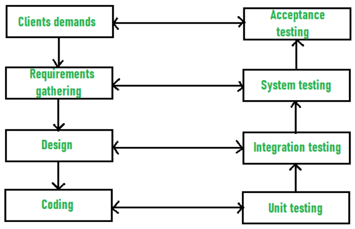






Software Design and Testing

Dynamic Testing: White box Testing



LEVELS OF TESTING

Chapter 5

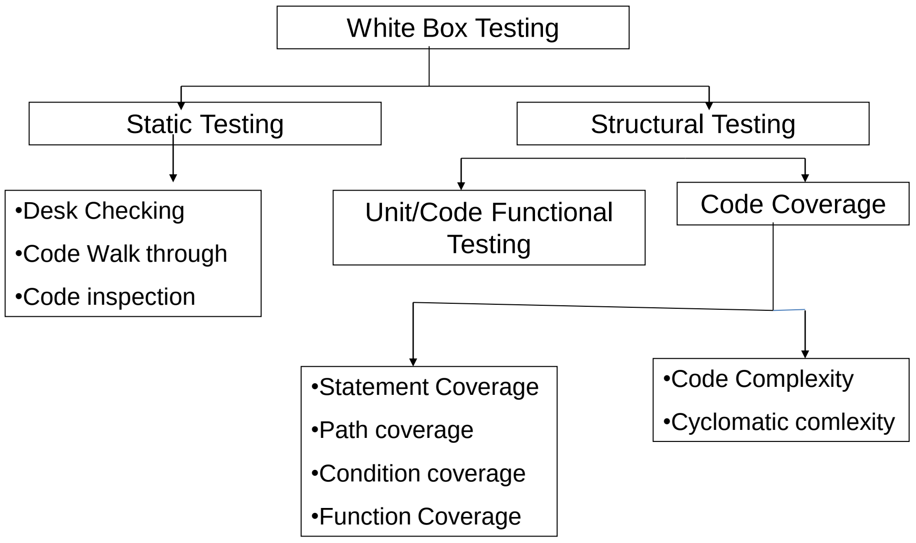
Outline

- Need of white-box testing.
- Basis Path Testing.
- Graph and Loop Testing.
- Data Flow testing.

Introduction

- White-box or structural or development testing is another effective testing technique in dynamic testing.
- The structure means the logic of the program which has been implemented in the language code.
- Every software product is realized by means of a program code. White box testing is a way of testing the external functionality of the code by examining and testing the program code that realizes the external functionality.
- White box testing takes into account the program code, code structure, and internal design flow.
- It is also known as **glass-box** testing, as everything that is required to implement the software is visible.
- This testing is also called **structural** or **development** testing.
- A number of defects come about because of incorrect translation of requirements and design into program code.

Types of White Box Testing



Need Of White-box Testing

- White-box testing techniques are used for testing the module for initial stage testing. Black-box testing is the second stage for testing the software.
  - Though test cases for black box can be designed earlier than white-box test cases, they cannot be executed until the code is produced and checked using white-box testing techniques.
  - Thus, white-box testing is not an alternative but an essential stage.
- There are categories of bugs which can be revealed by white-box testing, but not through black-box testing. There may be portions in the code which are not checked when executing functional test cases, but these will be executed and tested by white-box testing.

Need Of White-box Testing

- Errors which have come from the design phase will also be reflected in the code, therefore we must execute white-box test cases for verification of code (unit verification).
- We often believe that a logical path is not likely to be executed when, in fact, it may be executed on a regular basis. White-box testing explores these paths too.
- Some typographical errors are not observed and go undetected and are not covered by black-box testing techniques. White-box testing techniques help detect these errors.

Logic Coverage Criteria

- Structural testing considers the program code, and test cases are designed based on the logic of the program such that every element of the logic is covered.
- Structural testing takes account of code, code structure, internal designs, and how they are coded. In structural testing tests are actually run by the computer on the built product.
- The knowledge of internal structure of the code can be used to find the number of test cases required to guarantee the given level of test coverage.
- Structural testing entails the actual product against some pre-designed test cases to exercise as much as of the code as possible or necessary. A given portion of the code is exercised if a test case causes the program to execute that portion of the code when running the test.

Logic Coverage Criteria

- Since a product is realized in terms of program code, if we can run test cases to exercise the different parts of the code, then that part of the product is realized by the code gets tested. Code coverage testing involves designing and executing test cases and finding out the percentage of code that is covered by the testing.
- This leads to the notation of code coverage
- Divide the program into elements(e;g; statements)
- Define the coverage of a test suite to be

$$\frac{\#\ of\ \ elements\ \ executed\ by\ \ suite}{\#\ of\ \ element\ in\ the\ \ program} * 100$$

Statement Coverage

- In most of the programming languages, the program construct may be a sequential control flow, a two-way decision statement like if – then – else, a multi-way decision statement like switch or even loops like while, do, repeat until and for.
- Statement coverage refers to writing test cases that execute each of the program statements. We assume that "more the code covered, the better is the testing of the functionality".
- For a set of sequential statements (i.e., with no conditional branches), test cases can be designed to run through from top to bottom.
- The statement coverage for a program, which is an indication of the percentage of statements actually executed in a set of tests, can be calculated by the following formula
- **Statement Coverage = (Total Statements Exercised) / (Total Number of Executable Statements in Program) x 100**

Statement Coverage

- Draw the flow in the following way:
  - Nodes represent entries, exits, decisions and each statement of code.
  - Edges represent non-branching and branching links between nodes.

Statement Coverage Example

- A commercial bank has introduced an incentive policy of giving bonus to all its deposit holders. The policy is as follows: A bonus of 2% of the balance held on 31st December is given to every one, irrespective of their balance, and 5% is given to female account holder if their balance is Rs. 5000. Write a C program for it.
- Also make the test cases and test data to test your program and execute your program with your test data.
- Find the smallest test case which covers maximum statements

Statement Coverage Example

```
#include <stdio.h>
void main()
{
          char sex;                              ⎫
          float balance, bonus;                  ⎪
          printf(“Enter sex F/M : ”);            ⎬ A
          scanf(“%c”,&sex);                      ⎪
          printf(“Enter balance : ”);            ⎪
          scanf(“%f”,&balance);                  ⎭
          if(sex == ‘F’)         ──────────────→  B
          {
                    if(balance > 5000)  ──────────────→  C
                              bonus = 0.05 * balance;  ──────────────→  D
                    else
                              bonus = 0.02 * balance;  ──────────────→  E
           }
          else
                    bonus = 0.02 * balance;  ──────────────→  F
          balance = balance + bonus              ⎫ G
          printf(“Your balance is %f”, balance); ⎭
}
```

Statement Coverage Example

Code	Source line	F,6000	M,6000	F,3000	M,3000
A	scanf sex scanf balance	*	*	*	*
B	if sex == ‘F’	*	*	*	*
C	If balance > 5000	*			
D	bonus = 0.05 * balance	*			
E	bonus = 0.02 * balance			*	
F	bonus = 0.02 * balance		*		*
G	printf()	*	*	*	*
	Total statement covered	5/7*100 = 71%	4/7*100 = 57%	4/7*100 =57%	4/7*100 =57%

Branch Coverage

- Here test cases are designed such that the different branch condition are given true and false value in turn.
- Here test requirements: Branches in a program.

$$\frac{\#\ of\ \ executed\ branches}{\#\ of\ branches\ in\ \ program} *100$$

- It is stronger testing criteria than the statement coverage based testing.
- Example : Take the above problem of balance
- Construct the control flow graph of it.

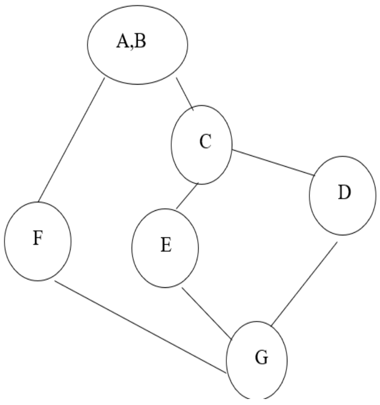
How to draw Control flow graph?

- Number all the statements of a program.
- Numbered statements: represent nodes of the control flow graph.
- An edge from one node to another node exists: if execution of the statement representing the first node can result in transfer of control to the other node.
- Elements of CFG
- Nodes: (3 types)
  - Statement nodes : single- entry, single exit sequence of statements
  - Predicate (decision) nodes: Conditions for branching.
  - Auxiliary nodes (optional) : for easier understanding (e.g. "merge point" for if)
- Edges : Possible flow of control

15

Branch Coverage Example

- Take the above problem of balance
- Construct the control flow graph of it.
- Solution :
- Control flow graph



Code	Source line
A	scanf sex scanf balance
B	if sex == 'F'
C	If balance > 5000
D	bonus = 0.05 * balance
E	bonus = 0.02 * balance
F	bonus = 0.02 * balance
G	printf()

16

Branch Coverage Example

Arcs	F,6000	M,6000	F,3000	M,3000
AB-C	*		*	
C-D	*			
C-E			*	
AB-F		*		*
F-G		*		*
E-G			*	
D-G	*			
	3/3*100 = 100%	2/3*100 = 67%	3/3*100 = 100%	2/3*100 = 67%

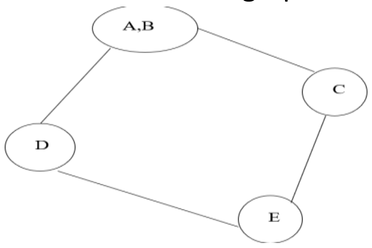
Condition Coverage

- For example, in if-then-else, there are 2^2 or 4 possible True / False conditions.
- The condition coverage which indicates the percentage of conditions covered by a set of test cases, is defined by the following formula

  Condition Coverage = (Total Decisions Exercised) / (Total Number of Decisions in Program) x 100
- Thus it becomes quite clear that this technique of condition coverage is much stronger criteria than path coverage, which in turn is a much stronger criteria than statement coverage.

Condition Coverage Example

- Take the above problem of balance
- Control flow graph



Code	Source line
A	scanf sex scanf balance
B	if sex == 'F' && balance > 5000
C	bonus = 0.05 * balance
D	bonus = 0.02 * balance
E	printf()

Combination	Possible Test case	Branch	% of condition coverage
T T	F, 6000	AB-C	
F F	M,2000	AB-D	
F T	M, 6000	AB-D	
T F	F, 2000	AB-D	

Path testing

- In path coverage technique, we split a program into a number of distinct paths. A program or a part of a program can start from the beginning and take any of the paths to its completion. The path coverage of a program may be calculated based on the following formula

  Path Coverage = (Total Path Exercised) / (Total Number of Paths in Program) X 100
- In path coverage based testing strategy requires us to design test cases such that all linearly independent paths in the program are executed at least once.
- A linearly independent path is defined in terms of the control flow graph (CFG) of a program.
- A control flow graph (CFG) describes: the sequence in which different instructions of a program get executed.
  - the way control flows through the program.

Path testing

- Consider the following flow graph having different paths.

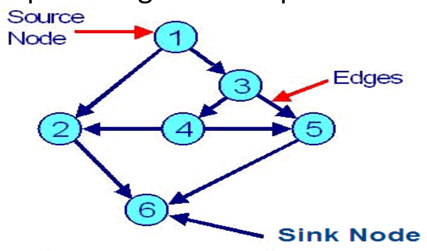
Some of the paths are

Path-1: 1 > 2 > 6

Path-2: 1 > 3 > 5 > 6

Path-3: 1 > 3 > 4 > 5 > 6

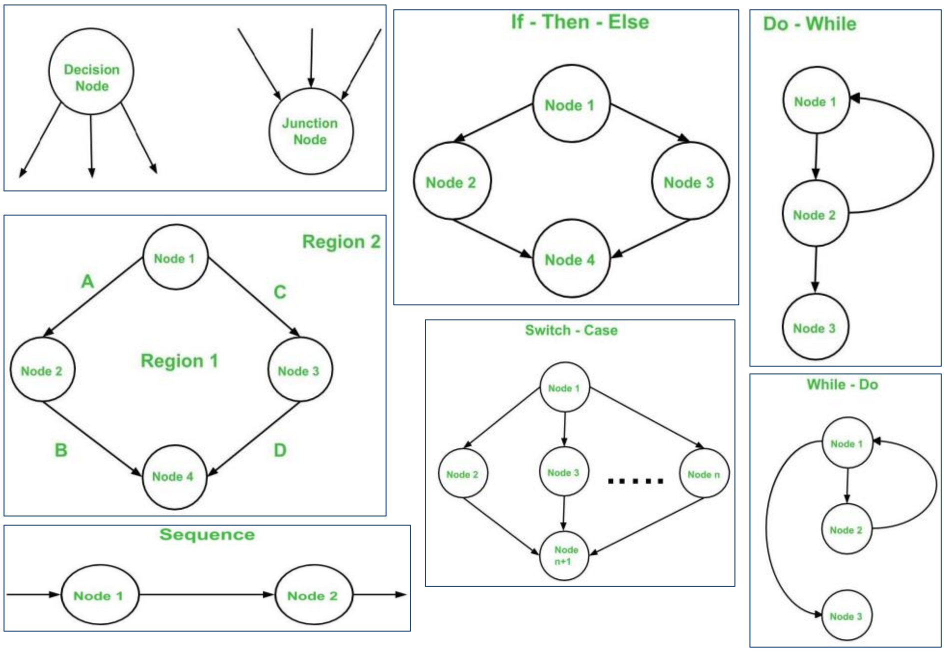
Path-4: 1 > 3 > 4 > 2 > 6



- Regardless of the number of statements in each of these paths, if we can execute these paths, then we would have covered most of the typical scenarios.
- Path coverage provides a stronger condition of coverage compared to statement coverage as it relates to the various logical paths in the program rather than just program statements.

21

Standard notations used in constructing a flow graph are as under



Cyclomatic Complexity

- McCabe's Cyclomatic metric, V(G) of a graph "G" with "n" vertices and "e" edges is given by the following formula
- V(G) = e – n + 2
- Steps to compute the complexity measure, V(G) are as under

1. Construct the flow graph from the source code or flow charts.
2. Identify independent paths.
3. Calculate Cyclomatic Complexity, V(G).
4. Design the test cases.

Meaning of V(G)

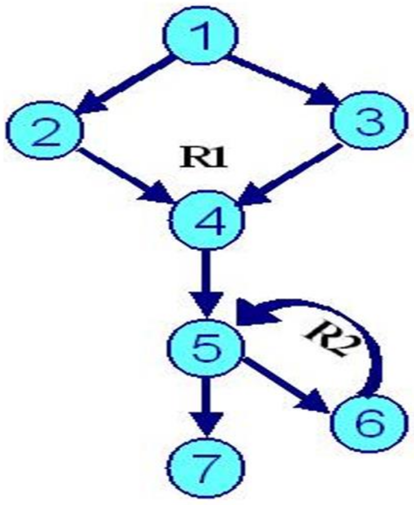
Complexity No.	Corresponding Meaning of V(G)
1-10	1) Well-written code, 2) Testability is high, 3) Cost / effort to maintain is low
10-20	1) Moderately complex code, 2) Testability is medium, 3) Cost / effort to maintain is medium.
20-40	1) Very complex code, 2) Testability is low, 3) Cost / effort to maintain is high.
> 40	1) Not testable, 2) Any amount of money / effort to maintain may not be enough.

Cyclomatic Complexity Example

```
void foo (float y, float a *, int n)

{
                                /*1*/
    float x = sin (y) ;         /*1*/
    if (x > 0.01)               /*1*/
        z = tan (x) ;           /*2*/
    else

        z = cos (x) ;           /*3*/
    for (int i = 0 ; i < x ; + + i)/*4*/
    {                           /*5*/
        a[i] = a[i] * z ;       /*6/
        cout < < a [i] ;        /*6*/
    }                           /*6*/
}                               /*7*/
```



25

Cyclomatic Complexity Example Calculation

- **Method – 1**: V(G) = e – n + 2 (Where "e" are edges & "n" are nodes) V(G) = 8 – 7 + 2=3
- **Method – 2**: V(G) = P + 1 (Where P- predicate nodes with out degree = 2) V(G) = 2 + 1 = 3 (Nodes 1 and 5 are predicate nodes)
- **Method – 3**: V(G) = Number of enclosed regions + 1 = 2+1=3
  - V(G) = 3 and is same by all the three methods.
- **Conclusion – 1**: By getting a value of V(G) = 3 means that it is a "well written" code, its "testability" is high and cost / effort to maintain is low.
- **Conclusion – 2**: There are 3 paths in this program which are independent paths and they form a basis-set.
  - Path 1: 1 – 2 – 4 – 5 - 7
  - Path 2: 1 – 3 – 4 – 5 - 7
  - Path 3: 1 – 3 – 4 – 5 – 6 - 7

26

Graph Matrix

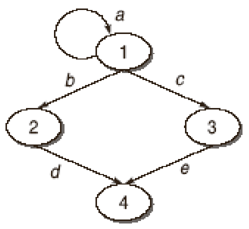
- Flow graph is an effective method of path testing.
- Moreover as the size of graph increases, manual path tracing becomes difficult and leads to error. A linked can be missed or covered twice.
- To avoid this Graph matrix is the solution to it.
- A graph matrix is a square matrix whose rows and columns are equal to the number of nodes in the flow graph.
- Each row and column identifies a particular node and matrix entries represent a connection between the nodes.

Graph Matrix

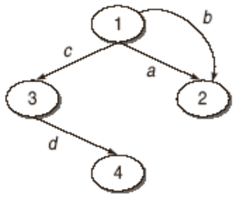
- The following points describe a graph matrix:
  - Each cell in the matrix can be a directed connection or link between one node to another node.
  - If there is a connection from node ‘a’ to node ‘b’, then it does not means that there is a connection from node ‘b’ to node ‘a’.
  - Conventionally, to represent a graph matrix, digits are used for nodes and letter symbols for edges or connection.
  - It can be used for finding set of all paths.
  - It may be noted that if there are several links between two nodes then " + " sign denotes a parallel link.

Graph Matrix

- Consider the following graph and representation of the graph matrix.



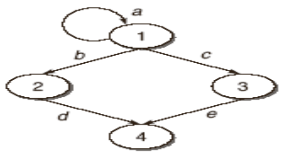
	1	2	3	4
1	a	b	c	
2				d
3				e
4				



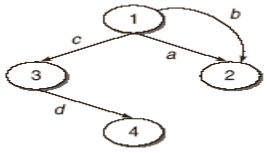
	1	2	3	4
1		a+b	c	
2				
3				d
4				

Connection Matrix

- If we add link weights to each cell entry, then graph matrix can be a powerful tool in testing.
- The links between two nodes are assigned a link weight which becomes the entry cell of matrix.
- If the connection exists, then the link weight is 1, otherwise 0.
- A matrix defined with link weight is called a connection matrix.



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

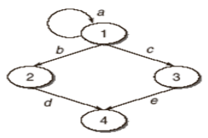


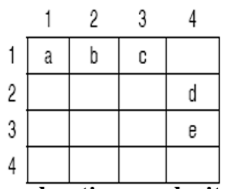
	1	2	3	4
1		1	1	
2				
3				1
4				

Connection Matrices to find Cyclomatic

1. For each row, count the number of 1s and write it in front of that row.
2. Subtract 1 from that count. Ignore the blank row, if any.
3. Add the final count of each row.
4. Add 1 to the sum calculated in step 3.
5. The final sum in step 4 is cyclomatic number of the graph.

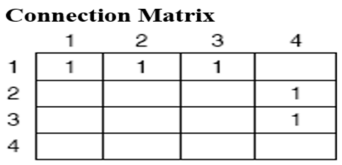
Graph Matrix ,





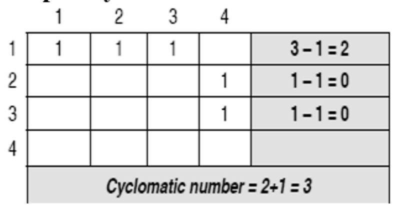
	1	2	3	4
1	a	b	c	
2				d
3				e
4				

Connection Matrix



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

Connection Matrix for cycolmatic complexity



	1	2	3	4	
1	1	1	1		3 – 1 = 2
2				1	1 – 1 = 0
3				1	1 – 1 = 0
4					
	Cyclomatic number = 2+1 = 3				

Data Flow testing

- Data flow testing is another form of structural testing. Here, we concentrate on the usage of variables and the focus points are:
    1. Statements where variables receive values
    2. Statements where these values are used or referenced
- Flow graphs are also used as a basis for the data flow testing as in the case of path testing.
- With a program data definition faults are nearly as frequent (22% of all faults) as control flow faults (24 percent).
- As we know variables are defined and referenced throughout the program. We may have few define/reference anomalies:
    1. A variable is defined but not used/referenced
    2. A variable is used but never defined
    3. A variable is defined twice before it is used

State of a Data Object

- **Defined (d)** : A data object is called defined when it is initialized, i.e. when it is on the left side of assignment statement. Defined state can also be used to mean that a file has been opened, a dynamically allocated objects has been allocated, something is pushed onto the stack, a record is written, and so on.
- **Killed/Undefined/Released (k)** : When the data has been reinitialized or the scope of a loop control variable finishes, i.e. existing the loop or memory is released dynamically or a file has been closed.
- **Usage (u)** : When the data object is on the right side of assignment or used as a control variable in a loop, or in an expression used to evaluate the control flow of a case statement, or as a pointer to an object, etc. In general, we say that the usage is either computational use **(c-use)** or predicate use **(p-use)**.

Two character Data Flow Anomalies:

Anomaly	Explanation	Effect of Anomaly
du	Define- use	Allowed. Normal Case
dk	Define- kill	Potential bug. Data is killed without use after definition.
ud	use –Define	Data is used and then redefined. Allowed. Usually not a bug because the language permits reassignment at almost any time.
uk	Use- kill	Allowed. Normal situation
ku	kill –Use	Serious bug because the data is used after being killed
kd	kill -Define	Data is killed and then redefined. Allowed.
dd	Define- Define	Redefining a variable without using it. Harmless bug, but not allowed.
uu	Use- use	Allowed. Normal Case
kk	Kill – kill	Harmless bug, but not allowed.

Terminology used in Data Flow Testing

- **Definition node**: Defining a variable means assigning value to a variable for the very first time in the program. For example, input statements, assignment statements, loop control statements, procedure calls, etc.
- **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 variable v is used at the statement corresponding to node n. for example, output statements, assignment statements (right), conditional statements , loop control statements, etc.
- A usage node can be of the following two types:
  - **Predicate usage node (p-use)** : if usage node n is predicate node, then n is predicated usage node. The predicate use or p-use is assigned to both branches out of the decision statements.

35

Terminology used in Data Flow Testing

  - **Computation usage node (c-use):** If usage node n corresponds to a computation statement in a program other than predicated, then it is called a computation usage node. c-use is when the variable appears on the RHS of an assignment statement. A c-use is said to occur on the assignment statement.
- **Loop free path segment** : It is path segment for which every node is visited once at most.
- **Simple path segment** : It is path segment in which at most one node is visited twice. A simple path segment is either loop-free or if there is a loop, only one node is involved.
- **Definition-use path (du-path)** : A du-path with respect to a variable v is a path between the definition node and the usage node of that variable. Usage node can be either p-use or c-use node.

36

Terminology used in Data Flow Testing

- **Definition-clear path (dc-path)** : A dc-path with respect to a variable v is a path between the definition node and the usage node such that no other node in the path is defining node of variable v.
- The du-path which are not dc-paths are important from testing view-point, as these are potential problematic spots for testing persons.
- Steps followed for DFT are given below:

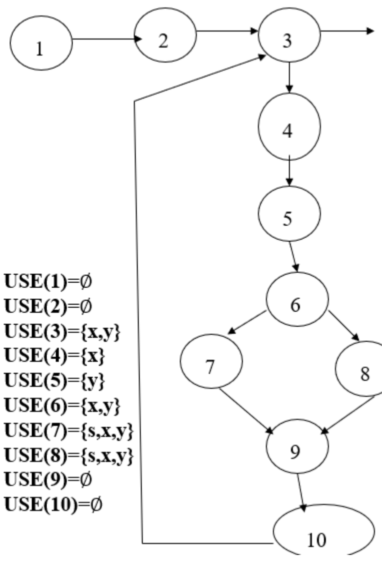
1. Draw the CFG
2. Prepare a table for define/use status of all variables used in your program.
3. Find all du-paths
4. Identify du-paths that are not dc-paths.

37

Data Flow Testing

```
1.  s:=0;
2.  x:=0;
3.  while (x<y) {
4.  x:=x+3;
5.  y:=y+2;
6.  if (x+y<10){
7.  s:=s+x+y;
    else
8.  s:=s+x-y;
9.  }
10. }
```

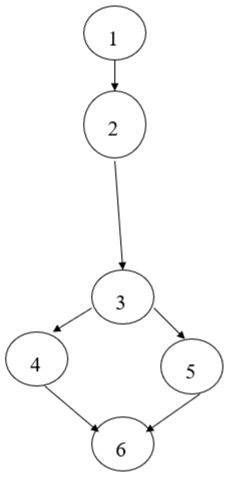
DEF(1)={s}	USE(1)=∅
DEF(2)={x}	USE(2)=∅
DEF(3)=∅	USE(3)={x,y}
DEF(4)={x}	USE(4)={x}
DEF(5)={y}	USE(5)={y}
DEF(6)=∅	USE(6)={x,y}
DEF(7)={s}	USE(7)={s,x,y}
DEF(8)={s}	USE(8)={s,x,y}
DEF(9)=∅	USE(9)=∅
DEF(10)= ∅	USE(10)=∅



38

Data Flow Testing

	Def	C-use	P-use
1.Read (x,y)	x, y		
2. z=x+2	z	x	
3. If(z<y)			z, y
4. w=x+1	w	x	
else			
5. y=y+1	y	y	
6. printf(x,y,w,z)		x, y, w,z	



$$\begin{tikzcd}
 & 1 \arrow[d] & \\
 & 2 \arrow[d] & \\
 & 3 \arrow[dl] \arrow[dr] & \\
4 \arrow[dr] & & 5 \arrow[dl] \\
 & 6 &
\end{tikzcd}$$