MODULE 2: TESTING TECHNIQUES

-by
Asst. Prof Rohini M. Sawant

INTRODUCTION

- After devising a testing strategy, we need various testing techniques so that we can design test cases in order to perform testing.
- There are two views of testing techniques: one view is to categorize based on verification and validation; another is based on static and dynamic testing.
- Static testing is performed without executing the code, and dynamic testing is performed with the execution of code.
- Dynamic testing techniques, namely black-box and white-box techniques, are very popular which we study in this module.

Testing Category	Techniques
Dynamic testing: Black-Box	Boundary value analysis, Equivalence class partitioning, State table-based testing, Decision table-based testing, Cause-effect graphing technique, Error guessing.
Dynamic testing: White-Box	Basis path testing, Graph matrices, Loop testing, Data flow testing, Mutation testing.
Static testing	Inspection, Walkthrough, Reviews.
Validation testing	Unit testing, Integration testing, Function testing, System testing, Acceptance testing, Installation testing.
Regression testing	Selective retest technique, Test prioritization.

2.1 BLACK BOX TESTING TECHNIQUES

- Black-box technique is one of the major techniques in dynamic testing for designing effective test cases.
- This technique considers only the functional requirements of the software or module.
- In other words, the structure or logic of the software is not considered.
- Therefore, this is also known as functional testing.
- The software system is considered as a black box, taking no notice of its internal structure, so it is also called as black-box testing technique.

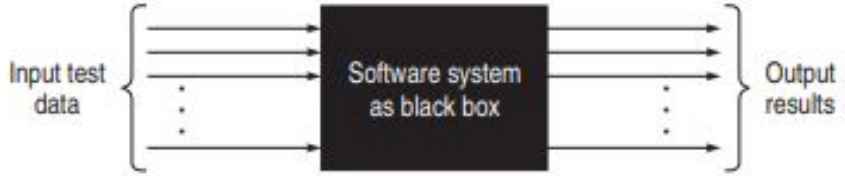


Figure 4.1 Black-box testing

2.1 BLACK BOX TESTING TECHNIQUES

Black-box testing attempts to find errors in the following categories:

- To test the modules independently.
- To test the functional validity of the software so that incorrect or missing functions can be recognized.
- To look for interface errors.
- To test the system behaviour and check its performance.
- To test the maximum load or stress on the system.
- To test the software such that the user/customer accepts the system within defined acceptable limits

2.1 BOUNDARY VALUE ANALYSIS (BVA)

- An effective test case design requires test cases to be designed such that they maximize the probability of finding errors.
- BVA technique addresses this issue. With the experience of testing team, it has been observed that test cases designed with boundary input values have a high chance to find errors.
- It means that most of the failures crop up due to boundary values. BVA is considered a technique that uncovers the bugs at the boundary of input values.
- Here, boundary means the maximum or minimum value taken by the input domain.

2.1 BOUNDARY VALUE ANALYSIS (BVA)

For example, if A is an integer between 10 and 255, then boundary checking can be on 10(9,10,11) and on 255(256,255,254). Similarly, B is another integer variable between 10 and 100, then boundary checking can be on 10(9,10,11) and 100(99,100,101), as shown in Fig. 4.2.

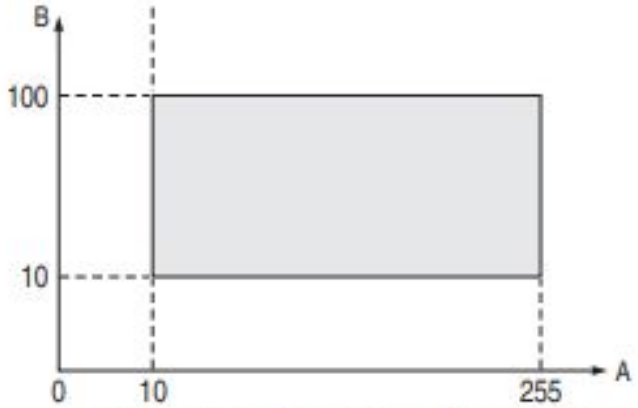


Figure 4.2 Boundary value analysis

2.1.1 BOUNDARY VALUE CHECKING (BVC)

In this method, the test cases are designed by holding one variable at its extreme value and other variables at their nominal values in the input domain.

The variable at its extreme value can be selected at:

(a) Minimum value (Min)

(b) Value just above the minimum value (Min+)

(c) Maximum value (Max)

(d) Value just below the maximum value (Max−)

2.1.1 BOUNDARY VALUE CHECKING (BVC)

Let us take the example of two variables, A and B. If we consider all the above combinations with nominal values, then following test cases can be designed:

1. Anom, Bmin
2. Anom, Bmin+
3. Anom, Bmax
4. Anom, Bmax–
5. Amin, Bnom
6. Amin+, Bnom
7. Amax, Bnom
8. Amax–, Bnom
9. Anom, Bnom

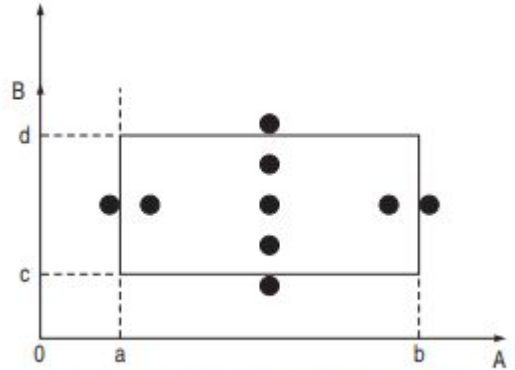


Figure 4.3 Boundary value checking

It can be generalized that for n variables in a module, **4n + 1 test cases** can be designed with boundary value checking method.

2.1.2 ROBUSTNESS TESTING METHOD

The idea of BVC can be extended such that boundary values are exceeded as:

- A value just greater than the Maximum value (Max+)
- A value just less than Minimum value (Min−)

When test cases are designed considering the above points in addition to BVC, it is called robustness testing.

Let us take the previous example again. Add the following test cases to the list of 9 test cases designed in BVC:

10. Amax+, Bnom
11. Amin−, Bnom
12. Anom, Bmax+
13. Anom, Bmin−

It can be generalized that for n input variables in a module, **6n + 1 test cases** can be designed with robustness testing.

2.1.3 WORST-CASE TESTING METHOD

We can again extend the concept of BVC by assuming more than one variable on the boundary.

The worst-case scenario is the set of conditions or inputs that would cause the system to experience its maximum stress or load. These conditions are typically extreme or unusual but are chosen to push the system to its limits.

It is called worst-case testing method. It can be generalized that for n input variables in a module, **5n test cases** are derived. Again, take the previous example of two variables, A and B. We can add the following test cases to the list of 9 test cases designed in BVC as:

10. A_{min}, B_{min}
11. A_{min+}, B_{min}
12. A_{min}, B_{min+}
13. A_{min+}, B_{min+}
14. A_{max}, B_{min}
15. A_{max-}, B_{min}
16. A_{max}, B_{min+}
17. A_{max-}, B_{min+}
18. A_{min}, B_{max}
19. A_{min+}, B_{max}
20. A_{min}, B_{max-}
21. A_{min+}, B_{max-}
22. A_{max}, B_{max}
23. A_{max-}, B_{max}
24. A_{max}, B_{max-}
25. A_{max-}, B_{max-}

Example 1: A program reads an integer number within the range [1,100] and determines whether it is a prime number or not. Design test cases for this program using BVC, robust testing, and worst-case testing methods.

(a) Test cases using BVC Since there is one variable, the total number of test cases will be 4n + 1 = 5.

In our example, the set of minimum and maximum values is sh
below:

Min value = 1
Min^{+} value = 2
Max value = 100
Max^{-} value = 99
Nominal value = 50–55

Using these values, test cases can be designed as shown below:

Test Case ID	Integer Variable	Expected Output
1	1	Not a prime number
2	2	Prime number
3	100	Not a prime number
4	99	Not a prime number
5	53	Prime number

(b) **Test cases using robust testing** Since there is one variable, the total number of test cases will be $6n + 1 = 7$. The set of boundary values is shown below:

Min value = 1
Min$^-$ value = 0
Min$^+$ value = 2
Max value = 100
Max$^-$ value = 99
Max$^+$ value = 101
Nominal value = 50–55

Using these values, test cases can be designed as shown below:

Test Case ID	Integer Variable	Expected Output
1	0	Invalid input
2	1	Not a prime number
3	2	Prime number
4	100	Not a prime number
5	99	Not a prime number
6	101	Invalid input
7	53	Prime number

(c) **Test cases using worst-case testing** Since there is one variable, the total number of test cases will be $5^n = 5$. Therefore, the number of test cases will be same as BVC.

Example 2: A program computes ab where a lies in the range [1,10] and b within [1,5]. Design test cases for this program using BVC, robust testing, and worst-case testing methods.

Solution:

(a) **Test cases using BVC** Since there are two variables, a and b, the total number of test cases will be $4n + 1 = 9$. The set of boundary values is shown below:

	a	b
Min value	1	1
Min$^+$ value	2	2
Max value	10	5
Max$^-$ value	9	4
Nominal value	5	3

Using these values, test cases can be designed as shown below:

Test Case ID	a	b	Expected Output
1	1	3	1
2	2	3	8
3	10	3	1000
4	9	3	729
5	5	1	5
6	5	2	25
7	5	4	625
8	5	5	3125
9	5	3	125

(b) **Test cases using robust testing** Since there are two variables, a and b, the total number of test cases will be $6n + 1 = 13$. The set of boundary values is shown below:

	a	b
Min value	1	1
Min⁻ value	0	0
Min⁺ value	2	2
Max value	10	5
Max⁺ value	11	6
Max⁻ value	9	4
Nominal value	5	3

Using these values, test cases can be designed as shown below:

Test Case ID	a	b	Expected output
1	0	3	Invalid input
2	1	3	1
3	2	3	8
4	10	3	1000
5	11	3	Invalid input
6	9	3	729
7	5	0	Invalid input
8	5	1	5
9	5	2	25
10	5	4	625
11	5	5	3125
12	5	6	Invalid input
13	5	3	125

(c) **Test cases using worst-case testing** Since there are two variables, a and b, the total number of test cases will be $5^n = 25$.

The set of boundary values is shown below:

	a	b
Min value	1	1
Min$^+$ value	2	2
Max value	10	5
Max$^-$ value	9	4
Nominal value	5	3

There may be more than one variable at extreme values in this case. Therefore, test cases can be designed as shown below:

Test Case ID	a	b	Expected Output
1	1	1	1
2	1	2	1
3	1	3	3
4	1	4	1
5	1	5	1
6	2	1	2
7	2	2	4
8	2	3	8
9	2	4	16
10	2	5	32
11	5	1	5
12	5	2	25
13	5	3	125
14	5	4	625
15	5	5	3125
16	9	1	9
17	9	2	81
18	9	3	729
19	9	4	6561
20	9	5	59049
21	10	1	10
22	10	2	100
23	10	3	1000
24	10	4	10000
25	10	5	100000

2.2 EQUIVALENCE CLASS TESTING

- Equivalence partitioning is a method for deriving test cases wherein classes of input conditions called equivalence classes are identified such that each member of the class causes the same kind of processing and output to occur.
- Thus, instead of testing every input, only one test case from each partitioned class can be executed.
- It means only one test case in the equivalence class will be sufficient to find errors.
- This test case will have a representative value of a class which is equivalent to a test case containing any other value in the same class.
- If one test case in an equivalence class detects a bug, all other test cases in that class have the same probability of finding bugs.

2.2 EQUIVALENCE CLASS TESTING

Equivalence partitioning method for designing test cases has the following goals:

- Completeness: Without executing all the test cases, we strive to touch the completeness of testing domain.
- Non-redundancy: When the test cases are executed having inputs from the same class, then there is redundancy in executing the test cases. Time and resources are wasted in executing these redundant test cases, as they explore the same type of bug.

To use equivalence partitioning, one needs to perform two steps:

1. Identify equivalence classes

2. Design test cases

2.2 EQUIVALENCE CLASS TESTING

IDENTIFICATION OF EQUIVALENT CLASSES

- Different equivalence classes are formed by grouping inputs for which the behaviour pattern of the module is similar.
- The rationale of forming equivalence classes like this is the assumption that if the specifications require exactly the same behaviour for each element in a class of values, then the program is likely to be constructed such that it either succeeds or fails for each value in that class.
- Two types of classes can always be identified as discussed below:
- **Valid equivalence classes**: These classes consider valid inputs to the program.
- **Invalid equivalence classes:** One must not be restricted to valid inputs only. We should also consider invalid inputs that will generate error conditions or unexpected behaviour of the program

2.2 EQUIVALENCE CLASS TESTING

IDENTIFYING THE TEST CASES

A few guidelines are given below to identify test cases through generated equivalence classes:

- Assign a unique identification number to each equivalence class.
- Write a new test case covering as many of the uncovered valid equivalence classes as possible, until all valid equivalence classes have been covered by test cases.
- Write a test case that covers one, and only one, of the uncovered invalid equivalence classes, until all invalid equivalence classes have been covered by test cases.

Example 1

A program determines the next date in the calendar. Its input is entered in the form of with the following range:

$1 \leq mm \leq 12$

$1 \leq dd \leq 31$

$1900 \leq yyyy \leq 2025$

Its output would be the next date or an error message ‘invalid date.’ Design test cases using equivalence class partitioning method.

Solution

First we partition the domain of input in terms of valid input values and invalid values, getting the following classes:

$$I_1 = \{<m, d, y> : 1 \le m \le 12\}$$
$$I_2 = \{<m, d, y> : 1 \le d \le 31\}$$
$$I_3 = \{<m, d, y> : 1900 \le y \le 2025\}$$
$$I_4 = \{<m, d, y> : m < 1\}$$
$$I_5 = \{<m, d, y> : m > 12\}$$
$$I_6 = \{<m, d, y> : d < 1\}$$
$$I_7 = \{<m, d, y> : d > 31\}$$
$$I_8 = \{<m, d, y> : y < 1900\}$$
$$I_9 = \{<m, d, y> : y > 2025\}$$

Test case ID	mm	dd	yyyy	Expected result	Classes covered by the test case
1	5	20	1996	21-5-1996	I_1, I_2, I_3
2	0	13	2000	Invalid input	I_4
3	13	13	1950	Invalid input	I_5
4	12	0	2007	Invalid input	I_6
5	6	32	1956	Invalid input	I_7
6	11	15	1899	Invalid input	I_8
7	10	19	2026	Invalid input	I_9

Example 2: A program reads three numbers, A, B, and C, with a range [1, 50] and prints the largest number. Design test cases for this program using equivalence class testing technique.

Solution:

1. First we partition the domain of input as valid input values and invalid values, getting the following classes:

$$I_1 = \{<A,B,C> : 1 \le A \le 50\}$$
$$I_2 = \{<A,B,C> : 1 \le B \le 50\}$$
$$I_3 = \{<A,B,C> : 1 \le C \le 50\}$$
$$I_4 = \{<A,B,C> : A < 1\}$$
$$I_5 = \{<A,B,C> : A > 50\}$$
$$I_6 = \{<A,B,C> : B < 1\}$$
$$I_7 = \{<A,B,C> : B > 50\}$$
$$I_8 = \{<A,B,C> : C < 1\}$$
$$I_9 = \{<A,B,C> : C > 50\}$$

Now the test cases can be designed from the above derived classes, taking one test case from each class such that the test case covers maximum valid input classes, and separate test cases for each invalid class.

The test cases are shown below:

Test case ID	A	B	C	Expected result	Classes covered by the test case
1	13	25	36	C is greatest	I_1, I_2, I_3
2	0	13	45	Invalid input	I_4
3	51	34	17	Invalid input	I_5
4	29	0	18	Invalid input	I_6
5	36	53	32	Invalid input	I_7
6	27	42	0	Invalid input	I_8
7	33	21	51	Invalid input	I_9

. We can derive another set of equivalence classes based on some possibilities for three integers, A, B, and C. These are given below:

$$I_1 = \{<A,B,C> : A > B, A > C\}$$
$$I_2 = \{<A,B,C> : B > A, B > C\}$$
$$I_3 = \{<A,B,C> : C > A, C > B\}$$
$$I_4 = \{<A,B,C> : A = B, A \neq C\}$$
$$I_5 = \{<A,B,C> : B = C, A \neq B\}$$
$$I_6 = \{<A,B,C> : A = C, C \neq B\}$$
$$I_7 = \{<A,B,C> : A = B = C\}$$

Test case ID	A	B	C	Expected Result	Classes Covered by the test case
1	25	13	13	A is greatest	I_1, I_5
2	25	40	25	B is greatest	I_2, I_6
3	24	24	37	C is greatest	I_3, I_4
4	25	25	25	All three are equal	I_7

2.3 STATE TABLE-BASED TESTING

- Tables are useful tools for representing and documenting many types of information relating to test case design.
- These are beneficial for the applications which can be described using state transition diagrams and state tables.

2.3.1 FINITE STATE MACHINE (FSM) An FSM is a behavioural model whose outcome depends upon both previous and current inputs. FSM models can be prepared for software structure or software behaviour. And it can be used as a tool for functional testing. Many testers prefer to use FSM model as a guide to design functional tests.

2.3.2 STATE TRANSITION DIAGRAMS OR STATE GRAPH

2.3 STATE TABLE-BASED TESTING

STATE TABLE: State graphs of larger systems may not be easy to understand. Therefore, state graphs are converted into tabular form for convenience sake, which are known as state tables.

It is a systematic approach in software testing used to test systems with complex state-dependent behavior. It involves creating and using a state table to represent the different states a system can be in and how it transitions from one state to another based on various inputs or events.

State tables also specify states, inputs, transitions, and outputs. The following conventions are used for state table:

- Each row of the table corresponds to a state.
- Each column corresponds to an input condition.
- The box at the intersection of a row and a column specifies the next state (transition) and the output, if any.

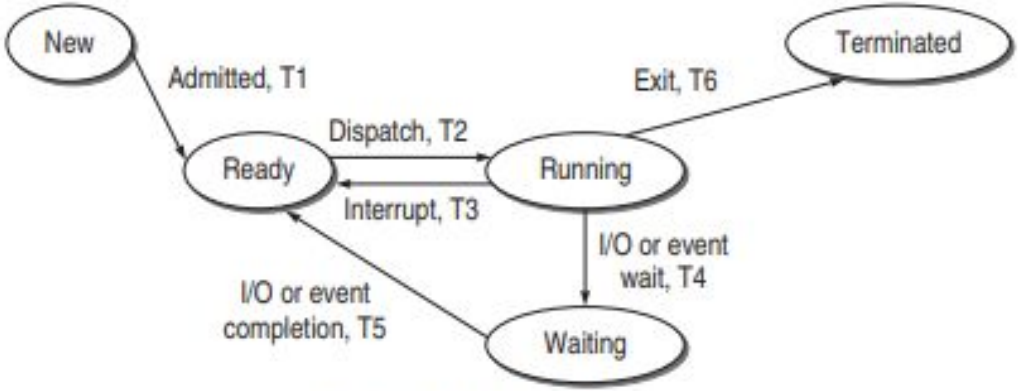


Figure 4.5 State graph

Table 4.1 State table

State\Input Event	Admit	Dispatch	Interrupt	I/O or Event Wait	I/O or Event Wait Over	Exit
New	**Ready/ T1**	New / T0	New / T0	New / T0	New / T0	New / T0
Ready	Ready/ T1	**Running/ T2**	Ready / T1	Ready / T1	Ready / T1	Ready / T1
Running	Running/T2	Running/ T2	**Ready / T3**	**Waiting/ T4**	Running/ T2	**Terminated/T6**
Waiting	Waiting/T4	Waiting / T4	Waiting/T4	Waiting / T4	**Ready / T5**	Waiting / T4

2.3 STATE TABLE-BASED TESTING

The steps to perform Testing are:

- Identify the states
- Prepare state transition diagram after understanding transitions between states
- Convert the state graph into the state table as discussed earlier
- Analyse the state table for its completeness
- Create the corresponding test cases from the state table

Test Source : a trace back to the corresponding cell in the state table

Current State : the initial condition to run the test

Event : the input triggered by the user

Output : the current value returned

Next State : the new state achieved

The test cases derived from the state table (Table 4.1) are shown below in Table 4.2.

Table 4.2 Deriving test cases from state table

Test Case ID	Test Source	Input		Expected Results	
		Current State	Event	Output	Next State
TC1	Cell 1	New	Admit	T1	Ready
TC2	Cell 2	New	Dispatch	T0	New
TC3	Cell 3	New	Interrupt	T0	New
TC4	Cell 4	New	I/O wait	T0	New
TC5	Cell 5	New	I/O wait over	T0	New
TC6	Cell 6	New	exit	T0	New
TC7	Cell 7	Ready	Admit	T1	Ready
TC8	Cell 8	Ready	Dispatch	T2	Running
TC9	Cell 9	Ready	Interrupt	T1	Ready
TC10	Cell 10	Ready	I/O wait	T1	Ready
TC11	Cell 11	Ready	I/O wait	T1	Ready
TC12	Cell 12	Ready	Exit	T1	Ready
TC13	Cell 13	Running	Admit	T2	Running
TC14	Cell 14	Running	Dispatch	T2	Running
TC15	Cell 15	Running	Interrupt	T3	Ready
TC16	Cell 16	Running	I/O wait	T4	Waiting
TC17	Cell 17	Running	I/O wait over	T2	Running
TC18	Cell 18	Running	Exit	T6	Terminated
TC19	Cell 19	Waiting	Admit	T4	Waiting
TC20	Cell 20	Waiting	Dispatch	T4	Waiting
TC21	Cell 21	Waiting	Interrupt	T4	Waiting
TC22	Cell 22	Waiting	I/O wait	T4	Waiting
TC23	Cell 23	Waiting	I/O wait over	T5	Ready
TC24	Cell 24	Waiting	Exit	T4	Waiting

CAUSE-EFFECT GRAPHING BASED TESTING

- Cause-Effect graphing is another technique for combinations of input conditions.
- It is the technique to represent the situations of combinations of input conditions and then we convert the cause-effect graph into decision table for the test cases.
- It helps in selecting combinations of input conditions in a systematic way, such that the number of test cases does not become unmanageably large.
- The following process is used to derive the test cases:

1. Division of specification: The specification is divided into workable pieces, as cause-effect graphing becomes complex when used on large specifications.
2. Identification of causes and effects: The next step is to identify causes and effects in the specifi cations. A cause is a distinct input condition identified in the problem. I Similarly, an effect is an output condition.

CAUSE-EFFECT GRAPHING BASED TESTING

3. Transformation of specification into a cause-effect graph Based on the analysis of the specification, it is transformed into a Boolean graph linking the causes and effects. This is the cause-effect graph. Complete the graph by adding the constraints, if any, between causes and effects.

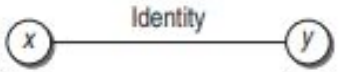
4. Conversion into decision table The cause-effect graph obtained is converted into a limited-entry decision table by verifying state conditions in the graph. Each column in the table represents a test case.

5. Deriving test cases The columns in the decision table are converted into test cases.

Basic Notations for Cause-Effect Graph

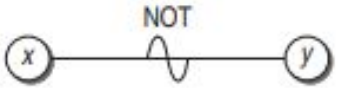
Identity

According to the identity function, if x is 1, y is 1; else y is 0.



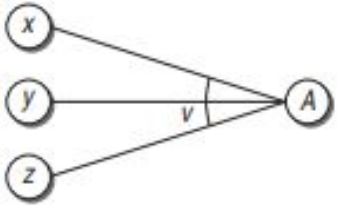
NOT

This function states that if x is 1, y is 0; else y is 1.



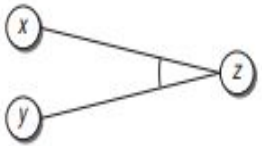
OR

The OR function states that if x or y or z is 1, A is 1; else A is 0.



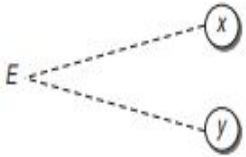
AND

This function states that if both x and y are 1, z is 1; else z is 0.



Exclusive

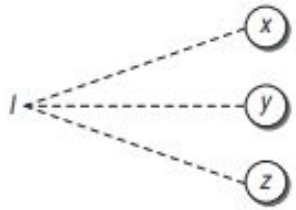
Sometimes, the specification contains an impossible combination of causes such that two causes cannot be set to 1 simultaneously. For this, Exclusive function is used. According to this function, it always holds that either x or y can be 1, i.e. x and y cannot be 1 simultaneously.



Basic Notations for Cause-Effect Graph

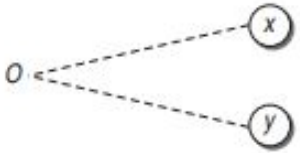
Inclusive

It states that at least one of x, y, and z must always be 1 (x, y, and z cannot be 0 simultaneously).



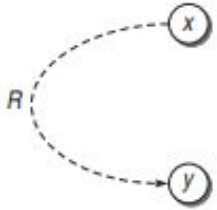
One and Only One

It states that one and only one of x and y must be 1.



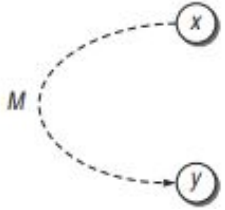
Requires

It states that for x to be 1, y must be 1, i.e. it is impossible for x to be 1 and y to be 0.



Mask

It states that if x is 1, y is forced to 0.



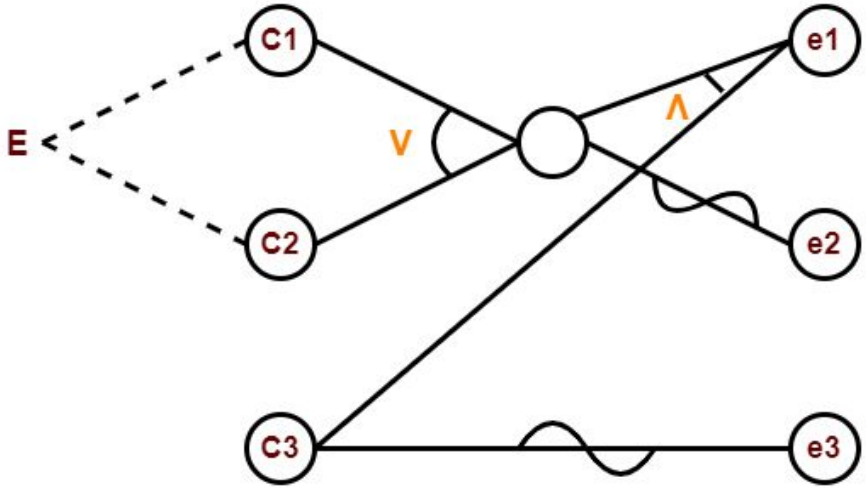
Example: If the character of the first column is ‘A’ or ‘B’ and the second column is a number, then the file is considered updated. If the first character is incorrect then message x should be printed. If the second column is not a number, then message y should be printed.

Solution: The causes represented by letter “C” are as follows-

- C1 : The character in column 1 is ‘A’
- C2 : The character in column 1 is ‘B’
- C3 : The character in column 2 is a number

The effects represented by letter “e” are as follows-

- e1 : File update is made
- e2 : Message x is printed
- e3 : Message y is printed



Test data	Causes			Effect		
	A1	A2	A3	M1	M2	M3
1	0	0	0	0	1	1
2	0	0	1	0	1	0
3	0	1	0	0	0	1
4	0	1	1	1	0	0
5	1	0	0	0	0	1
6	1	0	1	1	0	0

ERROR GUESSING

- Error guessing is the preferred method used when all other methods fail. Sometimes it is used to test some special cases.
- According to this method, errors or bugs can be guessed which do not fit in any of the earlier defined situations. So test cases are generated for these special cases.
- It is a very practical case wherein the tester uses his intuition and makes a guess about where the bug can be.
- The tester does not have to use any particular testing technique.
- However, this capability comes with years of experience in a particular field of testing
- Every testing activity is been always recorded.

ERROR GUESSING

- The history of bugs can help in identifying some special cases in the project.
- There is a high probability that errors made in a previous project is repeated again.
- In these situations, error guessing is an ad hoc approach, based on intuition, experience, knowledge of project, and bug history. Any of these can help to expose the errors.
- The basic idea is to make a list of possible errors in error-prone situations and then develop the test cases.
- Thus, there is no general procedure for this technique, as it is largely an intuitive and ad hoc process.

WHITE BOX TESTING

- White-box testing is another effective testing technique in dynamic testing. It is also known as glass-box testing, as everything that is required to implement the software is visible.
- The entire design, structure, and code of the software have to be studied for this type of testing.
- It is obvious that the developer is very close to this type of testing.
- Often, developers use white-box testing techniques to test their own design and code. This testing is also known as structural or development testing.
- It ensures that the internal parts of the software are adequately tested.
- It is applied in Unit Testing, Integration & System Testing.

NEED OF WHITE-BOX TESTING

1. 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.

2. Since white-box testing is complementary to black-box testing, there are categories of bugs which can be revealed by white-box testing, but not through black-box testing.

NEED OF WHITE-BOX TESTING

3. 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.

4. 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.

5. 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

Statement Coverage: The first kind of logic coverage can be identified in the form of statements. It is assumed that if all the statements of the module are executed once, every bug will be notified. If we want to cover every statement in the above code, then the following test cases must be designed:

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 1 just skips the while loop and all loop statements are not executed. Considering test case 2, the loop is also executed. However, every statement inside the loop is not executed. So two more cases are designed:

Test case 3: x > y

Test case 4: x < y

We can see that test case 3 and 4 are sufficient to execute all the statements in the code.

But, if we execute only test case 3 and 4, then conditions and paths in test case 1 will never be tested and errors will go undetected. Thus, statement coverage is a necessary but not a sufficient criteria for logic 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);
```

Figure 5.1 Sample code

LOGIC COVERAGE CRITERIA

Decision or Branch Coverage: Branch coverage states that each decision takes on all possible outcomes (True or False) at least once.

In other words, each branch direction must be traversed at least once.

In the previous sample code shown in Figure 5.1, while and if statements have two outcomes: True and False. So test cases must be designed such that both outcomes for while and if statements are tested. The test cases are designed as:

Test case 1: x = y

Test case 2: x != y

Test case 3: x < y

Test case 4: x > y

LOGIC COVERAGE CRITERIA

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: $I \leq 5, J < \text{COUNT}$
Test case 2: $I < 5, J > \text{COUNT}$

LOGIC COVERAGE CRITERIA

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.

Therefore, multiple condition coverage requires that we should write sufficient test cases such that all possible combinations of condition outcomes in each decision and all points of entry are invoked at least once.

BASIS PATH TESTING

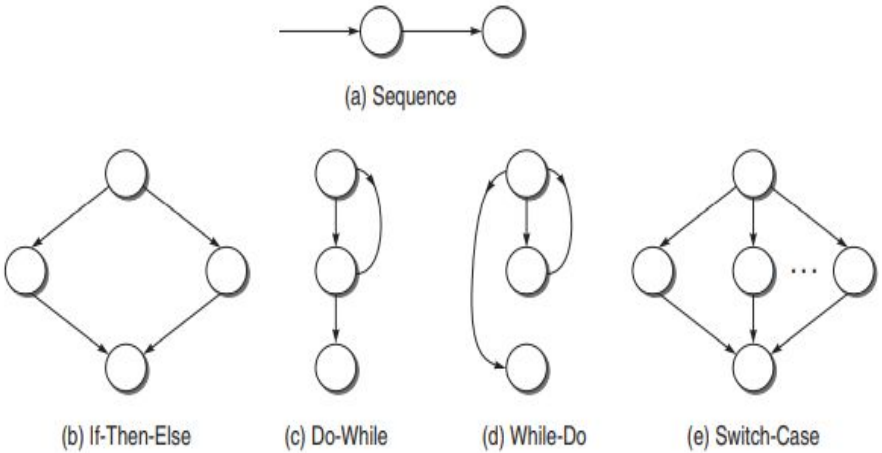
- Basis path testing is the oldest structural testing technique. The technique is based on the control structure of the program.
- Based on the control structure, a flow graph is prepared and all the possible paths can be covered and executed during testing.
- Path coverage is a more general criterion as compared to other coverage criteria and useful for detecting more errors.
- But the problem with path criteria is that programs that contain loops may have an infinite number of possible paths and it is not practical to test all the paths.
- Some criteria should be devised such that selected paths are executed for maximum coverage of logic.

CONTROL FLOW GRAPH

The control flow graph is a graphical representation of control structure of a program. Flow graphs can be prepared as a directed graph. A directed graph (V, E) consists of a set of vertices V and a set of edges E that are ordered pairs of elements of V. Based on the concepts of directed graph, following notations are used for a flow graph:

- Node It represents one or more procedural statements. The nodes are denoted by a circle. These are numbered or labeled.
- Edges or links They represent the flow of control in a program. This is denoted by an arrow on the edge. An edge must terminate at a node.
- Decision node A node with more than one arrow leaving it is called a decision node.
- Junction node A node with more than one arrow entering it is called a junction.
- Regions Areas bounded by edges and nodes are called regions. When counting the regions, the area outside the graph is also considered a region.

FLOW GRAPH NOTATIONS FOR DIFFERENT PROGRAMMING CONSTRUCTS



Cyclomatic complexity number can be derived through any of the following three formulae

1. $V(G) = e - n + 2p$
   where e is number of edges, n is the number of nodes in the graph, and p is number of components in the whole graph.
2. $V(G) = d + p$
   where d is the number of decision nodes in the graph.
3. $V(G)$ = number of regions in the graph

5.3.3 Path Testing Terminology

Path A path through a program is a sequence of instructions or statements that starts at an entry, junction, or decision and ends at another, or possibly the same, junction, decision, or exit. A path may go through several junctions, processes, or decisions, one or more times.

Segment Paths consist of segments. The smallest segment is a link, that is, a single process that lies between two nodes (e.g., junction-process-junction, junction-process-decision, decision-process-junction, decision-process-decision). A direct connection between two nodes, as in an unconditional GOTO, is also called a process by convention, even though no actual processing takes place.

Path segment A path segment is a succession of consecutive links that belongs to some path.

Length of a path The length of a path is measured by the number of links in it and not by the number of instructions or statements executed along the path. An alternative way to measure the length of a path is by the number of nodes traversed. This method has some analytical and theoretical benefits. If programs are assumed to have an entry and an exit node, then the number of links traversed is just one less than the number of nodes traversed.

Independent path An independent path is any path through the graph that introduces at least one new set of processing statements or new conditions. An independent path must move along at least one edge that has not been traversed before the path is defined [9,28].

Consider the following program segment:

```
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
```

(a) Draw the DD graph for the program.
(b) Calculate the cyclomatic complexity of the program using all the methods.
(c) List all independent paths.
(d) Design test cases from independent paths.

(a) DD graph

For a DD graph, the following actions must be done:

- Put the line numbers on the execution statements of the progra as shown in Fig. 5.4. Start numbering the statements after decl ing the variables, if no variables have been initialized. Otherwi start from the statement where a variable has been initialized.

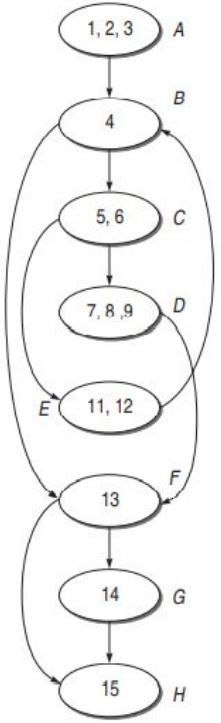


Figure 5.4 DD graph for Example 5.1

(b) Cyclomatic complexity

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

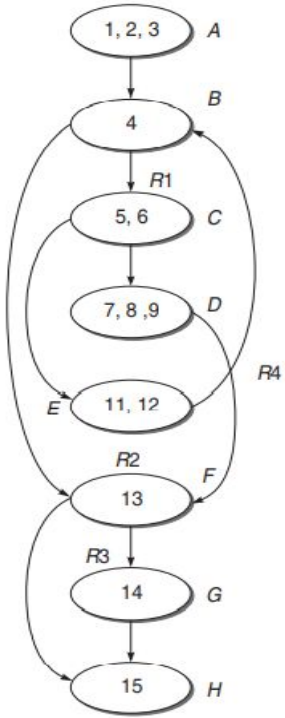


Figure 5.5 DD graph for Example 5.1 showing regions

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

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

(c) Independent paths

Since the cyclomatic complexity of the graph is 4, there will be 4 independent paths in the graph as shown below:

(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

(d) Test case design from the list of independent paths

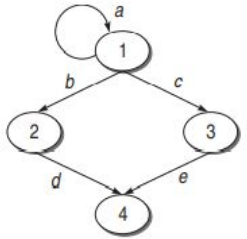
Test case ID	Input num	Expected result	Independent paths covered by test case
1	1	No output is displayed	A-B-F-H
2	2	Prime number	A-B-F-G-H
3	4	Not a prime number	A-B-C-D-F-H
4	3	Prime number	A-B-C-E-B-F-G-H

GRAPH MATRICES

- Flow graph is an effective aid in path testing as seen in the previous section.
- However, path tracing with the use of flow graphs may be a cumbersome and time-consuming activity.
- Moreover, as the size of graph increases, manual path tracing becomes difficult and leads to errors.
- A link can be missed or covered twice. So the idea is to develop a software tool which will help in basis path testing.
- Graph matrix, a data structure, is the solution which can assist in developing a tool for automation of path tracing.
- The reason being the properties of graph matrices are fundamental to test tool building.
- Moreover, testing theory can be explained on the basis of graphs. Graph theorems can be proved easily with the help of graph matrices.
- So graph matrices are very useful for understanding the testing theory.

GRAPH MATRIX

- A graph matrix is a square matrix whose rows and columns are equal to the number of nodes in the fl ow graph.
- Each row and column identifies a particular node and matrix entries represent a connection between the nodes.
- The following points describe a graph matrix:
- „ Each cell in the matrix can be a direct connection or link between one node to another node.
- „ If there is a connection from node ‘a’ to node ‘b’, then it does not mean that there is connection from node ‘b’ to node ‘a’.
- „ Conventionally, to represent a graph matrix, digits are used for nodes and letter symbols for edges or connections



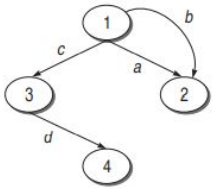
Consider the above graph and represent it in the form of a graph matrix.

Solution

The graph matrix is shown below.

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

Consider the graph and represent it in the form of a graph matrix.



Solution

The graph matrix is shown below.

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

CONNECTION MATRIX

- Till now, we have learnt how to represent a flow graph into a matrix representation.
- But this matrix is just a tabular representation and does not provide any useful information.
- If we add link weights to each cell entry, then graph matrix can be used as a powerful tool in testing.
- The links between two nodes are assigned a link weight which becomes the entry in the cell of matrix. The link weight provides information about control flow.
- In the simplest form, when the connection exists, then the link weight is 1, otherwise 0 (But 0 is not entered in the cell entry of matrix to reduce the complexity).
- A matrix defined with link weights is called a connection matrix. The connection matrix for Example 5.6 is shown below.

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

The connection matrix for Example 5.7 is shown below.

	1	2	3	4
1		1	1	
2				
3				1
4				

5.4.3 Use of Connection Matrix in Finding Cyclomatic Complexity Number

Connection matrix is used to see the control flow of the program. Further, it is used to find the cyclomatic complexity number of the flow graph. Given below is the procedure to find the cyclomatic number from the connection matrix:

Step 1: For each row, count the number of 1s and write it in front of that row.

Step 2: Subtract 1 from that count. Ignore the blank rows, if any.

Step 3: Add the final count of each row.

Step 4: Add 1 to the sum calculated in Step 3.

Step 5: The final sum in Step 4 is the cyclomatic number of the graph.

The cyclomatic number calculated from the connection matrix of Example 5.6 is shown below.

	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				

The cyclomatic number calculated from the connection matrix of Example 5.7 is shown below.

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

LOOP TESTING

- Loop testing can be viewed as an extension to branch coverage. Loops are important in the software from the testing viewpoint.
- If loops are not tested properly, bugs can go undetected. This is the reason that loops are covered in this section exclusively.
- Loop testing can be done effectively while performing development testing (unit testing by the developer) on a module.
- Sufficient test cases should be designed to test every loop thoroughly.
- There are four different kinds of loops. How each kind of loop is tested, is discussed ahead:

Simple loops Simple loops mean, we have a single loop in the flow, as shown in Fig. 5.9.

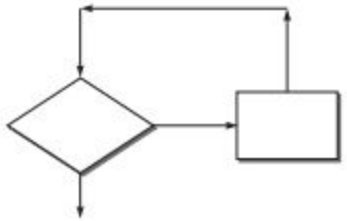


Figure 5.9 (a)

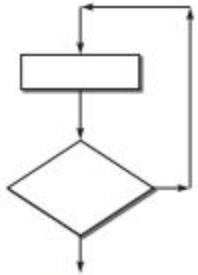


Figure 5.9 (b)

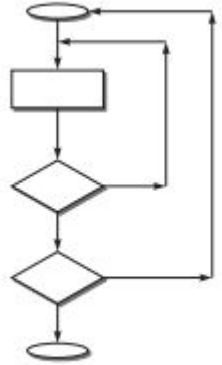


Figure 5.10 Nested loops

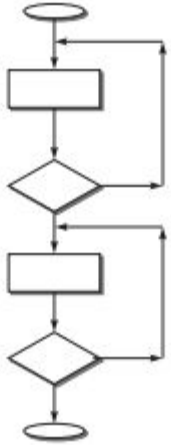


Figure 5.11 Concatenated loops

LOOP TESTING

The following test cases should be considered for simple loops while testing them:

- 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.
- 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 loops:

- When two or more loops are embedded, it is called a nested loop, as shown in Fig.
- If we have nested loops in the program, it becomes difficult to test. If we adopt the approach of simple tests to test the nested loops, then the number of possible test cases grows geometrically.
- Thus, the strategy is to 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

LOOP TESTING

Concatenated loops:

- The loops in a program may be concatenated (Fig. 5.11). Two loops are concatenated if it is possible to reach one after exiting the other, while still on a path from entry to exit.
- If the two loops are not on the same path, then they are not concatenated.
- The two loops on the same path may or may not be independent.

Unstructured loops: This type of loops is really impractical to test and they must be redesigned or at least converted into simple or concatenated loops.

DATA FLOW TESTING

- In path coverage, the stress was to cover a path using statement or branch coverage.
- However, data and data integrity is as important as code and code integrity of a module.
- We have checked every possibility of the control flow of a module. But what about the data flow in the module?
- Data flow testing is a white-box testing technique that can be used to detect improper use of data values due to coding errors.
- It gives a chance to look out for inappropriate data definition, its use in predicates, computations, and termination. It identifies potential bugs by examining the patterns in which that piece of data is used.

DATA FLOW TESTING

- To examine the patterns, the control flow graph of a program is used. This test strategy selects the paths in the module's control flow such that various sequences of data objects can be chosen.
- The major focus is on the points at which the data receives values and the places at which the data initialized has been referenced.
- Thus, we have to choose enough paths in the control flow to ensure that every data is initialized before use and all the defined data have been used somewhere.
- Data flow testing closely examines the state of the data in the control flow graph, resulting in a richer test suite.

DATA FLOW TESTING

STATE OF A DATA OBJECT A data object can be in the following states: „

- **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.
- **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 TESTING

DATA-FLOW ANOMALIES:

- Data-flow anomalies represent the patterns of data usage which may lead to an incorrect execution of the code.
- An anomaly is denoted by a two-character sequence of actions.
- For example, ‘dk’ means a variable is defi ned and killed without any use, which is a potential bug.
- There are nine possible two-character combinations out of which only four are data anomalies, as shown in Table 5.1.

Table 5.1 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.**

DATA FLOW TESTING

- It can be observed that not all data-flow anomalies are harmful, but most of them are suspicious and indicate that an error can occur.
- In addition to the above two-character data anomalies, there may be single-character data anomalies also.
- To represent these types of anomalies, we take the following conventions:
- ~x : indicates all prior actions are not of interest to x.
- x~ : indicates all post actions are not of interest to x.

Table 5.2 Single-character data-flow anomalies

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.

Definition node Defining a variable means assigning value to a variable for the very first time in a 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:

(i) Predicate Usage Node: If usage node n is a predicate node, then n is a predicate usage node.

(ii) Computation Usage Node: If usage node n corresponds to a computation statement in a program other than predicate, then it is called a computation usage node.

Loop-free path segment It is a path segment for which every node is visited once at most.

Simple path segment It is a 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 either be a p-usage or a c-usage node.

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

Consider the program given below for calculating the gross salary of an employee in an organization. If his basic salary is less than Rs 1500, then HRA = 10% of basic salary and DA = 90% of the basic. If his salary is either equal to or above Rs 1500, then HRA = Rs 500 and DA = 98% of the basic salary. Calculate his gross salary.

```
    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. }
```

Find out the define-use-kill patterns for all the variables in the source code of this application.

Solution

For variable 'bs', the define-use-kill patterns are given below.

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

For variable 'gs', the define-use-kill patterns are given below.

Pattern	Line Number	Explanation
~d	14	Normal case. Allowed
du	14-15	Normal case. Allowed
uk	15-16	Normal case. Allowed
K~	16	Normal case. Allowed

For variable 'da', the define-use-kill patterns are given below.

Pattern	Line Number	Explanation
~d	7	Normal case. Allowed
du	7-14	Normal case. Allowed
uk	14-16	Normal case. Allowed
K~	16	Normal case. Allowed

For variable 'hra', the define-use-kill patterns are given below.

Pattern	Line Number	Explanation
~d	1	Normal case. Allowed
dd	1-6 or 1-11	Double definition. Not allowed. Harmless bug.
du	6-14 or 11-14	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.
- During mutation testing, faults are introduced into a program by creating many versions of the program, each of which contains one fault.
- Test data are used to execute these faulty programs with the goal of causing each faulty program to fail. Faulty programs are called mutants of the original program and a mutant is said to be killed when a test case causes it to fail.
- When this happens, the mutant is considered dead and no longer needs to remain in the testing process, since the faults represented by that mutant have been detected, and more importantly, it has satisfied its requirement of identifying a useful test case.
- Thus, the main objective is to select efficient test data which have error-detection power. The criterion for this test data is to differentiate the initial program from the mutant.
- This distinguish-ability between the initial program and its mutant will be based on test results.

PRIMARY MUTANTS

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

We can consider the following mutants for the above example:

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

When the mutants are single modifications of the initial program using some operators as shown above, they are called *primary mutants*. Mutation operators are dependent on programming languages. Note that each of the mutated statements represents a separate program. The results of the initial program and its mutants are shown below.

Test Data	x	y	Initial Program Result	Mutant Result
TD1	2	2	4	0 (M1)
TD2(x and y # 0)	4	3	7	1.4 (M2)
TD3 (y #1)	3	2	5	4 (M3)
TD4(y #0)	5	2	7	2 (M4)

SECONDARY MUTANTS

This class of mutants is called secondary mutants when multiple levels of mutation are applied on the initial program. In this case, it is very diffi cult to identify the initial program from its mutants.

Let us take another example program as shown below:

```
if (a < b)
    c = a;
```

Now, mutants for this code may be as follows:

M1 :
```
if (a <= b-1)
        c = a;
```

M2:
```
if (a+1 <= b)
        c = a;
```

M3:
```
if (a == b)
        c = a+1;
```

MUTATION TESTING PROCESS

The mutation testing process is discussed below:

- Construct the mutants of a test program.
- Add test cases to the mutation system and check the output of the program on each test case to see if it is correct.
- If the output is incorrect, a fault has been found and the program must be modified and the process restarted.
- If the output is correct, that test case is executed against each live mutant.
- If the output of a mutant differs from that of the original program on the same test case, the mutant is assumed to be incorrect and is killed.
- After each test case has been executed against each live mutant, each remaining mutant falls into one of the following two categories. Σ One, the mutant is functionally equivalent to the original program. An equivalent mutant always produces the same output as the original program, so no test case can kill it. Two, the mutant is killable, but the set of test cases is insufficient to kill it. In this case, new test cases need to be created, and the process iterates until the test set is strong enough to satisfy the tester.
- The mutation score for a set of test data is the percentage of non-equivalent mutants killed by that data. If the mutation score is 100%, then the test data is called mutation-adequate.

STATIC TESTING

- Static testing techniques do not execute the software and they do not require the bulk of test cases.
- This type of testing is also known as non-computer based testing or human testing.
- All the bugs cannot be caught alone by the dynamic testing technique; static testing reveals the errors which are not shown by dynamic testing.
- Static testing techniques do not demonstrate that the software is operational or that one function of software is working; rather they check the software product at each SDLC stage for conformance with the required specifications or standards.
- The objectives of static testing can be summarized as follows:
- „ To identify errors in any phase of SDLC as early as possible
- „ To verify that the components of software are in conformance with its requirements
- „ To provide information for project monitoring
- „ To improve the software quality and increase productivity
- Static testing can be categorized into the following types: **„ Software inspections „ Walkthroughs „ Technical reviews**

INSPECTIONS

- Software inspections were first introduced at IBM by Fagan in the early 1970s.
- These can be used to tackle software quality problems because they allow the detection and removal of defects after each phase of the software development process.
- Inspection process is an in-process manual examination of an item to detect bug.
- This process does not require executable code or test cases.
- With inspection, bugs can be found on infrequently executed paths that are not likely to be included in test cases. Software inspection does not execute the code, so it is machine-independent, requires no target system resources or changes to the program's operational behaviour, and can be used much before the target hardware is available for dynamic testing purposes.
- The inspection process is carried out by a group of peers.
- The group of peers first inspect the product at the individual level. After this, they discuss the potential defects of the product observed in a formal meeting.
- The second important thing about the inspection process is that it is a formal process of verifying a software product. The documents which can be inspected are SRS, SDD, code, and test plan.

INSPECTIONS

INSPECTION TEAM: For the inspection process, a minimum of the following four team members are required.

- Author/Owner/Producer A programmer or designer responsible for producing the program or document. He is also responsible for fixing defects discovered during the inspection process.
- Inspector A peer member of the team, i.e. he is not a manager or supervisor. He is not directly related to the product under inspection and may be concerned with some other product. He finds errors, omissions, and inconsistencies in programs and documents.
- Moderator A team member who manages the whole inspection process. He schedules, leads, and controls the inspection session. He is the key person with the responsibility of planning and successful execution of the inspection.
- Recorder One who records all the results of the inspection meeting.

INSPECTION PROCESS

Planning: During this phase, the following is executed:

- The product to be inspected is identified.
- A moderator is assigned.
- The objective of the inspection is stated, i.e. whether the inspection is to be conducted for defect detection or something else. If the objective is defect detection, then the type of defect detection like design error, interface error, code error must be specifiEd. The aim is to define an objective for the meeting so that the effort spent in inspections is properly utilized.

Overview:

- In this stage, the inspection team is provided with the background information for inspection. The author presents the rationale for the product, its relationship to the rest of the products being developed, its function and intended use, and the approach used to develop it.
- This information is necessary for the inspection team to perform a successful inspection.

INSPECTION PROCESS

Individual preparation:

- After the overview, the reviewers individually prepare themselves for the inspection process by studying the documents provided to them in the overview session.
- They point out potential errors or problems found and record them in a log. This log is then submitted to the moderator. The moderator compiles the logs of different members and gives a copy of this compiled list to the author of the inspected item

Inspection meeting:

Once all the initial preparation is complete, the actual inspection meeting can start. The inspection meeting starts with the author of the inspected item who has created it. The author first discusses every issue raised by different members in the compiled log file. After the discussion, all the members arrive at a consensus whether the issues pointed out are in fact errors and if they are errors, should they be admitted by the author. It may be possible that during the discussion on any issue, another error is found. Then, this new error is also discussed and recorded as an error by the author. The basic goal of the inspection meeting is to uncover any bug in the item. However, no effort is made in the meeting to fix the bug

INSPECTION PROCESS

Rework: The summary list of the bugs that arise during the inspection meeting needs to be reworked by the author. The author fixes all these bugs and reports back to the moderator.

Follow-up:

- It is the responsibility of the moderator to check that all the bugs found in the last meeting have been addressed and fixed.
- He prepares a report and ascertains that all issues have been resolved. The document is then approved for release.
- If this is not the case, then the unresolved issues are mentioned in a report and another inspection meeting is called by the moderator

STRUCTURED WALKTHROUGHS

The idea of structured walkthroughs was proposed by Yourdon [106]. It is a less formal and less rigorous technique as compared to inspection. The common term used for static testing is inspection but it is a very formal process. If you want to go for a less formal process having no bars of organized meeting, then walkthroughs are a good option. A typical structured walkthrough team consists of the following members:

- Coordinator Organizes, moderates, and follows up the walkthrough activities.
- Presenter/Developer Introduces the item to be inspected. This member is optional.
- Scribe/Recorder Notes down the defects found and suggestion proposed by the members.
- Reviewer/Tester Finds the defects in the item.
- Maintenance Oracle Focuses on long-term implications and future maintenance of the project.
- Standards Bearer Assesses adherence to standards.
- User Representative/Accreditation Agent Reflects the needs and concerns of the user.

STRUCTURED WALKTHROUGHS

- A walkthrough is less formal, has fewer steps and does not use a checklist to guide or a written report to document the team's work.
- Rather than simply reading the program or using error checklists, the participants 'play computer'.
- The person designated as a tester comes to the meeting armed with a small set of paper test cases—representative sets of inputs and expected outputs for the program or module.
- During the meeting, each test case is mentally executed. That is, the test data are walked through the logic of the program.
- The state of the program is monitored on a paper or any other presentation media. The walkthrough should have a follow-up process similar to that described in the inspection process.

TECHNICAL REVIEWS

- A technical review is intended to evaluate the software in the light of development standards, guidelines, and specifications and to provide the management with evidence that the development process is being carried out according to the stated objectives.
- A review is similar to an inspection or walkthrough, except that the review team also includes management. Therefore, it is considered a higher-level technique as compared to inspection or walkthrough.
- A technical review team is generally comprised of management-level representatives and project management. Review agendas should focus less on technical issues and more on oversight than an inspection. The purpose is to evaluate the system relative to specifications and standards, recording defects and deficiencies.
- The moderator should gather and distribute the documentation to all team members for examination before the review. He should also prepare a set of indicators to measure the following points:
- „ Appropriateness of the problem definition and requirements
- „ Adequacy of all underlying assumptions
- „ Adherence to standards
- „ Consistency
- „ Completeness
- „ Documentation

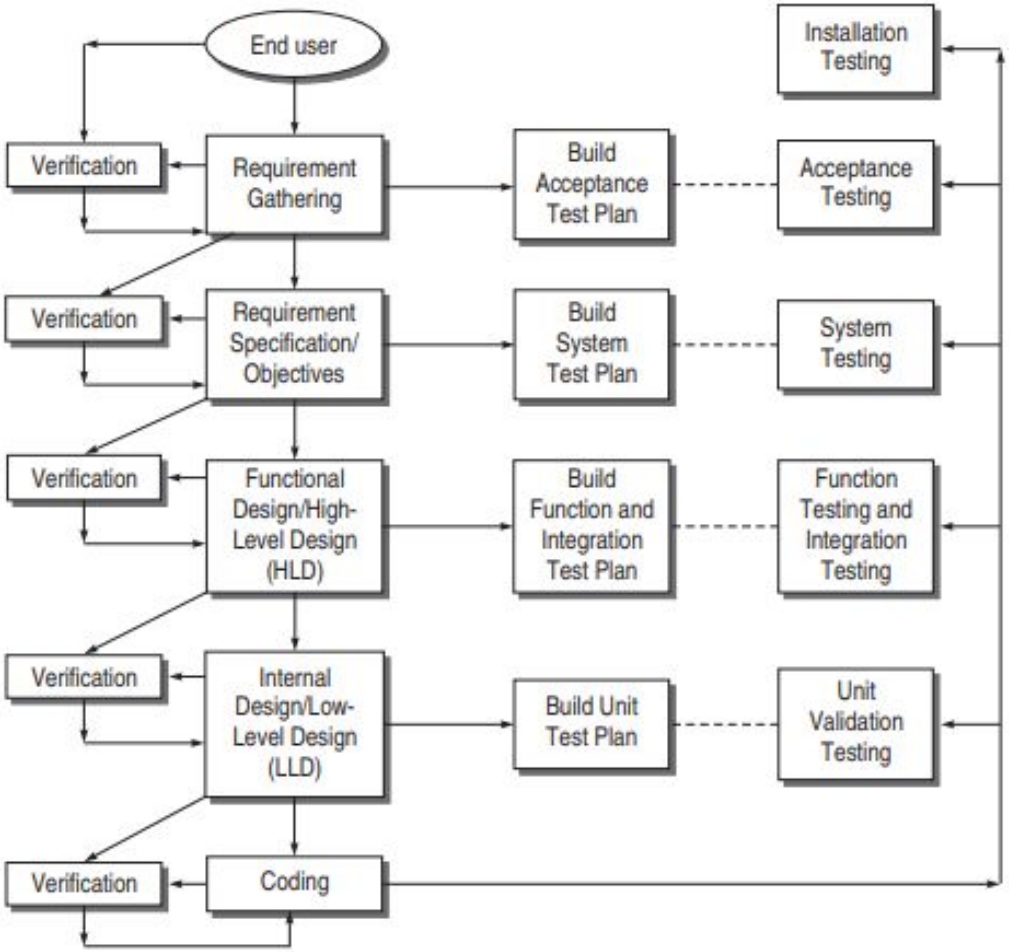


Figure 7.1 V&V activities

UNIT VALIDATION TESTING

- Since unit is the smallest building block of the software system, it is the first piece of system to be validated.
- Before we validate the entire software, units or modules must be validated. Unit testing is normally considered an adjunct to the coding step.
- Units must also be validated to ensure that every unit of software has been built in the right manner in conformance with user requirements.
- Unit tests ensure that the software meets at least a baseline level of functionality prior to integration and system testing.
- Though software is divided into modules but a module is not an isolated entity.
- The module under consideration might be getting some inputs from another module or the module is calling some other module. It means that a module is not independent and cannot be tested in isolation.
- While testing the module, all its interfaces must be simulated if the interfaced modules are not ready at the time of testing the module under consideration.

UNIT VALIDATION TESTING

- **Drivers:** Suppose a module is to be tested, wherein some inputs are to be received from another module.
- However, this module which passes inputs to the module to be tested is not ready and under development.
- In such a situation, we need to simulate the inputs required in the module to be tested.
- For this purpose, a main program is prepared, wherein the required inputs are either hard-coded or entered by the user and passed on to the module under test.
- This module where the required inputs for the module under test are simulated for the purpose of module or unit testing is known as a driver module.

UNIT VALIDATION TESTING

- A driver can be defined as a software module which is used to invoke a module under test and provide test inputs, control and monitor execution, and report test results or most simplistically a line of code that calls a method and passes a value to that method.
- A test driver provides the following facilities to a unit to be tested:

„ Initializes the environment desired for testing.

„ Provides simulated inputs in the required format to the units to be tested.

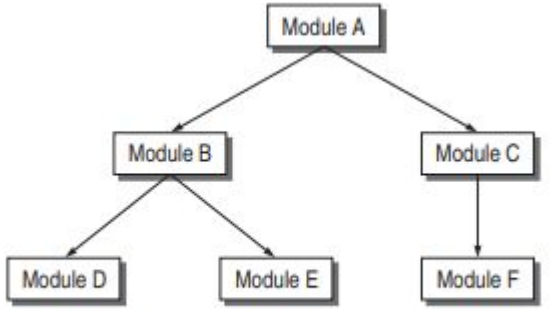


Figure 7.2 Design hierarchy of an example system

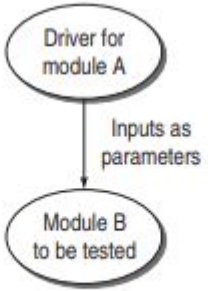


Figure 7.3 Driver Module for module A

UNIT VALIDATION TESTING

- **Stubs:** The module under testing may also call some other module which is not ready at the time of testing.
- Therefore, these modules need to be simulated for testing.
- In most cases, dummy modules instead of actual modules, which are not ready, are prepared for these subordinate modules.
- These dummy modules are called stubs.
- Thus, a stub can be defined as a piece of software that works similar to a unit which is referenced by the unit being tested, but it is much simpler than the actual unit.
- A stub works as a ‘stand-in’ for the subordinate unit and provides the minimum required behaviour for that unit.

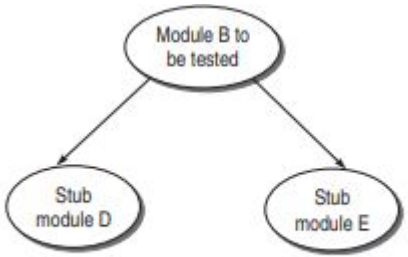


Figure 7.4 Stubs

UNIT VALIDATION TESTING

The benefits of designing stubs and drivers are:

- Stubs allow the programmer to call a method in the code being developed, even if the method does not have the desired behaviour yet.
- By using stubs and drivers effectively, we can cut down our total debugging and testing time by testing small parts of a program individually, helping us to narrow down problems before they expand.

However, drivers and stubs represent overheads also. Overhead of designing them may increase the time and cost of the entire software system. Therefore, they must be designed simple to keep overheads low. **Stubs and drivers are generally prepared by the developer of the module under testing.**

INTEGRATION TESTING

- In the modular design of a software system where the system is composed of different modules, integration is the activity of combining the modules together when all the modules have been prepared.
- Integration of modules is according to the design of software specified earlier.
- Integration aims at constructing a working software system.
- Thus, integration testing is necessary for the following reasons:

„ Integration testing exposes inconsistency between the modules such as improper call or return sequences.

„ Data can be lost across an interface.

„ One module when combined with another module may not give the desired result.

„ Data types and their valid ranges may mismatch between the modules.

DECOMPOSITION-BASED INTEGRATION

- The idea for this type of integration is based on the decomposition of design into functional components or modules.
- The functional decomposition is shown as a tree in Fig. 7.7. In the tree designed for decomposition-based integration, the nodes represent the modules present in the system and the links/edges between the two modules represent the calling sequence.
- The nodes on the last level in the tree are leaf nodes. In the tree structure shown in Fig. 7.7, module A is linked to three subordinate modules, B, C, and D. It means that module A calls modules, B, C, and D.
- All integration testing methods in the decomposition-based integration assume that all the modules have been unit tested in isolation.
- Thus, with the decomposition-based integration, we want to test the interfaces among separately tested modules.

DECOMPOSITION-BASED INTEGRATION

- Integration methods in decomposition-based integration depend on the methods on which the activity of integration is based.
- One method of integrating is to integrate all the modules together and then test it.
- Another method is to integrate the modules one by one and test them incrementally.
- Based on these methods, integration testing methods are classified into two categories:
- (a) non-incremental and
- (b) incremental.

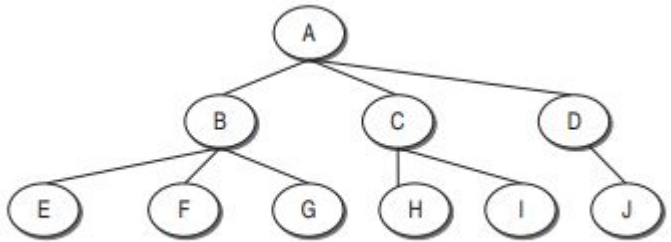


Figure 7.7 Decomposition Tree

Non-Incremental Integration Testing

- In this type of testing, either all untested modules are combined together and then tested or unit tested modules are combined together.
- It is also known as Big-Bang integration testing. Big-Bang method cannot be adopted practically. This theory has been discarded due to the following reasons:

1. Big-Bang requires more work.

2. Actual modules are not interfaced directly until the end of the software system.

3. It will be difficult to localize the errors since the exact location of bugs cannot be found easily.

Incremental Integration Testing

- In this type, you start with one module and unit test it. Then combine the module which has to be merged with it and perform test on both the modules.
- In this way, incrementally keep on adding the modules and test the recent environment.
- Thus, an integrated tested software system is achieved. Incremental integration testing is beneficial for the following reasons:

  1. Incremental approach does not require many drivers and stubs.

  2. Interfacing errors are uncovered earlier.

  3. It is easy to localize the errors since modules are combined one by one. The first suspect is the recently added module. Thus, debugging becomes easy.

  4. Incremental testing is a more thorough testing.

Types of Incremental Integration Testing

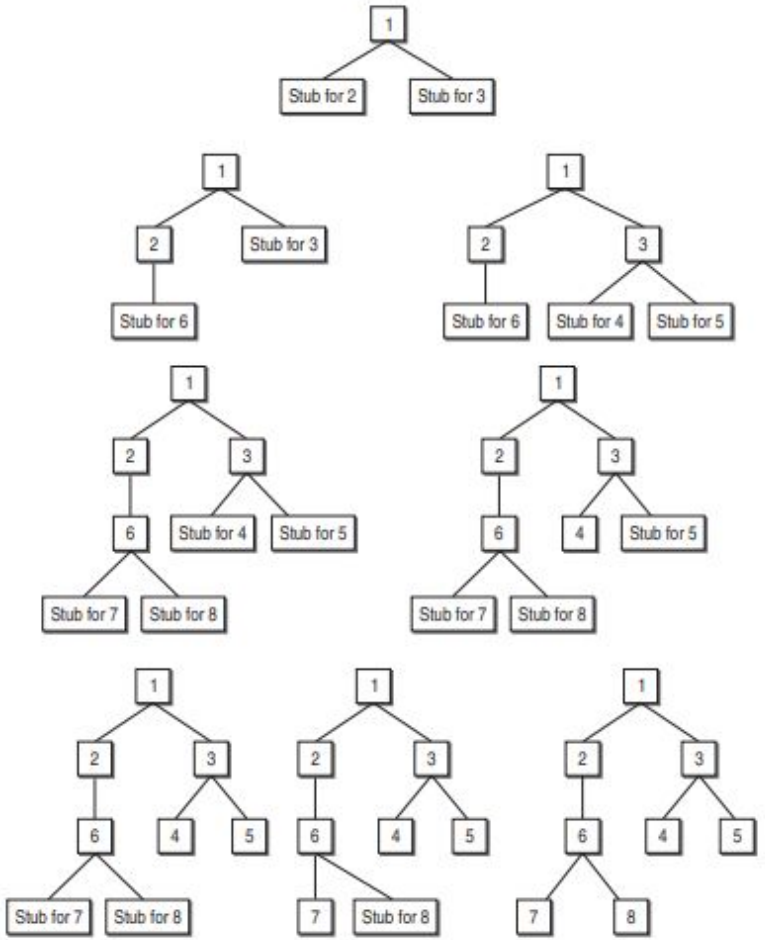
Design hierarchy of a software can be seen in a tree-like structure, as shown in Fig. 7.8. In this tree-like structure, incremental integration can be done either from top to bottom or bottom to top. Based on this strategy, incremental integration testing is divided into two categories.

1. Top-down Integration Testing
2. Bottom-up Integration Testing

Top-down Integration Testing

- The strategy in top-down integration is to look at the design hierarchy from top to bottom.
- Start with the high-level modules and move downward through the design hierarchy.
- Modules subordinate to the top module are integrated in the following two ways:
- **Depth first integration:** In this type, all modules on a major control path of the design hierarchy are integrated first. In the example shown in Fig. 7.8, modules 1, 2, 6, 7/8 will be integrated first. Next, modules 1, 3, 4/5 will be integrated.
- **Breadth first integration:** In this type, all modules directly subordinate at each level, moving across the design hierarchy horizontally, are integrated first. In the example shown in Fig. 7.8, modules 2 and 3 will be integrated first. Next, modules 6, 4, and 5 will be integrated. Modules 7 and 8 will be integrated last.

The top-down integration for Fig. 7.8 is shown below.



Bottom-up Integration Testing

- This strategy begins with the terminal or modules at the lowest level in the software structure.
- After testing these modules, they are integrated and tested moving from bottom to top level.
- Since the processing required for modules subordinate to a given level is always available, stubs are not required in this strategy.
- Bottom-up integration can be considered as the opposite of top-down approach. Unlike top-down strategy, this strategy does not require the architectural design of the system to be complete.
- Thus, bottom-up integration can be performed at an early stage in the developmental process. It may be used where the system reuses and modifies components from other systems.

Bottom-up Integration Testing

The steps in bottom-up integration are as follows:

1. Start with the lowest level modules in the design hierarchy. These are the modules from which no other module is being called.

2. Look for the super-ordinate module which calls the module selected in step 1. Design the driver module for this super-ordinate module.

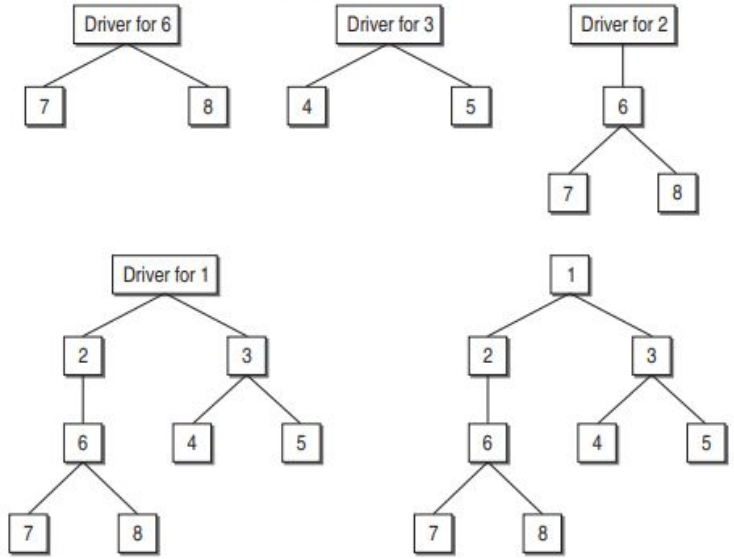
3. Test the module selected in step 1 with the driver designed in step 2.

4. The next module to be tested is any module whose subordinate modules (the modules it calls) have all been tested.

5. Repeat steps 2 to 5 and move up in the design hierarchy.

6. Whenever, the actual modules are available, replace stubs and drivers with the actual one and test again.

Bottom-up integration for Fig. 7.8, is shown below:



CALL GRAPH-BASED INTEGRATION

- If we can refine the functional decomposition tree into a form of module calling graph, then we are moving towards behavioural testing at the integration level.
- This can be done with the help of a call graph, as given by Jorgensen.
- A call graph is a directed graph, wherein the nodes are either modules or units, and a directed edge from one node to another means one module has called another module.
- The call graph can be captured in a matrix form which is known as the adjacency matrix. For example, see Fig. 7.9, which is a call graph of a hypothetical program.
- The figure shows how one unit calls another. Its adjacency matrix is shown in Fig. 7.10. This matrix may help the testers a lot.

CALL GRAPH-BASED INTEGRATION

- The call graph shown in Fig. 7.9 can be used as a basis for integration testing.
- The idea behind using a call graph for integration testing is to avoid the efforts made in developing the stubs and drivers.
- If we know the calling sequence, and if we wait for the called or calling function, if not ready, then call graph-based integration can be used.

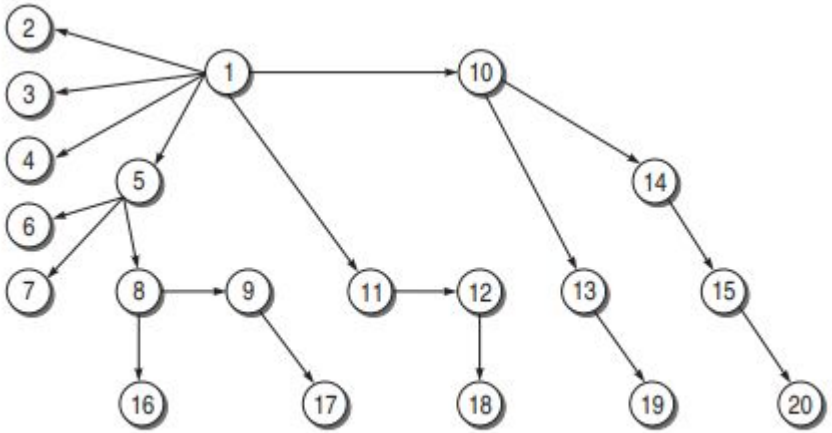


Figure 7.9 Example call graph

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1	×	×	×	×						×	×									
2																				
3																				
4																				
5					×	×	×													
6																				
7																				
8									×							×				
9																	×			
10													×	×						
11												×								
12																		×		
13																			×	
14															×					
15																				×
16																				
17																				
18																				
19																				
20																				

Figure 7.10 Adjacency matrix

PATH-BASED INTEGRATION

- As we have discussed, in a call graph, when a module or unit executes, some path of source instructions is executed/
- And it may be possible that in that path execution, there may be a call to another unit. At that point, the control is transferred from the calling unit to the called unit. This passing of control from one unit to another unit is necessary for integration testing.
- Also, there should be information within the module regarding instructions that call the module or return to the module. This must be tested at the time of integration.
- It can be done with the help of path-based integration defined by Paul C. Jorgenson [78]. We need to understand the following definitions for path-based integration.

PATH-BASED INTEGRATION

- Source node- It is an instruction in the module at which the execution starts or resumes.
- Sink node-It is an instruction in a module at which the execution terminates.
- Module execution path (MEP) It is a path consisting of a set of executable statements within a module like in a flow graph.
- Message-When the control from one unit is transferred to another unit, then the programming language mechanism used to do this is known as a message
- MM-path-It is a path consisting of MEPs and messages. The path shows the sequence of executable statements; it also crosses the boundary of a unit when a message is followed to call another unit. In other words, MM-path is a set of MEPs and transfer of control among different units in the form of messages. MM-path graph It can be defined as an extended flow graph where nodes are MEPs and edges are messages. It returns from the last called unit to the first unit where the call was made. In this graph, messages are highlighted with thick lines.

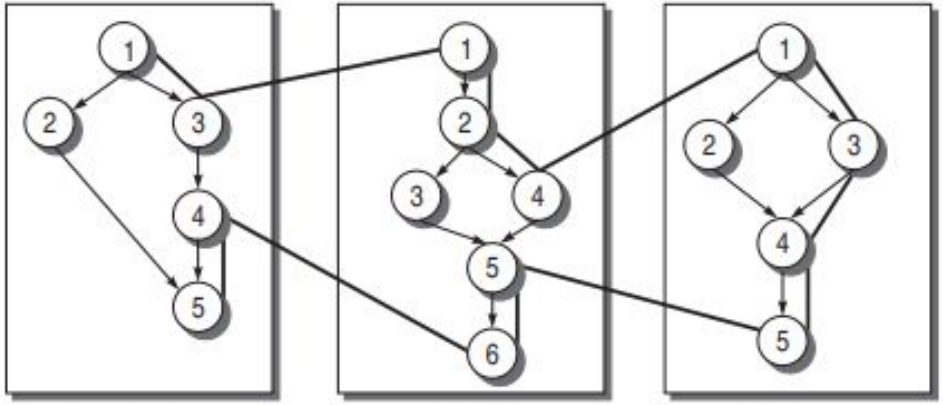


Figure 7.12 MM-path

Table 7.3 MM-path details

	Source Nodes	Sink Nodes	MEPs
Unit A	1,4	3,5	MEP(A,1) = <1,2,5> MEP(A,2) = <1,3> MEP(A,3) = <4,5>
Unit B	1,5	4,6	MEP(B,1) = <1,2,4> MEP(B,2) = <5,6> MEP(B,3) = <1,2,3,4,5,6>
Unit C	1	5	MEP(C,1) = <1,3,4,5> MEP(C,2) = <1,2,4,5>

FUNCTION TESTING

- Function Testing is defined as the process of attempting to detect discrepancies between the functional specifications of a software and its actual behaviour.
- Thus, the objective of function test is to measure the quality of the functional (business) components of the system.
- Tests verify that the system behaves correctly from the user/business perspective and functions according to the requirements, models, or any other design paradigm used to specify the application. The function test must determine if each component or business event:

1. performs in accordance to the specifications,

2. responds correctly to all conditions that may present themselves by incoming events/data,

3. moves data correctly from one business event to the next, and

4. is initiated in the order required to meet the business objectives of the system.

FUNCTION TESTING

The primary processes/deliverables for requirements based function test are discussed below:

- Test planning: During planning, the test leader with assistance from the test team defines the scope, schedule, and deliverables for the function test cycle. He delivers a test plan (document) and a test schedule (work plan).
- Partitioning/functional decomposition: Functional decomposition of a system (or partitioning) is the breakdown of a system into its functional components or functional areas.
- Requirement definition: The testing organization needs specified requirements in the form of proper documents to proceed with the function test. These requirements need to be itemized under an appropriate functional partition.

FUNCTION TESTING

- Test case design: A tester designs and implements a test case to validate that the product performs in accordance with the requirements. These test cases need to be itemized under an appropriate functional partition and mapped/ traced to the requirements being tested.
- Traceability matrix formation: Test cases need to be traced/mapped back to the appropriate requirement. A function coverage matrix is prepared. This matrix is a table, listing specific functions to be tested, the priority for testing each function, and test cases that contain tests for each function.

Table 7.4 Function coverage matrix

Functions/Features	Priority	Test Cases
F1	3	T2,T4,T6
F2	1	T1, T3,T5

- Test case execution As in all phases of testing, an appropriate set of test cases need to be executed and the results of those test cases recorded.

SYSTEM TESTING

- When all the modules have been integrated with the software system, then the software is incorporated with other system elements.
- System testing is the process of attempting to demonstrate that a program or system does meet its original requirements and objectives, as stated in the requirement specification.
- System testing is actually a series of different tests to test the whole system on various grounds where bugs have the probability to occur.
- The ground can be performance, security, maximum load, etc.
- The integrated system is passed through various tests based on these grounds and depending on the environment and type of project.
- After passing through these tests, the resulting system is a system which is ready for acceptance testing which involves the user

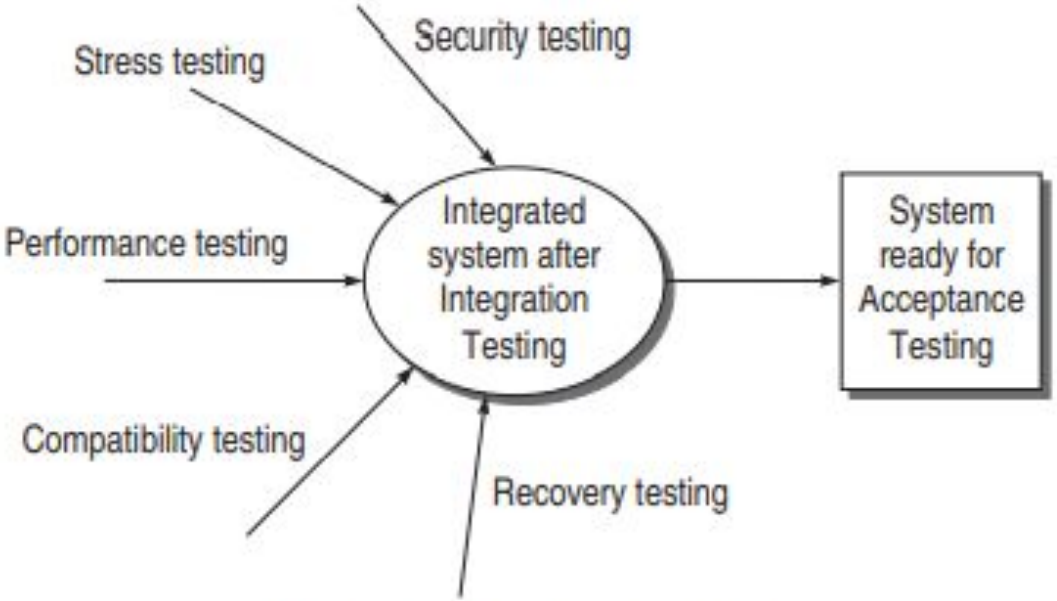


Figure 7.14 System testing

RECOVERY TESTING

- Recovery testing is the activity of testing how well the software is able to recover from crashes, hardware failures, and other similar problems. It is the forced failure of the software in various ways to verify that the recovery is properly perf
- Recovery Testing is just like the exception handling feature of a programming language.
- It is the ability of a system to restart operations after the integrity of the application has been lost.
- The main purpose of this test is to determine how good the developed software is when it faces a disaster.
- Disaster can be anything from unplugging the system which is running the software from power, network etc., also stopping the database, or crashing the developed software itself.
- Eg: While the application is running, suddenly restart the computer and thereafter, check the validity of application's data integrity.
- While the application receives data from the network, unplug the cable and plug-in after awhile, and analyse the application's ability to continue receiving data from that point, when the network connection disappeared.

SECURITY TESTING

- Safety and security issues are gaining importance due to the proliferation of commercial applications on the Internet and the increasing concern about privacy.
- Security is a protection system that is needed to assure the customers that their data will be protected.
- For example, if Internet users feel that their personal data/information is not secure, the system loses its accountability.
- Security testing is the process of attempting to devise test cases to evaluate the adequacy of protective procedures and countermeasures.
- Testers must use a risk-based approach, grounded in both the system's architectural reality and the attacker's mindset, to gauge software security adequately.
- By identifying risks and potential loss associated with those risks in the system and creating tests driven by those risks, the tester can properly focus on areas of code in which an attack is likely to succeed.

PERFORMANCE TESTING

- Performance specifications (requirements) are documented in a performance test plan.
- Ideally, this is done during the requirements development phase of any system development project, prior to any design effort.
- For performance testing, generally a generic set of sample data is created in the project and because of time constraints, that data is used throughout development and functional testing. However, these datasets tend to be unrealistic, with insufficient size and variation, which may result in performance surprises when an actual customer, using much more data than contained in the sample dataset, attempts to use the system.
- The following tasks must be done for this testing:

„ Decide whether to use internal or external resources to perform tests, depending on in-house expertise (or the lack of it).

„ Gather performance requirements (specifications) from users and/or business analysts.

PERFORMANCE TESTING

„ Develop a high-level plan (or project charter), including requirements, resources, timelines, and milestones.

„ Develop a detailed performance test plan (including detailed scenarios and test cases, workloads, environment info, etc).

„ Choose test tool(s).

„ Specify test data needed.

„ Develop detailed performance test project plan, including all dependencies and associated timelines.

„ Configure the test environment (ideally identical hardware to the production platform), router configuration, deployment of server instrumentation, database test sets developed, etc.

„ Execute tests, probably repeatedly (iteratively), in order to see whether any unaccounted factor might affect the results.

LOAD TESTING

- Normally, we don’t test the system with its full load, i.e. with maximum values of all resources in the system. The normal system testing takes into consideration only nominal values of the resources.
- However, there is high probability that the system will fail when put under maximum load.
- Therefore, it is important to test the system with all its limits.
- When a system is tested with a load that causes it to allocate its resources in maximum amounts, it is called load testing.
- The idea is to create an environment more demanding than the application would experience under normal workloads.
- As the load is being increased, transactions may suffer excessive delay.
- For example, when many users work simultaneously on a web server, the server responds slow.
- In this way, through load testing, we are able to determine the maximum sustainable load the system can handle.

STRESS TESTING

- Stress testing is also a type of load testing, but the difference is that the system is put under loads beyond the limits so that the system breaks.
- Thus, stress testing tries to break the system under test by overwhelming its resources in order to find the circumstances under which it will crash.
- The areas that may be stressed in a system are Input transactions, Disk space, Output, Communications, Interaction with users.
- Stress testing is important for real-time systems where unpredictable events may occur, resulting in input loads that exceed the values described in the specification, and the system cannot afford to fail due to maximum load on resources.
- Therefore, in real-time systems, the entire threshold values and system limits must be noted carefully. Then, the system must be stress-tested on each individual value.

COMPATIBILITY TESTING

- Compatibility testing is to check the compatibility of a system being developed with different operating system, hardware and software configuration available, etc.
- Many software systems interact with multiple CPUs. Some software control real-time process and embedded software interact with devices. In many cases, users require that the devices be interchangeable, removable, or re-configurable.
- **Configuration Testing** allows developers/testers to evaluate system performance and availability when hardware exchanges and reconfigurations occur.
- Compatibility may also extend to upgrades from previous versions of the software. Therefore, in **Conversion Testing**, the system must be upgraded properly and all the data and information from the previous version should also be considered.

ACCEPTANCE TESTING

- It is the formal testing conducted to determine whether a software system satisfies its acceptance criteria and to enable buyers to determine whether to accept the system or not.
- After the software has passed all the system tests and defect repairs have been made, the customer/client must be involved in the testing with proper planning.
- The purpose of acceptance testing is to give the enduser a chance to provide the development team with feedback as to whether or not the software meets their needs.
- Acceptance testing must take place at the end of the development process. It consists of tests to determine whether the developed system meets the predetermined functionality, performance, quality, and interface criteria acceptable to the user.
- System testing is invariably performed by the development team which includes developers and testers. User acceptance testing, on the other hand, should be carried out by end-users.

ACCEPTANCE TESTING

Entry Criteria

- System testing is complete and defects identified are either fixed or documented.
- Acceptance plan is prepared and resources have been identified.
- Test environment for the acceptance testing is available.

Exit Criteria

- Acceptance decision is made for the software.
- In case of any warning, the development team is notified.

Acceptance testing is classified into the following two categories: „

- **Alpha Testing**: Tests are conducted at the development site by the end users. The test environment can be controlled a little in this case.
- **Beta Testing:** Tests are conducted at the customer site and the development team does not have any control over the test environment.

ALPHA TESTING

- Alpha is the test period during which the product is complete and usable in a test environment, but not necessarily bug-free.
- It is the final chance to get verification from the customers that the tradeoffs made in the final development stage are coherent.
- Therefore, alpha testing is typically done for two reasons:

(i) to give confidence that the software is in a suitable state to be seen by the customers (but not necessarily released).

(ii) to find bugs that may only be found under operational conditions. Any other major defects or performance issues should be discovered in this stage.

ALPHA TESTING

Entry Criteria to Alpha

- All features are complete/testable (no urgent bugs).
- High bugs on primary platforms are fixed/verified.
- 50% of medium bugs on primary platforms are fixed/verified.
- All features have been tested on primary platforms.
- Performance has been measured/compared with previous releases (user functions).
- Usability testing and feedback (ongoing).
- Alpha sites are ready for installation.

Exit Criteria from Alpha

After alpha testing, we must:

- Get responses/feedbacks from customers.
- Prepare a report of any serious bugs being noticed.
- Notify bug-fixing issues to developers.

BETA TESTING

- Once the alpha phase is complete, development enters the beta phase.
- Beta is the test period during which the product should be complete and usable in a production environment.
- Beta also serves as a chance to get a final ‘vote of confidence’ from a few customers to help validate our own belief that the product is now ready for volume shipment to all customers.
- Versions of the software, known as beta-versions, are released to a limited audience outside the company. The software is released to groups of people so that further testing can ensure the product has few or no bugs.

BETA TESTING

Entry Criteria to Beta

- Positive responses from alpha sites.
- Customer bugs in alpha testing have been addressed.
- There are no fatal errors which can affect the functionality of the software.
- Secondary platform compatibility testing is complete.
- Regression testing corresponding to bug fixes has been done.
- Beta sites are ready for installation.

Guidelines for Beta Testing

- Don't expect to release new builds to beta testers more than once every two weeks.
- Don't plan a beta with fewer than four releases.
- If you add a feature, even a small one, during the beta process, the clock goes back to the beginning of eight weeks and you need another 3–4 releases.

PROGRESSIVE TESTING vs REGRESSIVE TESTING

- Whatever test case design methods or testing techniques, discussed until now, have been referred to as progressive testing or development testing. From verification to validation, the testing process progresses towards the release of the product.
- A system under test (SUT) is said to regress if
- „ a modified component fails, or
- „ a new component, when used with unchanged components, causes failures in the unchanged components by generating side-effects or feature interactions.
- Therefore, now the following versions will be there in the system:

Baseline version: The version of a component (system) that has passed a test suite.

Delta version: A changed version that has not passed a regression test.

Delta build: An executable configuration of the SUT that contains all the delta and baseline components.

- Thus, it can be said that most test cases begin as progressive test cases and eventually become regression test cases.
- Regression testing is not another testing activity. Rather, it is the re-execution of some or all of the already developed test cases

REGRESSION TESTING

- Regression testing is the selective retesting of a system or component to verify that modifications have not caused unintended effects and that the system or component still complies with its specified requirements.
- Thus, regression testing can be defined as the software maintenance task performed on a modified program to instill confidence that changes are correct and have not adversely affected the unchanged portions of the program
- The purpose of regression testing is to ensure that bug-fixes and new functionalities introduced in a new version of the software do not adversely affect the correct functionality inherited from the previous version.
- After a program has been modified, we must ensure that the modifications work correctly and check that the unmodified parts of the program have not been adversely affected by these modifications.
- Thus during regression testing, the modified software is executed on all tests to validate that it still behaves the same as it did in the original software, except where a change is expected

REGRESSION TESTABILITY

- Regression testability refers to the property of a program, modification, or test suite that lets it be effectively and efficiently regression-tested.
- Leung and White [47] classify a program as regression testable if most single statement modifications to the program entail rerunning a small proportion of the current test suit.
- To consider regression testability, a regression number is computed.
- It is the average number of affected test cases in the test suite that are affected by any modification to a single instruction.
- This number is computed using information about the test suite coverage of the program.
- If regression testability is considered at an early stage of development, it can provide significant savings in the cost of development and maintenance of the software.

OBJECTIVES OF REGRESSION TESTING

- It tests to check that the bug has been addressed: The first objective in bugfix testing is to check whether the bug-fixing has worked or not.
- It finds other related bugs: It may be possible that the developer has fixed only the symptoms of the reported bugs without fixing the underlying bug.
- It tests to check the effect on other parts of the program: It may be possible that bug-fixing has unwanted consequences on other parts of a program. Therefore, regression tests are necessary to check the influence of changes in one part on other parts of the program

REGRESSION TESTING TYPES

REGRESSION TESTING TYPES:

- **Bug-Fix regression**: This testing is performed after a bug has been reported and fixed. Its goal is to repeat the test cases that expose the problem in the first place.
- **Side-Effect regression/Stability regression**: It involves re-testing a substantial part of the product. The goal is to prove that the change has no detrimental effect on something that was earlier in order. It tests the overall integrity of the program, not the success of software fixes.

DEFINING REGRESSION TEST PROBLEM

Let us first define the notations used in regression testing before defining the regression test problem.

- P denotes a program or procedure,
- P' denotes a modified version of P,
- S denotes the specification for program P,
- S' denotes the specification for program P',
- P(i) refer to the output of P on input i,
- P'(i) refer to the output of P' on input i, and
- T = {t1,..... tn} denotes a test suite or test set for P

REGRESSION TESTING PROBLEM

Given a program P, it's modified version P', and a test set T that was used earlier to test P; find a way to utilize T to gain sufficient confidence in the correctness of P'.

REGRESSION TESTING TECHNIQUES:

- **Regression test selection technique :** This technique attempt to reduce the time required to retest a modified program by selecting some subset of the existing test suite.
- **Test case prioritization technique:** Regression test prioritization attempts to reorder a regression test suite so that those tests with the highest priority, according to some established criteria, are executed earlier in the regression testing process rather than those with lower priority.
- **Test suite reduction technique:** It reduces testing costs by permanently eliminating redundant test cases from test suites in terms of codes or functionalities exercised.

SELECTIVE RETEST TECHNIQUE

SELECTIVE RETEST TECHNIQUE: We consider the following selective retest techniques:

- Minimization techniques: Minimization-based regression test selection techniques attempt to select minimal sets of test cases from T that yield coverage of modified or affected portions of P.
- Dataflow techniques: Dataflow coverage-based regression test selection techniques select test cases that exercise data interactions that have been affected by modifications.
- Safe techniques: Most regression test selection techniques, minimization and dataflow techniques among them, are not designed to be safe. Techniques that are not safe can fail to select a test case that would have revealed a fault in the modified program. In contrast, when an explicit set of safety conditions can be satisfied, safe regression test selection techniques guarantee that the selected subset T' contains all test cases in the original test suite T that can reveal faults in P'

SELECTIVE RETEST TECHNIQUE

- Ad hoc/Random techniques: When time constraints prohibit the use of a retestall approach, but no test selection tool is available, developers often select test cases based on ‘intuitions’ or loose associations of test cases with functionality. Another simple approach is to randomly select a predetermined number of test cases from T.
- Retest-all technique: The retest-all technique simply reuses all existing test cases. To test P ¢, the technique effectively selects all test cases in T.

REGRESSION TEST PRIORITIZATION

- The regression test prioritization approach is different as compared to selective retest techniques.
- The prioritization methods order the execution of RTS with the intention that a fault is detected earlier.
- In other words, regression test prioritization attempts to reorder a regression test suite so that those tests with the highest priority, according to some established criterion, are executed earlier in the regression testing process than those with a lower priority.
- By prioritizing the execution of a regression test suite, these methods reveal important defects in a software system earlier in the regression testing process.