



**SE-3002**

**SOFTWARE QUALITY ENGINEERING**

**RUBAB JAFFAR**

[RUBAB.JAFFAR@NU.EDU.PK](mailto:RUBAB.JAFFAR@NU.EDU.PK)

# **Part II-Software Testing**

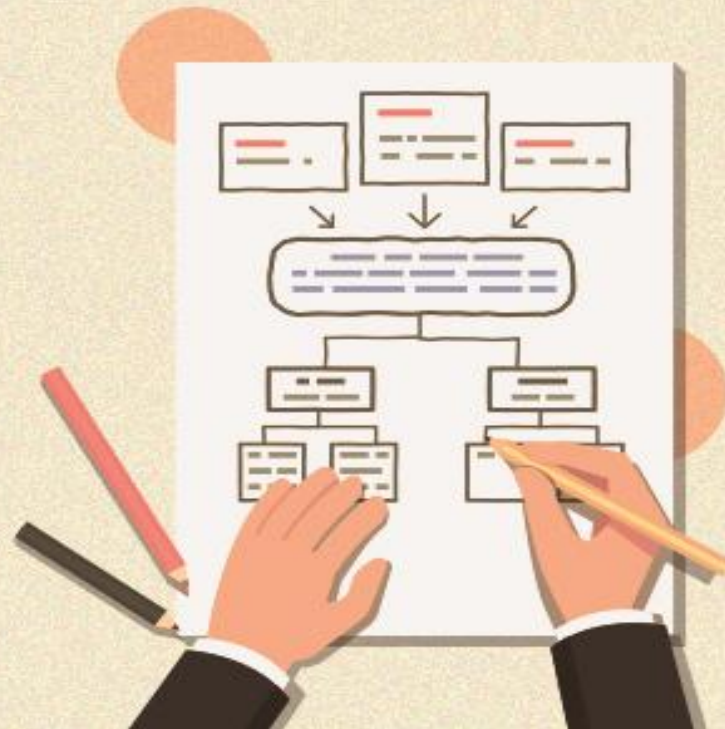
Structural Testing

## **Lecture # 25, 26, 27**

## **8,9,11 Nov**

# TODAY'S OUTLINE

- Structural Testing
  - Data Flow Testing
  - Slice based testing
  - Mutation testing



# WHAT IS **DATA FLOW** TESTING?

---

APPLICATION, EXAMPLES  
AND STRATEGIES

# DATA FLOW TESTING

- In control flow testing, we find various paths of a program and design test cases to execute those paths.
- We may like to execute every statement of the program at least once before the completion of testing.
- Consider the following program:

```
1. # include <stdio.h>
2. void main ()
3. {
4.   int a, b, c;
5.   a = b + c;
6.   printf ("%d", a);
7. }
```

# DATA FLOW TESTING

- Data flow testing may help us to minimize such mistakes. It is done to cover the path testing and branch testing gap.
- It has nothing to do with dataflow diagrams. It is based on variables, their usage and their definition(s) (assignment) in the program.
- The main points of concern are:
  - Statements where these values are used (referenced).
  - Statements where variables receive values (definition).
- Data flow testing focuses on variable definition and variable usage.
- The process is conducted to detect the bugs because of the incorrect usage of data variables or data values.

# DEFINE/REFERENCE ANOMALIES

- Some of the define / reference anomalies are given as:
  - A variable is defined but never used / referenced.
  - A variable is used but never defined.
  - A variable is defined twice before it is used.
  - A variable is used before even first-definition.
- Define / reference anomalies may be identified by static analysis of the program.

# DATA FLOW TESTING TERMS DEFINITIONS

- A program is first converted into a program graph.
- Defining node
  - A node of a program graph is a defining node for a variable , if and only if, the value of the variable is defined in the statement corresponding to that node. It is represented as  $DEF ( , n)$  where is the variable and n is the node corresponding to the statement in which is defined.
- Usage node
  - A node of a program graph is a usage node for a variable , if and only if, the value of the variable is used in the statement corresponding to that node. It is represented as  $USE ( , n)$ , where ' ' is the variable and 'n' in the node corresponding to the statement in which ' ' is used.
  - A usage node  $USE ( , n)$  is a predicate use node (denoted as P-use), if and only if, the statement corresponding to node 'n' is a predicate statement otherwise  $USE ( , n)$  is a computation use node (denoted as C-use).

# DATA FLOW TESTING TERMS DEFINITIONS

- Definition use Path
- A definition use path (denoted as du-path) for a variable ' ' is a path between two nodes 'm' and 'n' where 'm' is the initial node in the path but the defining node for variable ' ' (denoted as DEF ( , m)) and 'n' is the final node in the path but usage node for variable ' ' (denoted as USE ( , n)).



# IDENTIFICATION OF DU PATHS

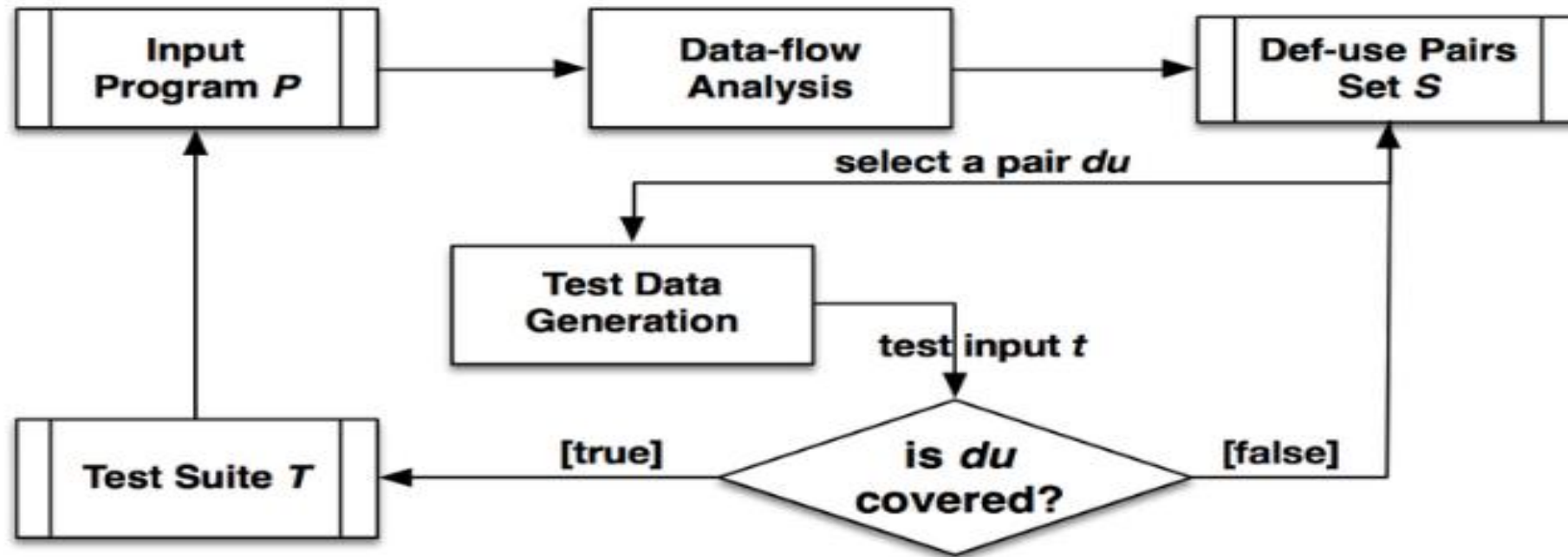
- The various steps for the identification of du and dc paths are given as:
- Draw the program graph of the program.
- Find all variables of the program and prepare a table for define / use status of all variables using the following format:

S. No.	Variable(s)	Defined at node	Used at node

- Generate all du-paths from define/use variable table of above step using the following

S. No.	Variable	du-path(begin, end)

# DATA FLOW TESTING CYCLE



# TESTING STRATEGIES USING DU-PATHS

- We want to generate test cases which trace every definition to each of its use and every use is traced to each of its definition. Some of the testing strategies are given as:
- **Test all du-paths**
- All du-paths generated for all variables are tested. This is the strongest data flow testing strategy covering all possible du-paths.
- **Test all uses**
- Find at least one path from every definition of every variable to every use of that variable which can be reached by that definition.
- For every use of a variable, there is a path from the definition of that variable to the use of that variable.

# TESTING STRATEGIES USING DU-PATHS

- **Test all definitions**
- Find paths from every definition of every variable to at least one use of that variable;
- The first requires that each definition reaches all possible uses through all possible du-paths, the second requires that each definition reaches all possible uses, and the third requires that each definition reaches at least one use.

# TYPES OF DATA FLOW TESTING

- Static Data Flow Testing
- No actual execution of the code is carried out in Static Data Flow testing. Generally, the definition, and usage pattern of the data variables is scrutinized through a control flow graph.
- Dynamic Data Flow Testing
- The code is executed to observe the transitional results. Dynamic data flow testing includes:
  - Identification of definition and usage of data variables.
  - Identifying viable paths between definition and usage pairs of data variables.
  - Designing & crafting test cases for these paths.

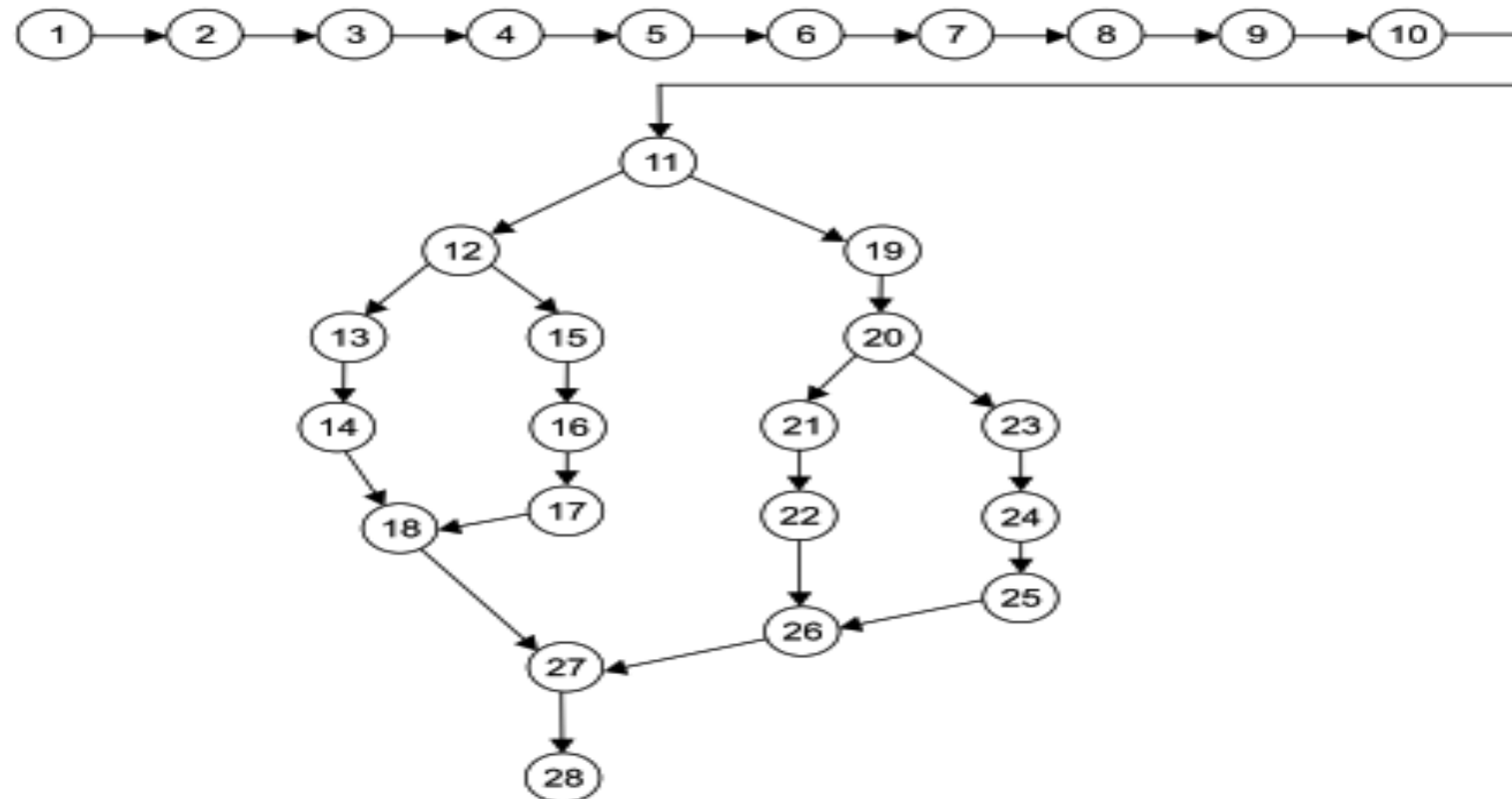
# DATA FLOW TESTING LIMITATIONS

- Testers require good knowledge of programming.
- Time-consuming
- Costly process.

## EXAMPLE: FIND THE LARGEST NUMBER AMONGST THREE NUMBERS.

```
1.      #include<stdio.h>
2.      #include<conio.h>
3.      void main()
4.      {
5.          float A,B,C;
6.          clrscr();
7.          printf("Enter number 1:\n");
8.          scanf("%f", &A);
9.          printf("Enter number 2:\n");
10.         scanf("%f", &B);
11.         printf("Enter number 3:\n");
12.         scanf("%f", &C);
13.         /*Check for greatest of three numbers*/
14.         if(A>B) {
15.             if(A>C) {
16.                 printf("The largest number is: %f\n",A);
17.             }
18.         }
19.         else {
20.             if(C>B) {
21.                 printf("The largest number is: %f\n",C);
22.             }
23.         }
24.         else {
25.             printf("The largest number is: %f\n",B);
26.         }
27.         getch();
28.     }
```

# STEP I





## STEP II & III

S. No.	Variable	Defined at node	Used at node
1.	A	6	11, 12, 13
2.	B	8	11, 20, 24
3.	C	10	12, 16, 20, 21

The du-paths with beginning node and end node are given as:

Variable	du-path (Begin, end)
A	6, 11
	6, 12
	6, 13
B	8, 11
	8, 20
	8, 24
C	10, 12
	10, 16
	10, 20
	10, 21

# TEST CASES

Test all du-paths					
S. No.	Inputs			Expected Output	Remarks
	A	B	C		
1.	9	8	7	9	6-11
2.	9	8	7	9	6-12
3.	9	8	7	9	6-13
4.	7	9	8	9	8-11
5.	7	9	8	9	8-11, 19, 20
6.	7	9	8	9	8-11, 19, 20, 23, 24
7.	8	7	9	9	10-12
8.	8	7	9	9	10-12, ,15, 16
9.	7	8	9	9	10, 11, 19, 20
10.	7	8	9	9	10, 11, 19-21

# SLICE BASED TESTING

- we prepare various subsets (called slices) of a program with respect to its variables and their selected locations in the program.
- Each variable with one of its location will give us a program slice.
- A large program may have many smaller programs (its slices), each constructed for different variable subsets.
- “Program slicing is a technique for restricting the behaviour of a program to some specified subset of interest. A slice  $S(, n)$  of program  $P$  on variable , or set of variables, at statement  $n$  yields the portions of the program that contributed to the value of just before statement  $n$  is executed.  $S(, n)$  is called a slicing criteria.
- Slices can be computed automatically on source programs by analyzing data flow.
- Hence, slices are smaller than the original program and may be executed independently.
- Only two things are important here, variable and its selected location in the program.

# GUIDELINES FOR SLICING

- All statements where variables are defined and redefined should be considered.
- All statements where variables receive values externally should be considered.
- All statements where output of a variable is printed should be considered.
- The status of all variables may be considered at the last statement of the program.

# CREATION OF PROGRAM SLICES

- Consider the portion of a program given in Figure 4.2 for the identification of its slices.
- $a = 3;$
- $b = 6;$
- $c = b^2;$
- $d = a^2 + b^2;$
- $c = a + b;$

# PROGRAM SLICES

- We identify two slices for variable 'c' at statement number 3 and statement number 5.

1.	a	=	3;
2.	b	=	6;
5.	c	=	a + b;
S(c, 5)			

Variable 'c' at statement 5

2.	b	=	6;
3.	c	=	b <sup>2</sup> ;
S(c, 3)			

Variable 'c' at statement 3

# EXAMPLE PROGRAM

- Consider the following program.

1. void main ( )
2. {
3. int a, b, c, d, e;
4. printf ("Enter the values of a, b and c \ n");
5. scanf ("%d %d %d", & a, &b, &c);
6. d = a+b;
7. e = b+c;
8. printf ("%d", d);
9. printf ("%d", e);
10. }

# SOME SLICES

```
1. main ( )
2. {
3.   int a, b, c, d, e;
4.   printf ("Enter the values of a, b and c \ n");
5.   scanf ("%d %d %d", &a, &b, &c);
7.   e = b + c;
9.   printf ("%d", e);
10. }
```

Slice on criterion S (c, 10) = (1, 2, 3, 4, 5, 7, 9, 10)

```
1. main ( )
2. {
3.   int a, b, c, d, e;
4.   printf ("Enter the values of a, b and c \ n");
5.   scanf ("%d %d %d", &a, &b, &c);
7.   e = b + c;
10. }
```

Slice on criterion S (c,7) = (1, 2, 3, 4, 5, 7,10)

```
1. main ( )
2. {
3.   int a, b, c, d, e;
4.   printf ("Enter the values of a, b and c \ n");
5.   scanf ("%d %d %d", &a, &b, &c);
10. }
```

Slice on criterion S (a, 5) = (1, 2, 3, 4, 5, 10)

```
1. main ( )
2. {
3.   int a, b, c, d, e;
4.   printf ("Enter the values of a, b and c \ n");
5.   scanf ("%d %d %d", &a, &b, &c);
6.   d = a + b;
8.   printf ("%d", d);
10. }
```

Slice on criterion S (d,10) = (1, 2, 3, 4, 5, 6, 8, 10)

```
1. main ( )
2. {
3.   int a, b, c, d, e;
4.   printf ("Enter the values of a, b and c \ n");
5.   scanf ("%d %d %d", &a, &b, &c);
6.   d = a + b;
10. }
```

Slice on criterion S (d,6) = (1, 2, 3, 4, 5, 6, 10)



# GENERATION OF TEST CASES

- Every slice should be independently executable and may cover some lines of source code of the program.
- The test cases for the slices of the program given
- Generated slices to find the largest number amongst three numbers are S(A, 6), S(A, 13), S(A, 28), S(B, 8), S(B, 24), S(B, 28), S(C, 10), S(C, 16), S(C, 21), S(C, 28).

# TEST CASES

. Test cases using program slices of program to find the largest among three numbers						
S. No.	Slice	Lines covered	A	B	C	Expected output
1.	S(A, 6)	1-6, 28	9			No output
2.	S(A, 13)	1-14, 18, 27, 28	9	8	7	9
3.	S(A, 28)	1-14, 18, 27, 28	8	8	7	9
4.	S(B, 8)	1-4, 7, 8, 28		9		No output
5.	S(B, 24)	1-11, 18-20, 22-28	7	9	8	9
6.	S(B, 28)	1-11, 19, 20, 23-28	7	9	8	9
7.	S(C, 10)	1-4, 9, 10, 28			9	No output
8.	S(C, 16)	1-12, 14-18, 27, 28	8	7	9	9
9.	S(C, 21)	1-11, 18-22, 26-28	7	8	9	9
10.	S(C, 28)	1-11, 18-22, 26-28	7	8	9	9

# LIMITATIONS OF SLICE BASED TESTING

- Slice based testing is a popular structural testing technique and focuses on a portion of the program with respect to a variable location in any statement of the program.
- Hence slicing simplifies the way of testing a program's behaviour with respect to a particular subset of its variables.
- But slicing cannot test a behaviour which is not represented by a set of variables or a variable of the program.

# MUTATION TESTING

- Popular technique to assess the effectiveness of a test suite.
- A large number of test cases for any program can be generated but cant execute all of them due to time or resources.
- So the idea is to select a few test cases using any testing technique and prepare a test suite.
- But how to assess the effectiveness of a selected test suite? Is this test suite adequate for the program?
- If the test suite is not able to make the program fail, there may be one of the following reasons:
- The test suite is effective but hardly any errors are there in the program. How will a test suite detect errors when they are not there?
- The test suite is not effective and could not find any errors. Although there may be errors, they could not be detected due to poor selection of test suite. How will errors be detected when the test suite is not effective?

# MUTATION AND MUTANTS

- Mutation testing may help us to assess the effectiveness of a test suite and may also enhance the test suite, if it is not adequate for a program.
- Mutation is the process of changing a program.
- This change may be limited to one, two or very few changes in the program.

# MUTATION AND MUTANTS

- To mutate a program means to change a program. We generally make only one or two changes in order to assess the effectiveness of the selected test suite
- We may make many mutants of a program by making small changes in the program.
- Every mutant will have a different change in a program.
- Every change of a program may give a different output as compared to the original program.
- The original program and mutant are syntactically correct and should compile correctly.

## EXAMPLE: FIND THE LARGEST NUMBER AMONGST THREE NUMBERS.

```
1.      #include<stdio.h>
2.      #include<conio.h>
3.      void main()
4.      {
5.          float A,B,C;
6.          clrscr();
7.          printf("Enter number 1:\n");
8.          scanf("%f", &A);
9.          printf("Enter number 2:\n");
10.         scanf("%f", &B);
11.         printf("Enter number 3:\n");
12.         scanf("%f", &C);
13.         /*Check for greatest of three numbers*/
14.         if(A>B) {
15.             if(A>C) {
16.                 printf("The largest number is: %f\n",A);
17.             }
18.         }
19.         else {
20.             if(C>B) {
21.                 printf("The largest number is: %f\n",C);
22.             }
23.         }
24.         else {
25.             printf("The largest number is: %f\n",B);
26.         }
27.         getch();
28.     }
```

# FIRST ORDER MUTANT OF THE EXAMPLE PROGRAM

- Many changes can be made in the program.
- Mutant  $M_1$  is obtained by replacing the operator ' $>$ ' of line number 11 by the operator ' $=$ '.
- Mutant  $M_2$  is obtained by changing the operator ' $>$ ' of line number 20 to operator ' $<$ '.
- These changes are simple changes. Only one change has been made in the original program to obtain mutant  $M_1$  and mutant  $M_2$ .

```
/*Check for greatest of three numbers*/  
11. if(A>B){ ← if(A=B) { mutated statement ('>' is replaced by '=')  
12.   if(A>C) {
```

$M_1$ : First order mutant

```
19.   else {  
20.   if(C>B) { ← if(C<B) { mutated statement ('>' is replaced by '<')  
21.     printf("The largest number is: %f\n",C);
```

$M_2$ : First order mutant



# HIGH ORDER MUTANT

- The mutants generated by making only one change are known as first order mutants.
- We may obtain second order mutants by making two simple changes in the program and third order mutants by making three simple changes, and so on.
- The second order mutant ( $M_3$ ) of the example program can be obtained by making two changes in the program and thus changing operator ' $>$ ' of line number 11 to operator ' $<$ ' and operator ' $>$ ' of line number 20 to ' $>=$ '.
- The second order mutants and above are called higher order mutants.
- Generally, in practice, we prefer to use only first order mutants in order to simplify the process of mutation.

## SECOND ORDER MUTANT

```
11.    if(A>B) { ← if(A<B) { mutated statement (replacing '>' by '<')
12.        if(A>C) {
13.            printf("The largest number is: %f\n",A);
14.        }
15.    else {
16.        printf("The largest number is: %f\n",C);
17.    }
18.    }
19.    else {
20.    if(C>B) { ← if(C≥B) { mutated statement (replacing '>' by '≥')
21.        printf("The largest number is: %f\n",C);
```

# MUTATION OPERATORS

- Mutants are produced by applying mutant operators. An operator is essentially a grammatical rule that changes a single expression to another expression. The changed expression should be grammatically correct as per the used language.
- If one or more mutant operators are applied to all expressions of a program, we may be able to generate a large set of mutants.
- We should measure the degree to which the program is changed. If the original expression is  $x + 1$ , and the mutant for that expression is  $x + 2$ , that is considered as a lesser change as compared to a mutant where the changed expression is  $(y * 2)$  by changing both operands and the operator.
- If  $x - y$  is changed to  $x - 5$  to make a mutant, then we should not use the value of  $y$  to be equal to 5. If we do so, the fault will not be revealed.

# MUTATION OPERATORS

- Some of the mutant operators for object oriented languages like Java, C++ are given as:
  - Static modifier change
  - Changing the access modifier, like public to private.
  - Argument order change
  - Super Keyword change
  - Operator change
  - Any operand change by a numeric value.

# MUTATION SCORE

- When we execute a mutant using a test suite, we may have any of the following outcomes:
  - The results of the program are affected by the change and any test case of the test suite detects it. If this happens, then the mutant is called a killed mutant.
  - The results of the program are not affected by the change and any test case of the test suite does not detect the mutation. The mutant is called a live mutant.
- The mutation score associated with a test suite and its mutants is calculated as:

$$\text{Mutation Score} = \frac{\text{Number of mutants killed}}{\text{Total number of mutants}}$$

# MUTATION SCORE

- The mutation score measures how sensitive the program is to the changes and how accurate the test suite is.
- A mutation score is always between 0 and 1.
- A higher value of mutation score indicates the effectiveness of the test suite although effectiveness also depends
- on the types of faults that the mutation operators are designed to represent.
- The live mutants are important for us and should be analyzed thoroughly.
- Why is it that any test case of the test suite not able to detect the changed behaviour of the program?
- One of the reasons may be that the changed statement was not executed by these test cases. If executed, then also it has no effect on the behaviour of the program.

## EXAMPLE: PROGRAM TO FIND THE LARGEST OF THREE NUMBERS

- Generate five mutants ( $M_1$  to  $M_5$ ) and calculate the mutation score of this test suite.

S. No.	A	B	C	Expected Output
1.	6	10	2	10
2.	10	6	2	10
3.	6	2	10	10
4.	6	10	20	20

# FIVE MUTANTS

## Mutated statements

Mutant No.	Line no.	Original line	Modified Line
M <sub>1</sub>	11	if(A>B)	if (A<B)
M <sub>2</sub>	11	if(A>B)	if(A>(B+C))
M <sub>3</sub>	12	if(A>C)	if(A<C)
M <sub>4</sub>	20	if(C>B)	if(C=B)
M <sub>5</sub>	16	printf("The Largest number is:%f\n",C);	printf("The Largest number is:%f\n",B);



# PROGRAM RESULTS AFTER EXECUTING MUTANTS

Actual output of mutant $M_1$					
Test case	A	B	C	Expected output	Actual output
1.	6	10	2	10	6
2.	10	6	2	10	6
3.	6	2	10	10	10
4.	6	10	20	20	20

Actual output of mutant $M_2$					
Test case	A	B	C	Expected output	Actual output
1.	6	10	2	10	10
2.	10	6	2	10	10
3.	6	2	10	10	10
4.	6	10	20	20	20

# PROGRAM RESULTS AFTER EXECUTING MUTANTS

Actual output of mutant $M_3$					
Test case	A	B	C	Expected output	Actual output
1.	6	10	2	10	10
2.	10	6	2	10	2
3.	6	2	10	10	6
4.	6	10	20	20	20

Actual output of mutant $M_4$					
Test case	A	B	C	Expected output	Actual output
1.	6	10	2	10	10
2.	10	6	2	10	10
3.	6	2	10	10	10
4.	6	10	20	20	10

Actual output of mutant $M_5$					
Test case	A	B	C	Expected output	Actual output
1.	6	10	2	10	10
2.	10	6	2	10	10
3.	6	2	10	10	2
4.	6	10	20	20	20

# MUTATION SCORE

$$\begin{aligned}\text{Mutation Score} &= \frac{\text{Number of mutants killed}}{\text{Total number of mutants}} \\ &= \frac{4}{5} \\ &= 0.8\end{aligned}$$



That is all