Software Quality Assurance and Testing Lecture - 04

ABDUS SATTER
LECTURER
INSTITUTE OF INFORMATION TECHNOLOGY
UNIVERSITY OF DHAKA

Dynamic Testing

WHITE BOX TESTING TECHNIQUES

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.

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. Therefore the intention in white-box testing is to cover the whole logic. The basic forms of logic coverage are
  - Statement Coverage
  - Decision or Branch Coverage
  - Condition Coverage
  - Decision/condition Coverage

Statement Coverage

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

Test case 1: x= y= n, where n is any number
Test case 2: x= n, y= n', where n and n' are different numbers.

Test case 3: x > y
Test case 4: x < y

Decision or Branch Coverage

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

Test case 1: x= y
Test case 2: x!= y
Test case 3: x < y
Test case 4: x > y

Condition Coverage

- Condition coverage states that each condition in a decision takes on all possible outcomes at least once. For example, consider the following statement:

  while ((I <5) && (J <COUNT))

- In this loop statement, two conditions are there. So test cases should be designed such that both the conditions are tested for True and False outcomes.
- The following test cases are designed:
  - Test case 1: I <5, J <COUNT
  - Test case 2: I >5, J >COUNT

Decision/condition Coverage

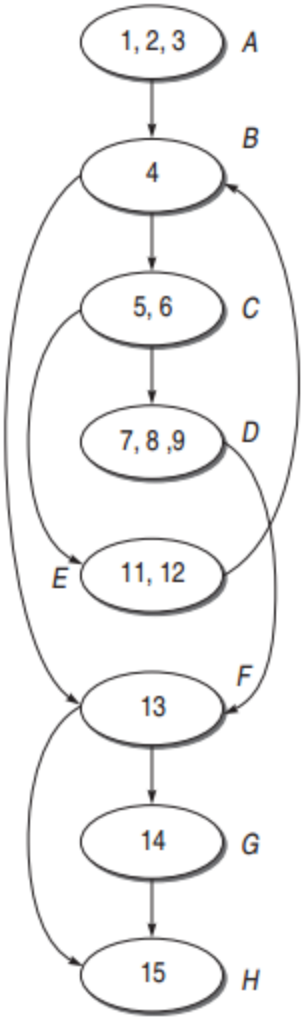
- if (A && B) is being tested, the condition coverage would allow one to write two test cases:
  - Test case 1: A is True, B is False.
  - Test case 2: A is False, B is True.
- But these test cases would not cause the THEN clause of the IF to execute (i.e. execution of decision). The obvious way out of this dilemma is a criterion called decision/condition coverage. It requires sufficient test cases such that each condition in a decision takes on all possible outcomes at least once, each decision takes on all possible outcomes at least once, and each point of entry is invoked at least once
  - Test case 1: A is True, B is True
  - Test case 1: A is False, B is False

Multiple Condition Coverage

- In case of multiple conditions, even decision/condition coverage fails to exercise all outcomes of all conditions. The reason is that we have considered all possible outcomes of each condition in the decision, but we have not taken all combinations of different multiple conditions. Certain conditions mask other conditions. For example, if an AND condition is False, none of the subsequent conditions in the expression will be evaluated. Similarly, if an OR condition is True, none of the subsequent conditions will be evaluated.
- For (A && B)
  - Test case 1: A= True, B= True
  - Test case 2: A= True, B= False
  - Test case 3: A= False, B= True
  - Test case 4: A= False, B= False

CFG

```
main()
{
     int number, index;
1.   printf("Enter a number");
2.   scanf("%d, &number);
3.   index = 2;
4.   while(index <= number - 1)
5.   {
6.        if (number % index == 0)
7.        {
8.             printf("Not a prime number");
9.             break;
10.       }
11.       index++;
12.       }
13.       if(index == number)
14.            printf("Prime number");
15. } //end main
```

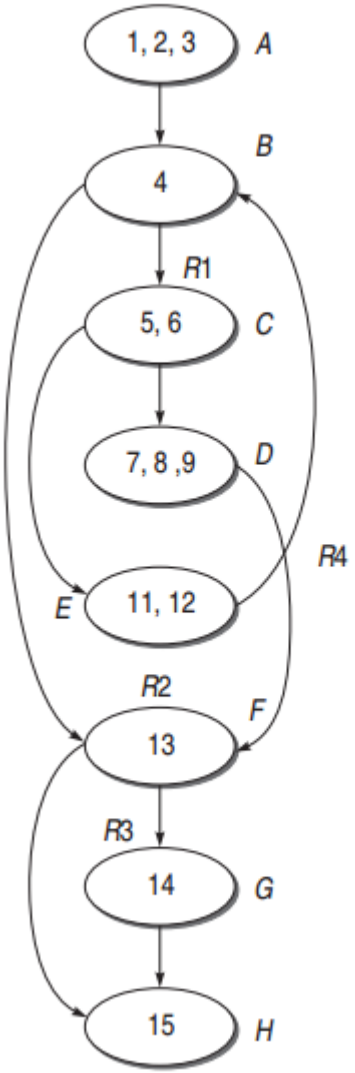


Cyclomatic Complexity

$$V(G) = e - n + 2 * p$$
$$= 10 - 8 + 2$$
$$= 4$$

$V(G)$ = Number of predicate nodes + 1
= 3 (Nodes B, C, and F) + 1
= 4

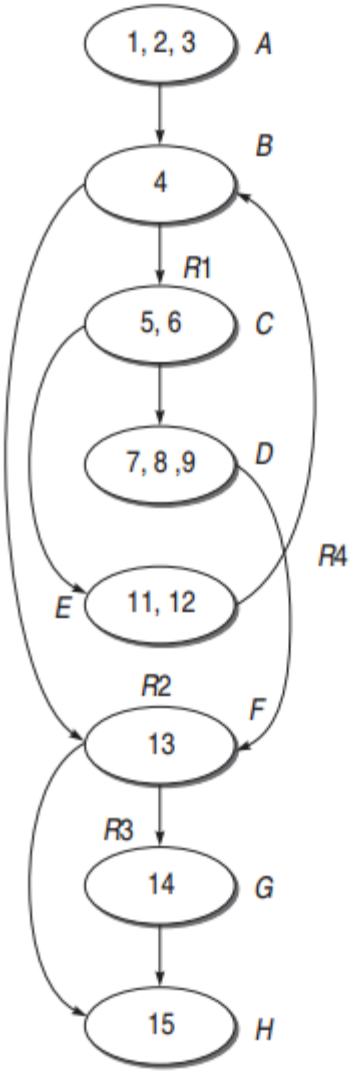
$V(G)$ = Number of regions
= 4($R1$, $R2$, $R3$, $R4$)



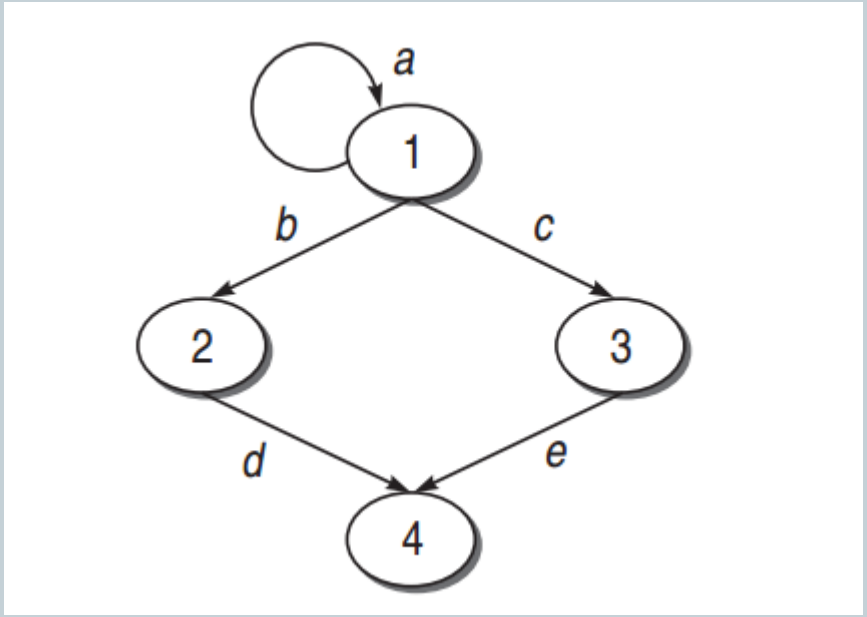
Independent Path & Test Cases

- (i) A-B-F-H
- (ii) A-B-F-G-H
- (iii) A-B-C-E-B-F-G-H
- (iv) A-B-C-D-F-H

Test case ID	Input num	Expected result	Independent paths covered by test case
1	1	No output is displayed	A-B-F-H
2	2	Prime number	A-B-F-G-H
3	4	Not a prime number	A-B-C-D-F-H
4	3	Prime number	A-B-C-E-B-F-G-H

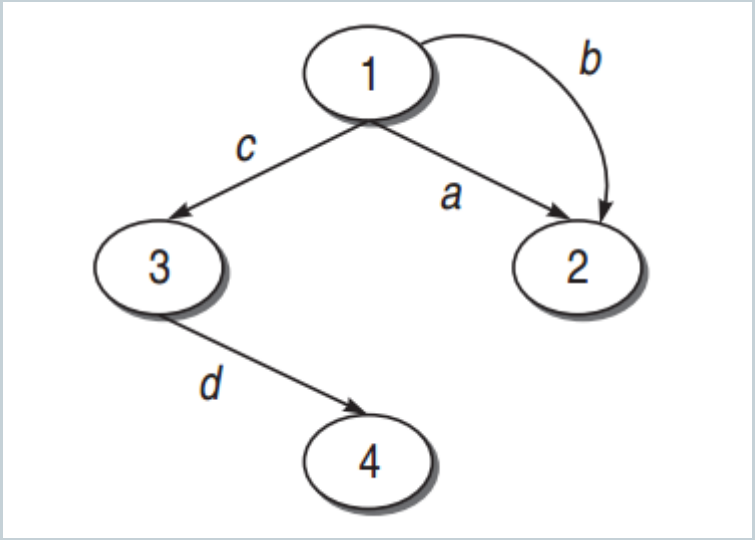


Graph Matrix



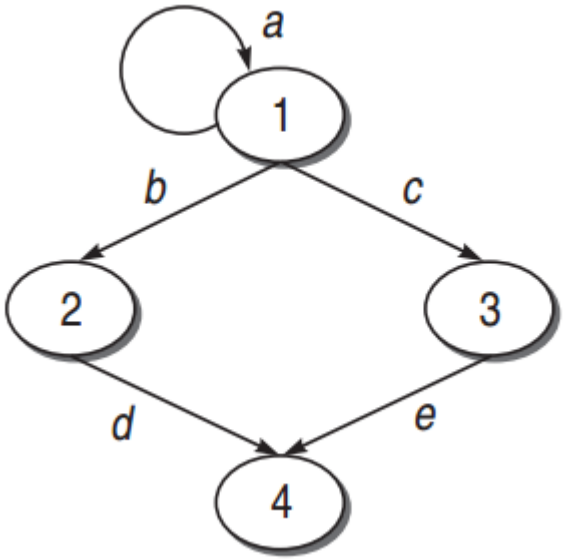
	1	2	3	4
1	a	b	c	
2				d
3				e
4				

Graph Matrix



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

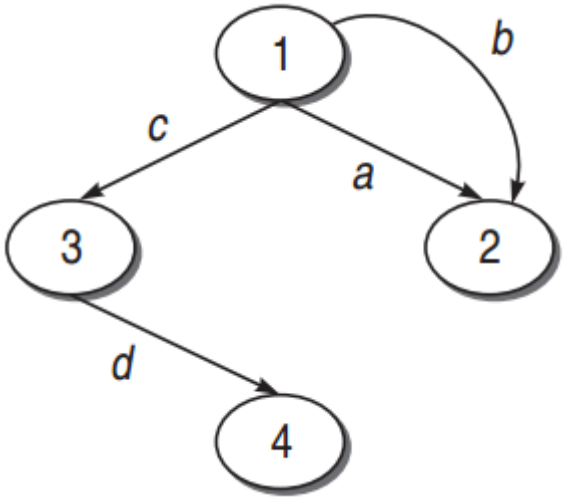
Connection Matrix



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

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

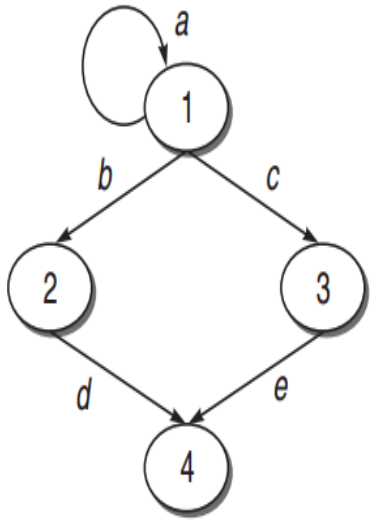
Connection Matrix



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

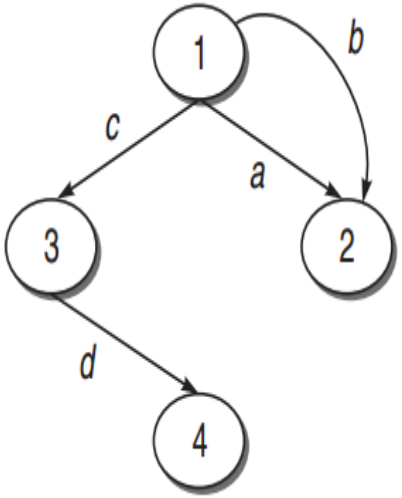
	1	2	3	4
1		1	1	
2				
3				1
4				

Use Of Connection Matrix In Finding Cyclomatic Complexity Number



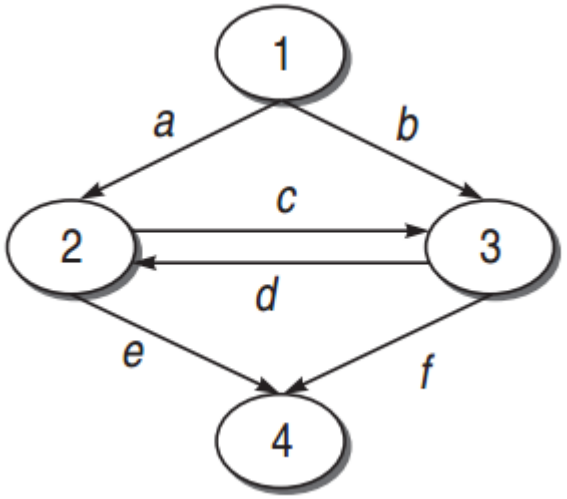
	1	2	3	4	
1	1	1	1		3 – 1 = 2
2				1	1 – 1 = 0
3				1	1 – 1 = 0
4					
	Cyclomatic number = 2+1 = 3				

Use Of Connection Matrix In Finding Cyclomatic Complexity Number



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

Use of Graph Matrix to Find K-Link Paths



$$\begin{pmatrix} 0 & a & b & 0 \\ 0 & 0 & c & e \\ 0 & d & 0 & f \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

$$\begin{pmatrix} 0 & a & b & 0 \\ 0 & 0 & c & e \\ 0 & d & 0 & f \\ 0 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} 0 & a & b & 0 \\ 0 & 0 & c & e \\ 0 & d & 0 & f \\ 0 & 0 & 0 & 0 \end{pmatrix} = \begin{pmatrix} 0 & bd & ac & ae+bf \\ 0 & cd & 0 & cf \\ 0 & 0 & dc & de \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

Loop Testing (Simple Loop)

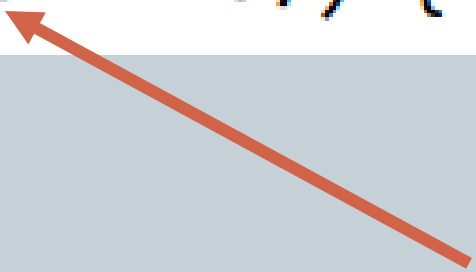
- Check whether you can bypass the loop or not. If the test case for bypassing the loop is executed and, still you enter inside the loop, it means there is a bug.
- Check whether the loop control variable is negative.
- Write one test case that executes the statements inside the loop.
- Write test cases for a typical number of iterations through the loop.
- Write test cases for checking the boundary values of the maximum and minimum number of iterations defined (say min and max) in the loop. It means we should test for min, min+1, min−1, max−1, max, and max+1 number of iterations through the loop.

Loop Testing (Nested Loop)

- Adopt the approach of simple tests to test the nested loops
- Start with the innermost loops while holding outer loops to their minimum values
- Continue this outward in this manner until all loops have been covered

Data Flow Testing

```
int a;
if(a == 67) { }
```



Potential Bug

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 an assignment statement. Defined state can also be used to mean that a file has been opened, a dynamically allocated object has been allocated, something is pushed onto the stack, a record written, and so on [9].
- **Killed/Undefined/Released (k):** When the data has been reinitialized or the scope of a loop control variable finishes, i.e. exiting 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).

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.**

Data-Flow Anomalies

- ~x: indicates all prior actions are not of interest to x.
- x~ : indicates all post actions are not of interest to x.

Anomaly	Explanation	Effect of Anomaly
~d	First definition	Normal situation. Allowed.
~u	**First Use**	**Data is used without defining it. Potential bug.**
~k	**First Kill**	**Data is killed before defining it. Potential bug.**
D~	**Define last**	**Potential bug.**
U~	Use last	Normal case. Allowed.
K~	Kill last	Normal case. Allowed.

Data Flow Testing

```
     main()
     {
1.       float bs, gs, da, hra = 0;
2.       printf("Enter basic salary");
3.       scanf("%f", &bs);
4.       if(bs < 1500)
5.       {
6.            hra = bs * 10/100;
7.            da = bs * 90/100;
8.       }
9.       else
10.      {
11.           hra = 500;
12.           da = bs * 98/100;
13.      }
14.      gs = bs + hra + da;
15.      printf("Gross Salary = Rs. %f", gs);
16.  }
```

For Variable bs

Pattern	Line Number	Explanation
~d	3	Normal Case, Allowed
du	3-4	Normal Case, Allowed
uu	4-6, 6-7, 7-14, 4-12, 12-14	Normal Case, Allowed
uk	14-16	Normal Case, Allowed
K~	16	Normal Case, Allowed

Mutation Testing

- Mutation testing is the process of mutating some segment of code (putting some error in the code) and then, testing this mutated code with some test data. If the test data is able to detect the mutations in the code, then the test data is quite good, otherwise we must focus on the quality of test data. Therefore, mutation testing helps a user create test data by interacting with the user to iteratively strengthen the quality of test data.

Mutation Testing

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

M1:x = x – y;
M2:x = x / y;
M3:x = x + 1;
M4:printf("%d", y);

Test Case ID	x	y	Initial Program Result	Mutant Result
TD1	2	2	4	0 (M1)
TD2	4	3	7	1.4 (M2)
TD3	3	2	5	4 (M3)
TD4	5	2	7	2 (M4)

Thank You

END OF CHAPTER