UNIT 4

IMPLEMENTATION

- The software engineer translates the design specifications into source codes in some programming language.
- The main goal of implementation is to produce quality source codes that can reduce the cost of testing and maintenance.
- The purpose of coding is to create a set of instructions in a programming language so that computers execute them to perform certain operations.
- Implementation is the software development phase that affects the testing and maintenance activities.
- A clear, readable, and understandable source code will make testing, debugging, and maintenance tasks easier.
- Source codes are written for functional requirements but they also cover some nonfunctional requirements.
- Unstructured and structured programming produce more complex and tedious codes than object-oriented, fourth generation languages, component based programming etc.
- A well- documented code helps programmers in understanding the source codes for testing and maintenance.
- Software engineers are instructed to follow the coding process, principles, standards, and guidelines for writing source codes.
- Finally, the code is tested to uncover errors and to ensure that the product satisfies the needs of the customer.

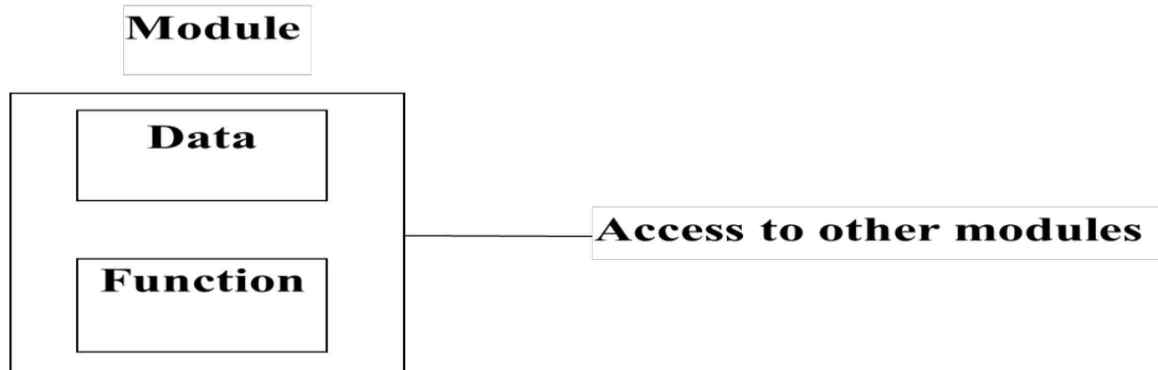
CODING PRINCIPLES

- Coding principles are closely related to the principles of design and modeling.
- Developed software goes through testing, maintenance, and reengineering. .
- Coding principles help programmers in writing an efficient and effective code, which is easier to test, maintain, and reengineer.

  Information Hiding

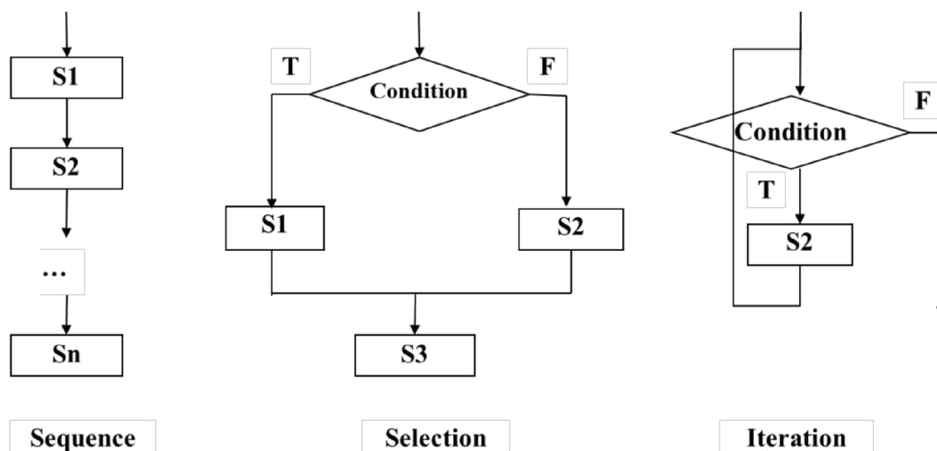
  - Data encapsulation binds data structures and their operations into a single module.
  - The operations declared in a module can access its data structures and allow other modules to access them via interfaces.
  - Other modules can access data structures through access specifiers and interfaces available in modern programming languages.
  - Information hiding is supported by data abstraction, which allows creating multiple instances of abstract data type.

- Most of object-oriented programming languages such as C++, Java etc., support the features of information hiding.
- Structured programming languages, such as C, Pascal, FORTRAN, etc., provide information hiding in a disciplined manner.



Structure Programming Features

- ❖ Structured programming features linearize the program flow in some sequential way that the programs follow during their execution.
- ❖ The organization of program flow is achieved through the following three basic constructs of structured programming.
  - – *Sequence:* It provides sequential ordering of statements, i.e., S1, S2, S3, … Sn.
  - – *Selection:* It provides branching of statements using if-then-else, switch-case, etc.
  - – *Iteration:* A statement can be executed repeatedly using while-do, repeat-until, while, etc.



Maximize Cohesion and Minimize Coupling

- – Writing modular programs with the help of functions, code, block, classes, etc., may increase dependency among modules in the software.
- – The main reason is the use of shared and global data items.

- Shared data should be used as little as possible.
- Minimizing dependencies among programs will maximize cohesion within modules; that is, there will be more use of local data rather than global data items.
- *High cohesion and low coupling make a program clear, readable, and maintainable.*

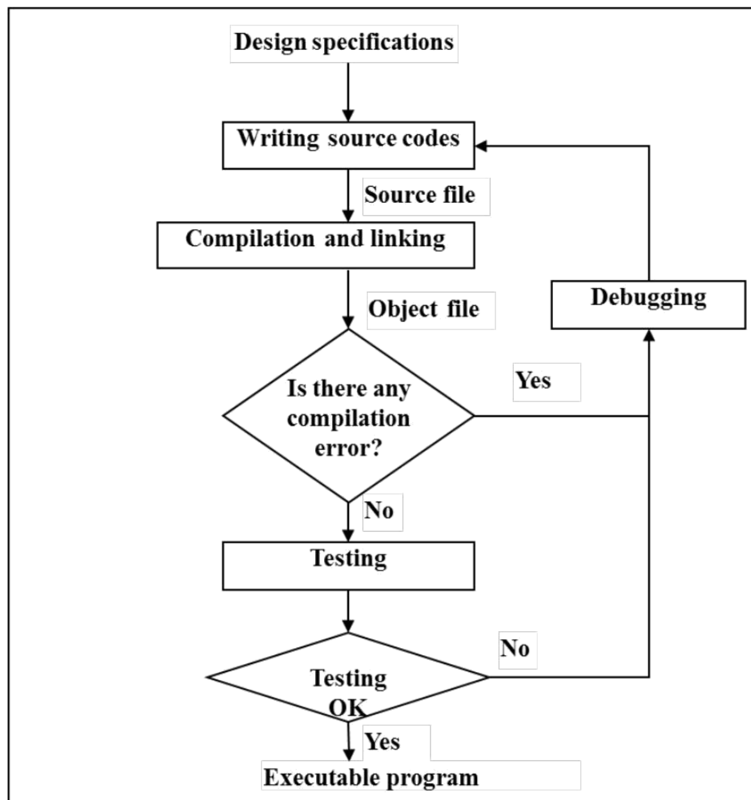
- Code Reusability allows the use of existing code several times.
- KIS (Keep It Simple)
  - Most of the software are structurally complex but can be made simple by using modularization and other designing principles.
- Simplicity, Extensibility, and Effortlessness
- Code Verification
- Code Documentation
- Separation of Concern
- Follow Coding Standards, Guidelines, and Styles

CODING PROCESS:

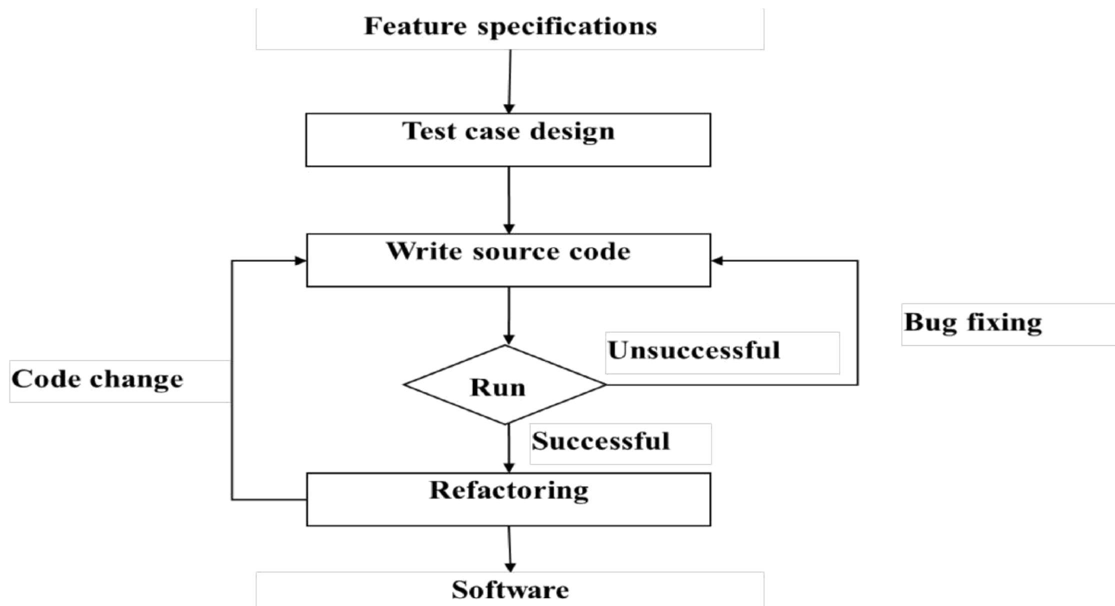
- The coding process describes the steps that programmers follow for producing source codes.
- The coding process allows programmers to write bug-free source codes.
- It involves mainly coding and testing phases to generate a reliable code.
- The coding process describes the steps that programmers follow for producing source codes.
- The coding process allows programmers to write bug-free source codes.
- Two widely used coding processes.
  - *Traditional Coding Process*
  - *Test-driven Development (TDD)*

Traditional Coding Process

- The traditional programming process is an iterative and incremental process which follows the “write-compile-debug” process.
- TDD was introduced by Extreme Programming (XP) in agile methodologies that follow the “coding with testing” process



- **Test-Driven Development**
  - Developed software goes through a repeated maintenance process due to lack of quality and inability to satisfy the customer needs.
  - System functionality is decomposed into several small features.
  - Test cases are designed before coding.
  - Unit tests are written first for the feature specification and then the small source code is written according to the specification.
  - Source code is run against the test case.
  - It is quite possible that the small code written may not meet the requirements, thus it will fail the test.
  - After failure, we need to modify the small code written before to meet the requirements and run it again.
  - If the code passes the test case implies the code is correct. The same process is repeated for another set of requirements specification.



CODE VERIFICATION

- Code verification is the process of identifying errors, failures, and faults in source codes, which cause the system to fail in performing specified tasks.
- Code verification ensures that functional specifications are implemented correctly using a programming language.
- There are several techniques in software engineering which are used for code verification.
  - *Code review*
  - *Static analysis*
  - *Testing*
- Code review
  - It is a traditional method for verification used in the software life cycle. It mainly aims at discovering and fixing mistakes in source codes.
  - Code review is done after a successful compilation of source codes. Experts review codes by using their expertise in coding.
  - The errors found during code verification are debugged.
- Following methods are used for code review:
  - *Code walkthrough*
  - *Code inspection*
  - *Pair programming*

Code walkthrough

- A code walkthrough is a technical and peer review process of finding mistakes in source codes.
- The walkthrough team consists of a review and a team of reviewers.

- The reviewers examine the code either using a set of test cases or by changing the source code.
- During the walkthrough meeting, the reviewers discuss their findings to correct mistakes or improve the code.
- The reviewers may also suggest alternate methods for code improvement.
- The walkthrough session is beneficial for code verification, especially when the code is not properly documented.
- Sometimes, this technique becomes time consuming and tedious. Therefore, the walkthrough session is kept short.

Code inspection

- It aims at detecting programming defects in the source code.
- The code inspection team consists of a programmer, a designer, and a tester.
- The inspectors are provided the code and a document of checklists.
- In the inspection process, definite roles are assigned to the team members, who inspect the code in a more rigorous manner. Also, the checklists help them to catch errors in a smooth manner.
- Code inspection takes less time as compared to code walkthrough.
- Most of the software companies prefer software inspection process for code review.

Pair programming

- It is an extreme programming practice in which two programmers work together at one workstation, i.e., one monitor and one keyboard. In the current practice, programmers can use two keyboards.
- During pair programming, code review is done by the programmers who write the code. It is possible that they are unable to see their own mistakes.
- With the help of pair programming, the pair works with better concentration.
- They catch simple mistakes such as ambiguous variable and method names easily. The pair shares knowledge and provides quick solution.
- Pair programming improves the quality of software and promotes knowledge sharing between the team members.

Static analysis

  - Source codes are not executed rather these are given as input to some tool that provides program behavior.
  - Static analysis is the process of automatically checking computer programs.
  - This is performed with program analysis tools.
  - Static analysis tools help to identify redundancies in source codes.
  - They identify idempotent operations, data declared but not used, dead codes, missing data, connections that lead to unreachable code segments, and redundant assignments.
  - They also identify the errors in interfacing between programs. They identify mismatch errors in parameters used by the team and assure compliance to coding standards.

Testing

  - Dynamic analysis works with test data by executing test cases.

- Testing is performed before the integration of programs for system testing.
- Also, it is intended to ensure that the software ensures the satisfaction of customer needs.

CODE DOCUMENTATION:

- Software development, operation, and maintenance processes include various kinds of documents.
- Documents act as a communication medium between the different team members of development.
- They help users in understanding the system operations.
- Documents prepared during development are problem statement, software requirement specification (SRS) document, design document, documentation in the source codes, and test document.
- These documents are used by the development and maintenance team members.
- There are following categories of documentation done in the system
    - *Internal documentation*
    - *System documentation*
    - *User documentation*
    - *Process documentation*
    - *Daily documentation*
- Implementation is the coding phase in which software engineers translate the design specifications into source codes in a programming language.
- Coding principles help programmers in writing an efficient and effective code, which is easier to test, maintain, and reengineer.
- Coding standards play the role of maintaining commonality in coding styles among programmers.
- Code verification is the process of identifying errors, failures, and faults in source codes, which cause the system to fail in performing the specified tasks.
- The *traditional coding process* and *test-driven development (TDD)* are two widely used coding processes.
- Documentation help users in understanding the system operations. Documents prepared during development are problem statement, software requirement specification (SRS) document, design document, documentation in the source codes, and test document.

Software Testing

Introduction:

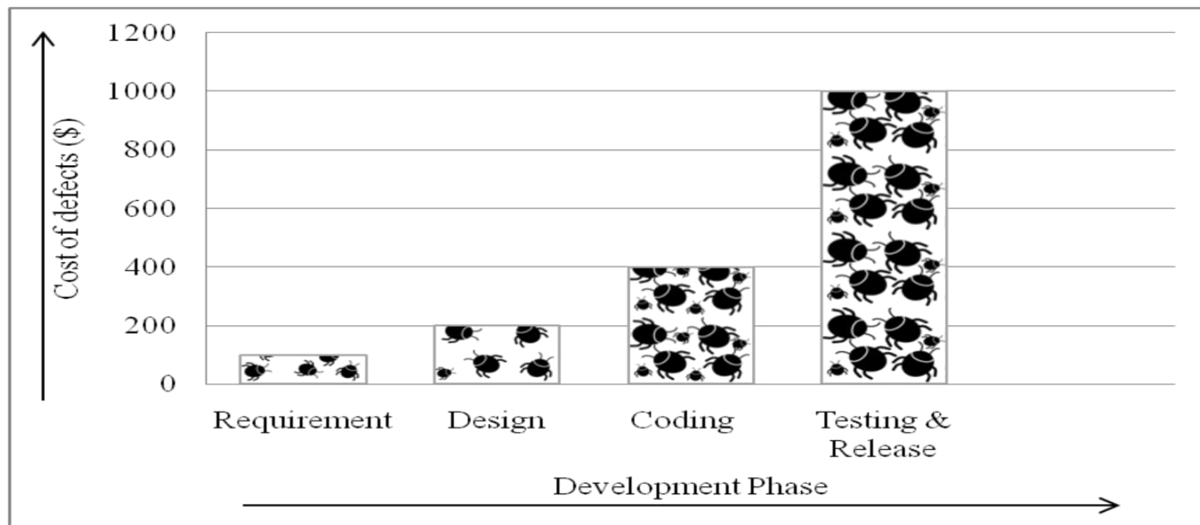
- ✓ *Software testing is the process of finding defects in the software so that these can be debugged and the defect-free software can meet the customer needs and expectations.*
- ✓ Software testing is one of the important phases in software development life cycle.
- ✓ A quality software can be achieved through testing.
- ✓ Effective testing reduces the maintenance cost and provides reliable outcomes.
- ✓ Example of Ineffective testing -the Y2K problem.
- ✓ *The intention of software testing process is to produce a defect-free system*.

TESTING FUNDAMENTALS:

- ✓ *Error* is the discrepancy between the actual value of the output of software and the theoretically correct value of the output for that given input.
- ✓ Error also known as variance, mistake, or problem is the unintended behavior of software.
- ✓ *Fault* is the cause of an error. *Fault* is also called defect or bug in the manifestation of one or more errors. It causes a system to fail in achieving the intended task.
- ✓ *Failure* is the deviation of the observed behavior from the specified behavior.
- ✓ It occurs when the faulty code is executed leading to an incorrect outcome. Thus, the presence of faults may lead to system failure.
- ✓ A failure is the manifestation of an error in the system or software.

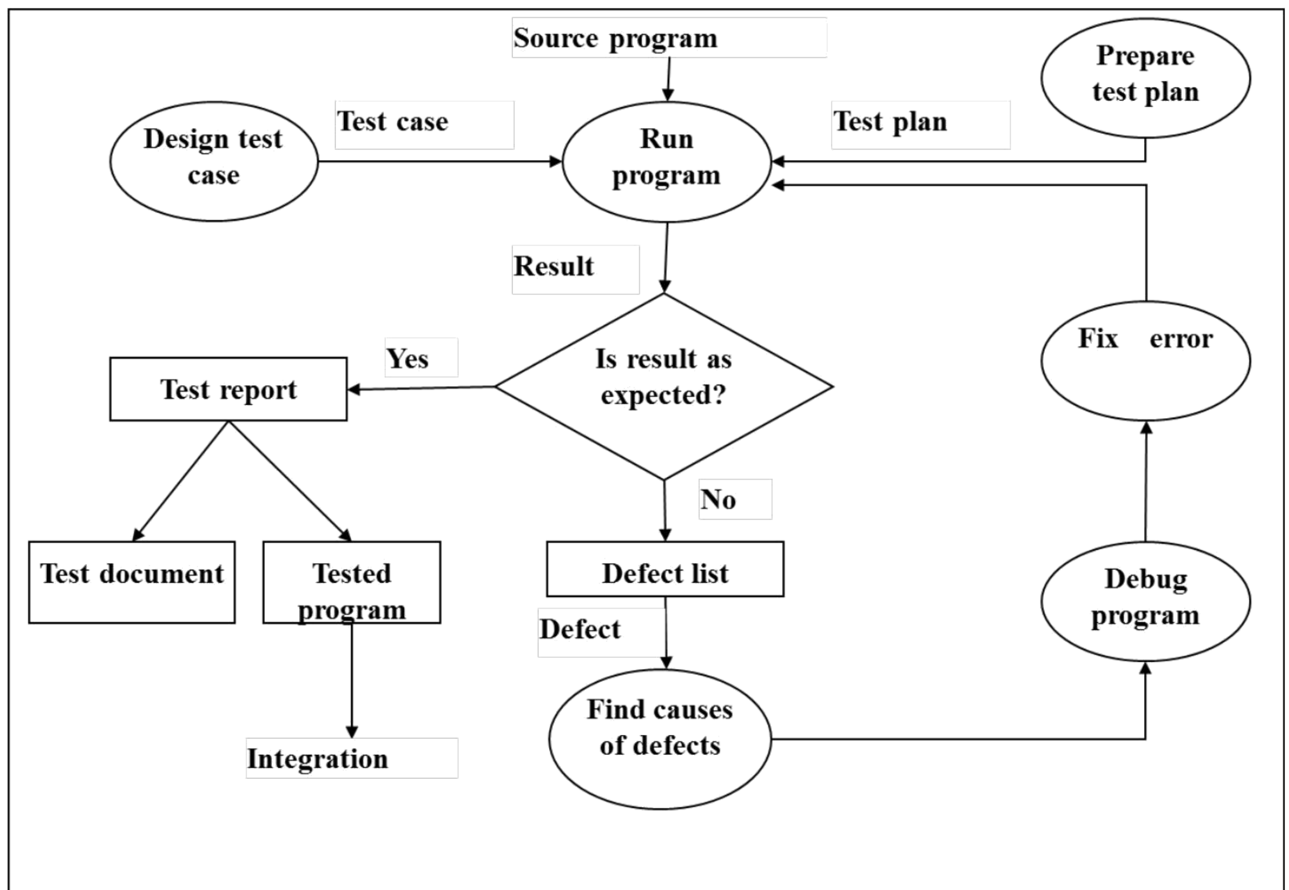
The Cost of Defects

Testing becomes more costly if the errors are not taken care in the earlier phases. In this case, defects increase in multiples in addition to the defects from previous phases.



Testing Process:

- ✓ Testing is a disciplined process of finding and debugging defects to produce defect-free software.
- ✓ During testing, a test plan is prepared that specifies the name of the module to be tested, reference modules, date and time, location, name of the tester, testing tools, etc.
- ✓ Software engineers design test cases while writing the source codes, Testers may also involve in test case design.
- ✓ Test cases include the input, output, and the conditions.
- ✓ Software tester runs the program using test cases according to the test plan and observes the test results



The Role of Software Testers:

- The goal of testers is to confirm that the software works properly by finding defects as early as possible and ensuring that these are fixed

Software tester does the following tasks for testing the software.

- ✓ *Prepare the test plan and test data.*
- ✓ *Design test cases and test scripts.*
- ✓ *Set up test environment.*
- ✓ *Perform testing.*
- ✓ *Track the defects in the defect management system.*
- ✓ *Participate in the test case review meetings.*
- ✓ *Prepare test report.*
- ✓ *Follow software standards.*

Software testing tools

- ✓ Software testers perform testing using software testing tools. There are various automated tools for testing, such as :
  - ✓ *Load Runner.*

- ✓ *Jmeter.*
- ✓ *DBMonster.*
- ✓ *TestTube.*
- ✓ *WinRunner.*
- ✓ *Pounder.*
- ✓ *Marathon, and so on.*

✓ The success of an automated tool depends on the design of test cases.

TEST PLANNING:

- ✓ Testing is a long activity in which several test cases are executed by different members of test team and may be in different environment and at different locations and machine.
- ✓ Test planning specifies the scope, approach, resources, and schedule of the testing activities.
- ✓ Test planning includes the following activities:
  - ✓ *Create test plan.*
  - ✓ *Design test cases.*
  - ✓ *Design test stubs and test drivers.*
  - ✓ *Test case execution.*
  - ✓ *Defect tracking and statistics.*
  - ✓ *Prepare test summary report.*

Creation of a Test Plan:

- A *test plan* is a document that describes the scope and activities of testing. It is a formal document for testing software.
- A test plan contains the following attributes:
  - *Test plan ID*
  - *Purpose*
  - *Test items*
  - *References*
  - *Features to be tested*
  - *Schedule*
  - *Responsibilities*
  - *Test environment*
  - *Test case libraries and standards*
  - *Test strategy*
  - *Test deliverables*
  - *Release criteria*
  - *Expected risk*

Design Test Cases:

- *A test case is a set of inputs and expected results under which a program unit is exercised with the purpose of causing failure and detecting faults*.
- A good test case is one that has the high probability of detecting defects in the system.
- A well-designed test case can be traceable, repeatable, and can be reused in other software development.
- *The intention of designing set of test cases for testing is to prove that program under test is incorrect.*
- The test case selection is the main objective to detect errors in the program unit.
- A possible way is to exercise all the possible paths and variables to find undiscovered errors. But performing *exhaustive testing* is difficult because it takes a lot of time and efforts. An exhaustive testing includes all possible input to the program unit.
- **Test script:** A test script is a procedure that is performed on a system under test to verify that the system functions as expected. Test case is the baseline to create test scripts using automated tool.
- **Test suite:** A test suite is a collection of test cases. It is the composite of test cases designed for a system.
- **Test data:** Test data are needed when writing and executing test cases for any kind of test. Test data is sometimes also known as *test mixture*.
- **Test harness:** Test harness is the collection of software, tools, input/output data, and configurations required for test.
- **Test scenario:** Test scenario is the set of test cases in which requirements are tested from end to end. There can be independent test cases or a series of test cases that follow each other.
- A test case includes the following fields:
  - *Test plan ID*
  - *Test case ID*
  - *Feature to be tested*
  - *Preconditions*
  - *Test script or test procedure*
  - *Test data*
  - *Expected results*

- **Example: Test case to issue a book to the student member.**

Test plan ID	TP-001
Test case ID	LTC-01
Feature to be tested	Issue a book to the member of library
Preconditions	1. Library membership is compulsory 2. Book quota limit should not exceed 5 for a student member
Test script	1. Verify library membership 2. Check book availability 3. Check the issue limits of books 4. Issue book 5. Add book in the account of member 6. Update library catalogue
Test data	1. Valid memberships: 'CS-5Jan12-30Jun12-MS-10' 2. Invalid membership: 'CS-4Jan12-30Jun12-MS-00' 3. Valid book limit: 1, 4 4. Invalid book limit: -1, 5, 6
Expected results	1. Book should be issued if there is valid membership and valid book limit. 2. Display renew membership for invalid membership 3. The book limit is over for invalid book limit
Test status	Pass

Test Stubs and Test Drivers:

✓ A *test driver* is a simulated module that calls the module under test. The test driver specifies the parameters to call the module under test.

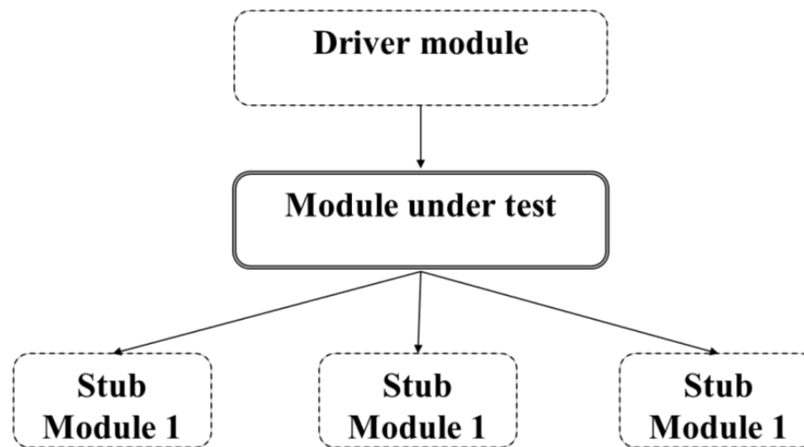


Figure 9.3: Test driver and test stub modules

- ✓ A *test stub* is a simulated module that is called by the module under test.
- ✓ The test stub and test drivers are the dummy modules that are basically written for the purpose of providing input/output or the interface behavior for testing.
- ✓ Test stub and test drivers are required at the time of unit testing a module.
- ✓ Design of test stub takes more effort as compared to test driver.

Test Case Execution:

- ✓ Once the test cases, test drivers, and test stubs are designed for a test plan, the next task is to execute the test cases.
- ✓ The test environment is set up to perform testing.
- ✓ Software tester runs the test procedure one by one using valid and invalid data and observes the results.
- ✓ On executing test cases, the expected results and their behavior is recorded in a test summary report.

Test Summary Report:

- ✓ Test summary report is prepared to ensure whether the module under test satisfies the acceptance criteria or not.
- ✓ This summary report is directed to the stakeholders to know the status of module.
- ✓ Test summary report covers the results of the items from the test plan, which were planned at the beginning to test the module.
- ✓ It includes the number of test cases executed and the type of errors observed.

Defect Tracking and Statistics:

- A project has a lot of defects, which are inspected, retested, and managed in a *test log*.

- Test team tracks various aspects of the testing progress, such as the location of defective modules and estimation of progress with respect to the schedule, resources, and completion criteria.
- A record of the ignored defects, unresolved defects, stopped due to extra effort and resource requirements, etc., are managed.
- The defects whose identification and the cause of occurrence is determined are debugged, fixed, and verified before closing the testing.

BLACK-BOX TESTING:

- ✓ *Black-box testing* is performed on the basis of functions or features of the software. In black-box testing, only the input values are considered for the design of test cases.
- ✓ The output values that the software provides on execution of test cases are observed. The internal logic or program structures are not considered during black-box testing.
- ✓ It is also known as *behavioral* or *functional testing*.
- ✓ There are a number of black-box test case design methods
  - ➢ *Equivalence class partitioning*
  - ➢ *Boundary value analysis*
  - ➢ *Cause-effect graphing*
  - ➢ *Error guessing*

Equivalence Class Partitioning:

- ✓ Equivalence class partitioning method allows to partition the input domain into a set of equivalence classes (i.e., sub domains).
- ✓ The equivalence class partitioning method has the following two aspects:
  - ➢ *Design of equivalence classes*
  - ➢ *Selection of test input data*
- ✓ *Design of equivalence classes:* The design of equivalence class is made by splitting an input domain into several equivalence classes and these are expected to have different behavior. \
- ✓ *Selection of test input data:* The selection of test input data has two ranges for each equivalence class, i.e., valid input and invalid input data. The valid input data belongs to the range of equivalence class.

- **Example 9.2:** Design test cases to find the characters from ASCII numbers using equivalence class partitioning method.
- The range of ASCII values vary from 0 to 256. The equivalence classes for the ASCII character set is shown below. An ASCII character may belong to one of the following equivalence classes (ECs).

- 0-31 EC1
- 32-95 EC2
- 96-127 EC3
- 128-256 EC4

- EC1: Control characters
- EC2: Basic printable characters
- EC3: Extended printable characters
- EC4: Unicode characters

Equivalence class	Valid test data	Invalid test data
EC1	13, 0, 23	-1, 33, 103
EC2	35, 95, 59	23, 99, 31
EC3	97, 117, 100,	91, 129, 172
EC4	138, 246, 182	125, 258, 265

Boundary Value Analysis:

- The boundary value analysis is the special case of equivalence class partitioning method that focuses on the boundary of the equivalence classes.
- It is generally observed that the human may make mistakes at the boundary level.
- Boundary value analysis is based on the idea of equivalence class partitioning. The boundary values are identified by relating the elements of input domain. For example, relational operators such as <, <=, >, >=, ==, etc., can be applied to relate the boundary level elements.
- Some of the boundary conditions can be 0 or negative values, empty files, missing files, sequencing errors, invalid number of parameters, data ranges, etc., and therefore, the boundary conditions are verified to ensure the correct outcomes.
- Test data are selected in such a way that the boundary values will appear in at least one of the test input data.

Boundary Value Analysis:

- **Example 9.3:** The student grading system is used to allot grades in the subjects. The grades can be A (86 to 100), B (61 to 85), C (46 to 60), D (30 to 45), and F (below 30).

Equivalence class	Test input domain	Test data
EC1	A (>=86 to <=100)	85, 87, 101
EC2	B (>=61 to <=85)	60, 84, 86
EC3	C (>=46 to <=60)	45, 59, 61
EC4	D (>=30 to <=45)	29, 31, 46
EC5	F (<30)	28, 29, 30, 31

Cause-Effect Graphing:

- The main drawback of equivalence class partitioning and boundary value analysis methods is the consideration of only single input domain.
- Cause-effect graphing technique begins with finding the relationships among input conditions known as *causes* and the output conditions known as *effects*.
- A cause is any condition in the requirement that affects the program output. Similarly, an effect is the outcome of some input conditions.
- The logical relationships among input and output conditions are expressed in terms of cause-effect graph.
- Each condition (either cause or effect) is represented as a node in the cause-effect graph. Each condition has the value whether true or false.
- The requirements are the basis for designing the cause-effect graph. The system is expected to perform tasks as it is intended.
- Example:

"Cash withdrawal" depends upon the "valid pin," "valid amount" and "cash availability" in the account. Using the case-effect graphing technique, the requirements can be stated in causes and effects as follows:

- *Causes:*
  - C1: Enter valid amount
  - C2: Enter valid pin
  - C3: Cash available in the account
- *Effects:*

- E1: Cash withdrawal

- Example: E1 will be successful if C1, C2, and C3 are true. This can be represented in the cause-effect graph as shown in Figure below. There can be other causes, such as invalid pin, invalid amount, cash not available, or the combination of these causes. Similarly, there can be the effects according to these causes.

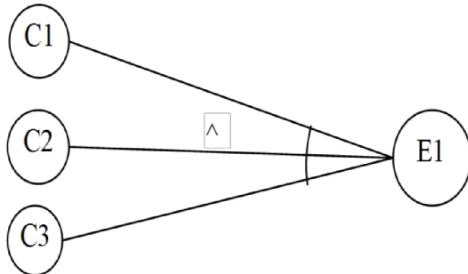
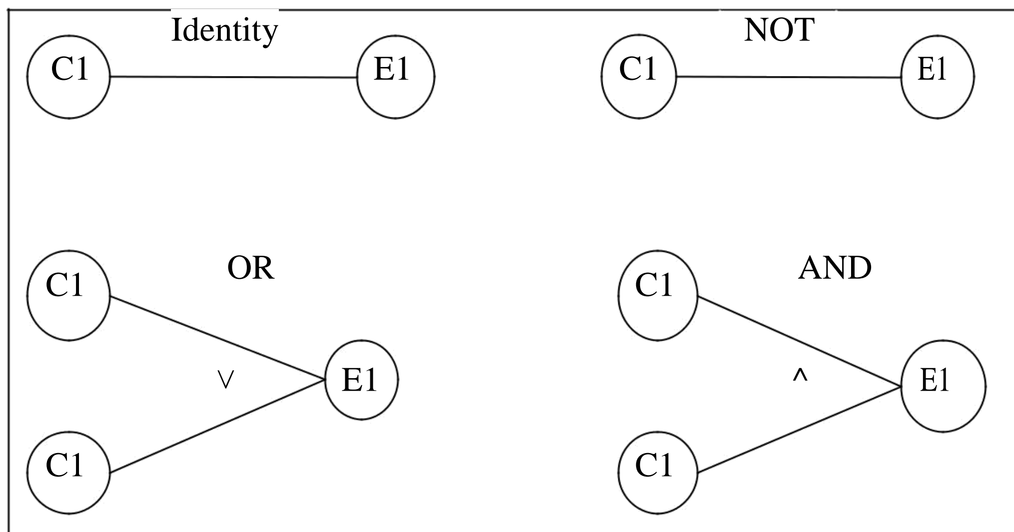


Figure 9.4: Cause-effect graph for cash withdrawal

Notations for cause-effect graph



- The process of cause-effect graphing testing is as follows:

  1. From the requirements, identify causes and effects and assign them a unique identification number.
  2. The relationships among causes and effects are established by combining the causes and effects; and annotated into the cause-effect graph.
  3. Transform the cause-effect graph into the decision table and each column in the decision table represents a test case.
  4. Generate tests from the decision table.

- **Example 9.4: Perform cause-effect graphing technique to issue a book to the student member of the library. The membership is provided for a session that can be renewed. This example is illustrated in Example 9.1.**
- In this example, the causes and the effects identified are as follows:
- *Causes:*

  C1: Library membership is valid

  C2: Membership expired

  C3: Verify book limit

  C4: Verify book availability

- *Effects:*

  E1: Renew membership

  E2: Exceed book limit

  E3: Issue book

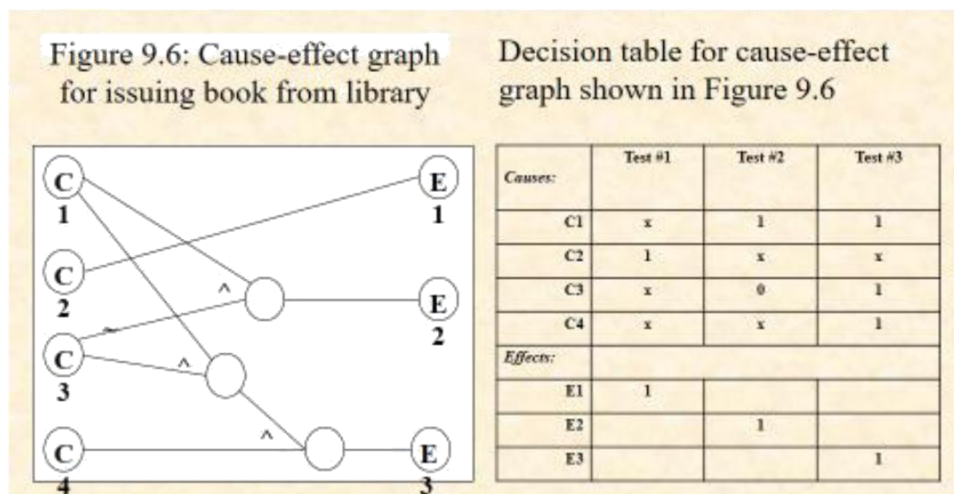


Figure 9.6: Cause-effect graph for issuing book from library

Decision table for cause-effect graph shown in Figure 9.6

Causes:	Test #1	Test #2	Test #3
C1	x	1	1
C2	1	x	x
C3	x	0	1
C4	x	x	1
Effects:			
E1	1		
E2		1	
E3			1

Error Guessing:

- ✓ The error guessing technique is based on guessing the error-prone areas in the program that might be done by the programmer.
- ✓ Error guessing is an intuitive and ad-hoc process of testing. Software testers use their experiences and knowledge to design test cases for such error-prone situations.
- ✓ Let us discuss some of such error-prone situations.
  - ✓ The Boolean variables have the values true (1) or false (0). There might be the chance of alteration of these values from 1 to 0 and vice versa.
  - ✓ The NULL position is another error-prone area in the program.

WHITE-BOX TESTING

- White-box testing is concerned with exercising the source code of a module and traversing a particular execution path.
- The internal logics, such as control structures, control flow, and data structures are considered during the white-box testing.
- White-box testing methods are applied at integration and testing phases**.** White-box testing is also known as *glass-box testing* or *structural testing.*
- The following white-box testing methods are widely used for testing the software:
  - *Control flow based testing*
  - *Path testing*
  - *Data flow based testing*
  - *Mutation testing*

Control Flow Based testing

- The control flow based testing strategy focuses on the control flows in the program.
- The goal of control flow based testing methods is to satisfy *test adequacy criteria*.
- A test is said to be adequate if it executes each statement at least once
- The goal of these coverage tests is to ensure that no defect should remain uncovered in the program fragments.
- Following are the control flow base coverage testing:
  - ***Statement coverage testing***
  - ***Branch coverage testing***
  - ***Condition coverage testing***

Statement coverage testing

- A source program written in a programming language consists of several statements. The statements are logically grouped into program blocks.
- The aim of statement coverage is to design test cases so that every statement of the program can be executed at least once during testing.

- Consider the following example written in C language: main ()

```
{
int a,b;
printf( "%d", a, b);
if (a < b)
printf( "%d", a+b );
else
printf( "%d", a*b);
}
```

- The test case for condition coverage criteria for above program should be made to cover both *true* and *false* conditions.
- For example, the following two test cases of a test suite will be adequate for statement coverage of the above program:

  Test suite = { (a=1,b=2), (a=2,b=1) }

- This test suite will cover all the statements and even the if-else block of the program.
- The statement coverage is an appealing technique but executing test cases with single input value cannot ensure proper results for all other input values.

Branch coverage testing

- The branch coverage testing is also known as decision coverage.
- In this coverage criterion, test cases are designed in such a way that all the outcomes of the decision have been considered.
- The branch coverage is more powerful than statement coverage criteria.
- Consider the following code fragment of C language with a test data { -2 }:

```
int a, b;
if (a < 0)
    a=a+b;
```

- On executing the above code for test data {-2}, it satisfies the statement coverage because it evaluates to *true*.
- However, it is not adequate for branch coverage because the decision (a<0) will not evaluate to *false*.

Condition coverage testing

- ✓ The simple conditions, such as (a<0) are covered using branch coverage criteria. For complex conditions use this testing technique.
- ✓ Complex conditions, which are made using logical operations, such as AND, OR, and XOR. In addition, negation operator NOT (~) is used to negate the outcome of a condition.
- ✓ The complex conditions are said to be tested if all its simple conditions are tested to true and false.
- ✓ see the following program fragment in C language:

```
int a, b;
if (a>=0 && b>0)
        a=a+b;
  else
        a=a-b;
```

- Consider the following test cases for the above program: Test case 1 = {a= 0, b= 2},

  Test case 2 = {a= 1, b= -1}
- The first test case will cover the *if* part but it will not be adequate for the *else* part and vice-versa.
- Therefore, both the test cases will cover statement coverage, branch coverage, and the condition coverage in the above program.

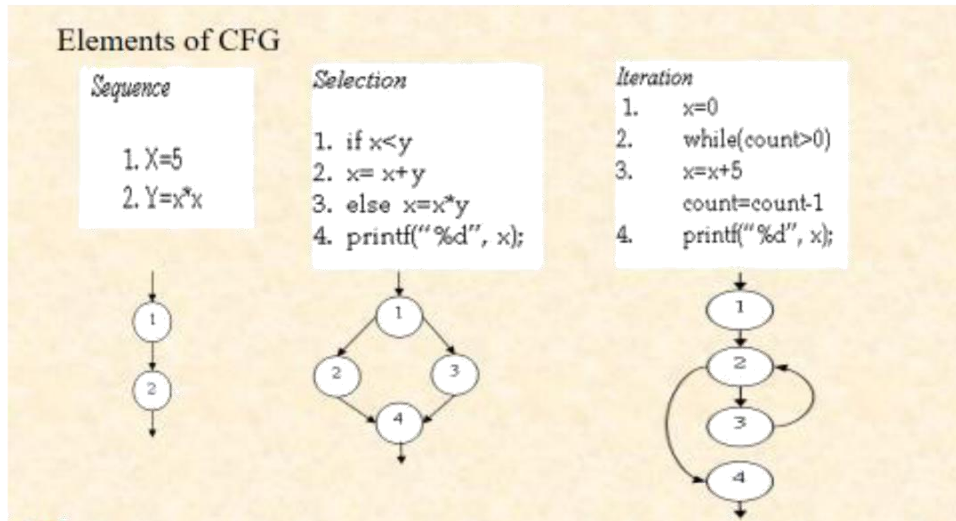
Path Testing

- ✓ Path testing is another white-box testing method, which focuses on identifying independent paths in a program.
- ✓ The purpose of identifying paths is to find the logical complexity measure (i.e., McCabe Cyclomatic complexity) in the program that helps to design test cases for all such independent paths.
- ✓ The idea is to design test cases to exercise all independent paths at least once during testing, the presence of faults will lead to failures.
- ✓ An easy way to identify paths in the program is the **control flow graph (CFG**).
- ✓ CFG describes the flow of control in the program through program graph.

Control Flow Graph (CFG)

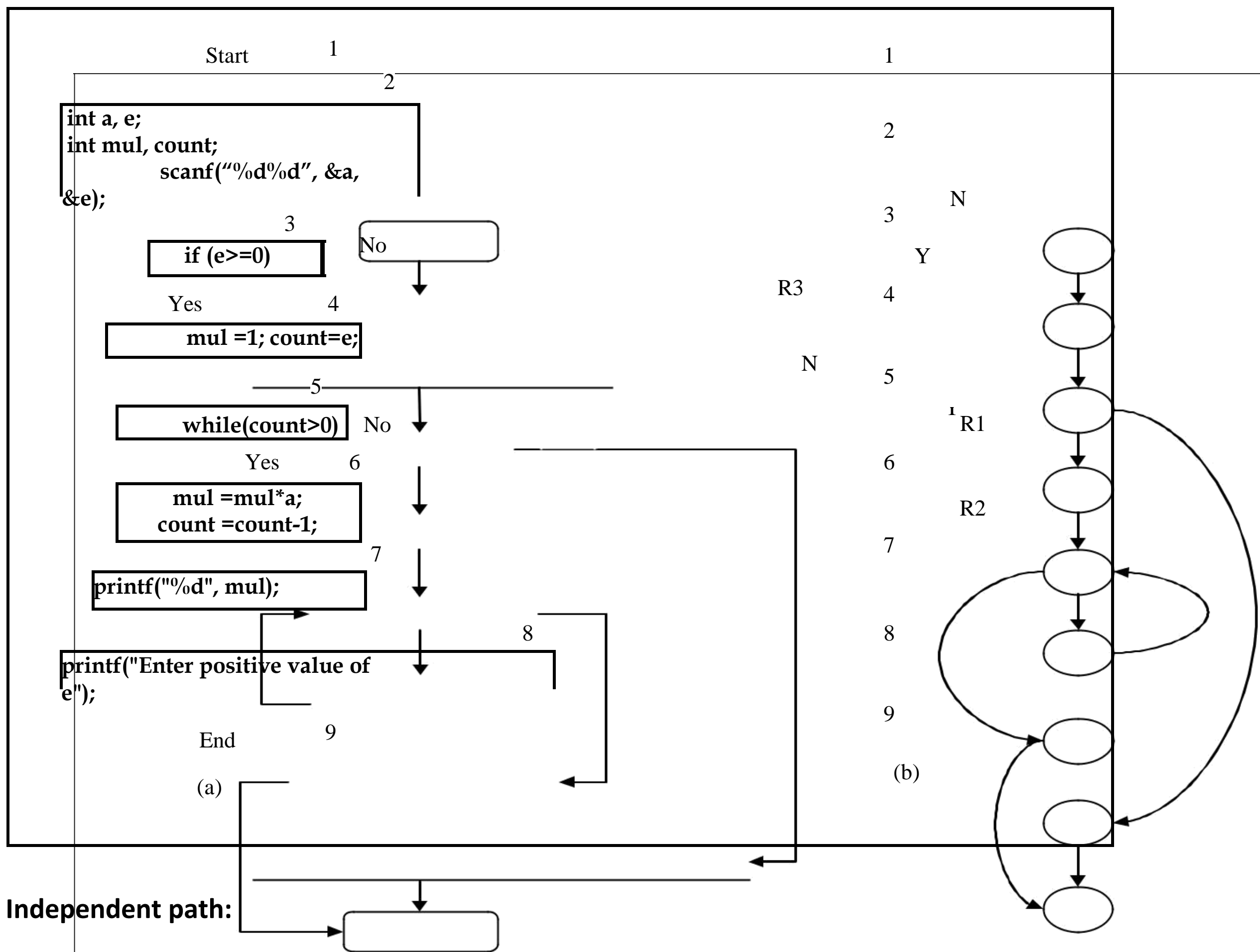
- ➢ A control flow graph (CFG) is also known as *flow graph* or *program graph.* It describes the flow of control within the program.
- ➢ It is a finite set of nodes and a finite set of directed edges.
- ➢ There is a *start* and an *end* node in the graph. The start node has no incoming edges and the end node has no outgoing edges.

- Every node in the graph is reachable from the start node and terminates at the end node.
- A *basic block* is a sequence of statements with a single entry and a single exit in the program.



Example: Write a program to compute a^e in C language and draw its corresponding CFG.

```
main(){                 int a, e;
            int mul, count;
            scanf("%d%d", &a, &e);
            if (e>=0) {
                  mul =1; count=e;
                  while(count>0)
                        {
                        mul =mul*a;
                        count =count-1;
                        }
                  printf("%d", mul);
                  }
            else
                  printf("Enter positive value of e");
}
```



Independent path:

- ✓ An independent path is any path through the program that introduces at least one new edge that is not included in any other path before it.
- ✓ The set of independent paths guarantee that all the statements and conditions will be executed at least once in the program.
- ✓ For example, there are following three independent paths for Figure 9.9.

P1: 1→ 2→ 3→ 8→ 9

P2: 1→ 2→ 3→ 4→ 5→ 6→ 7→ 9

P3: 1→ 2→ 3→ 4→ 5→ 6→ 5→ 7→ 9

Cyclamate Complexity Measure

- ➢ Cyclomatic complexity is the measurement of logical complexity in the program.
- ➢ The cyclomatic complexity is a number, which is computed through certain analytical formulas.

- The cyclomatic complexity number defines the required number of independent paths in a given CFG.
- This number helps to design the required number of test cases to ensure that each statement in the program will be executed at least once.
- It is computed through CFG.
- There are three ways to compute cyclomatic complexity

1. From CFG, cyclomatic complexity(C) is computed as follows:

    C = E - N + 2;

    where E is the number of edges in the CFG and N is the number of nodes in CFG. For example, the cyclomatic complexity (C) for the CFG shown in Figure 9.9 is as follows: C = 10 - 9 + 2 = 3.

2. From CFG, another way to compute cyclomatic complexity (C) is as follows:

    C = P + 1

    Where P indicates the predicates, i.e., number of selection and iteration nodes from where the branches are emerged. For example, the cyclomatic complexity (C) for CFG shown in Figure 9.9 is as follows: C = 2 + 1 = 3.

3. The number of *regions* is equal to the cyclomatic complexity. The *region* is the area bounded by edges and nodes in CFG. The outside area is also considered as a region at the time of computing cyclomatic complexity. For example, the cyclomatic complexity for the CFG shown in Figure 9.9 ìs 3 (i.e., R1 + R2 + R3).

Design of test cases

- The purpose of constructing CFG and finding the cyclomatic complexity is to design test case that can execute every statement of the program at least once. The process of path testing to design test cases from CFG is as follows:
    - *Step1: Construct CFG*
    - *Step2: Compute cyclomatic complexity*
    - *Step3: Identify independent paths*
    - *Step4: Prepare test cases*
- One possible test suite consisting of three test cases is as follows:
    - Test case 1= {a = 3, e = -1}
    - Test case 2= {a = 0, e = 1}
    - Test case 3= {a = 1, e = 0}

Data flow based testing

- Data flow based testing concentrates on data flow representation rather than control flow in the program. Data flow based test criteria focuses on data declared and used in the program to specify the test cases.
- Data flow based testing uses *definition-use (def-use) graph*. It describes the flow of definitions across basic blocks in the program.
- The def-use graph represents the data occurrences, which are discussed in the following three categories, namely, def, c-use, and p-use .

Definitions (def)

- A program contains variables in the program statements. The variable names in assignment statement are defined at right hand side. For example:

  a = b + c;

- This statement defines a that uses b and c. Similarly, the following statements

  P = &a;

  T = P + 5;

Computations (c-use)

- The c stands for computation of variables. A variable can be used in assignment statements, output statements, and parameters in a function that manipulates and computes them.
- For example, see the following statements that describe the c-use:

  x = y + z; assignment statement

  printf("%d",x); output statements

  sqrt(x); function

  A[i+1] = B [1]; address

Predicates (p-use)

- The p stands for predicates. A variable is also used as a condition in branch statements, such as *if-else*, *switch-case*, etc.
- For example, the following statements illustrate the p-use:

  while(y<=5)

  if(i<x)

Global and local definitions and uses

- The global and local variables are defined, used, and redefined within a block.
- Let us see the following program statements:

a = b + c;

x = a * 2;

a = x + 2;

- Here, the variable ***a*** is defined, used, and redefined. The definition, *a =b + c* is *local* to the block.
- The definition of ***a*** is *killed* or *masked* by the second definition, *a* = *x* + 2, which is now survived in the program. The second definition of **a** is global to use in the subsequent statements.
- The c-uses of variables *b* and *c* are *global* because these variables do not appear in this block before.
- The variable *x* is also *global*. Similarly, the first definition of *x* is *local* and second definition is *global* to the block.

Data flow based testing steps

- The data flow based testing is performed as follows to reach to the design of test cases:

1) *Construct a data flow graph from a program*
2) *Select data flow testing criteria*
3) *Determine feasible paths*
4) *Design test cases*

1) Construct a data flow graph from a program :

- *Compute def, c-use, and p-use for each basic block in the program.*
- *Associate def and c-uses with each node of the graph.*
- *Associate p-uses with each edge of the graph.*
- *The entry node has a definition of each edge parameter and each nonlocal variable used in the program.*
- *The exit node has an undefinition of each local variable.*

Example : A program to accept a number and find the sum of its individual digits repeatedly till the result is a single digit. For example, if the entered digits are 8953 then the program will compute and print as 8953, 25, 7.

```
1.    main()
2.    {      int n, s=0;                              ] 1
3.           scanf("%d", &n);                         ]
4.           for (; n!=0;)   {                          2
5.                  s = s + n%10;                     ] 3
6.                  n = n/10;                         ]
7.                  if (n == 0 && s>9)   {4
8.                         printf("\n%2d", s);            ] 5
9.                         n = s;                         ]
10.                        s = 0;                         ]
11.                 }                                 ] 6
12.          }                                        ]
13.          printf("%2d", s);                        ]
14.   }                                               ]
```

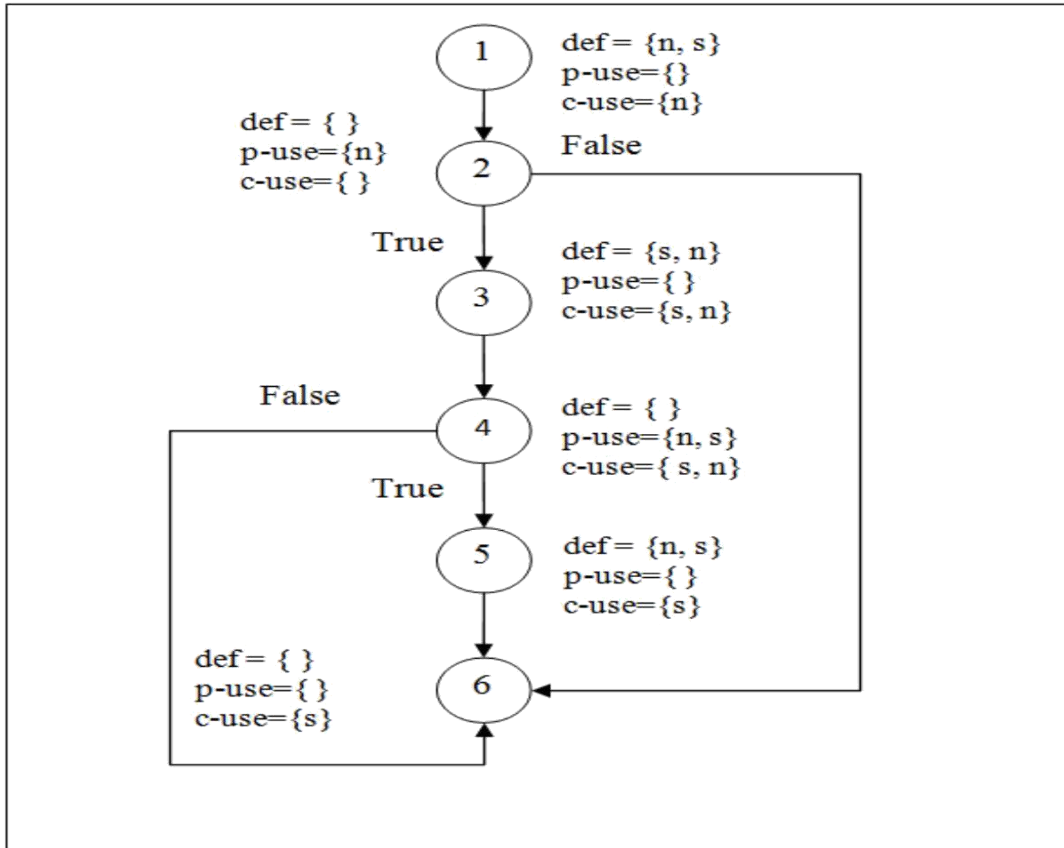


Figure 9.10: Def-use graph for the example 9.6

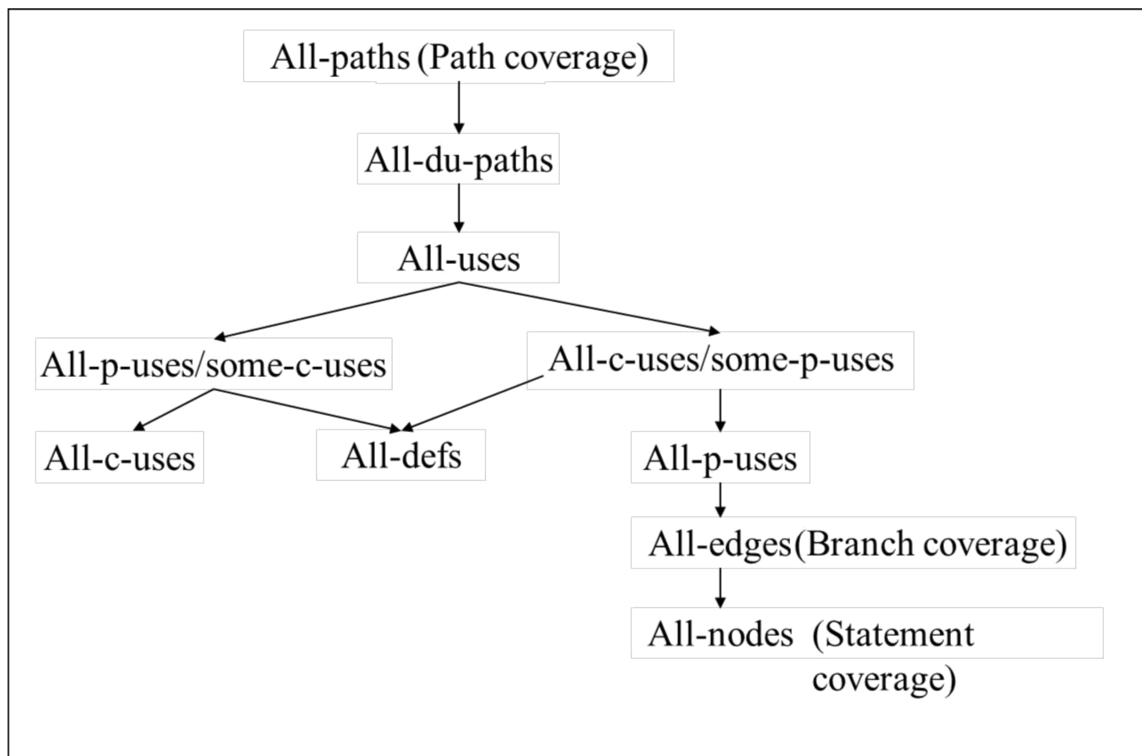
- Def-clear path
  - There can be many paths in a data flow graph. Suppose a variable *x* is defined at node *i* and used at node *j*. The path *p* = (*i*, n_1, ... n_m, *j*), m ≥ 0, (n_1...n_m are nodes) is called a *def-clear path* with respect to variable *x* from node *i* to node *j*, and from node *i* to edge (n_m, *j*), if *x* has been neither defined nor undefined in nodes n_1 – ... n_m.
  - In Figure 9.10, the path 3 -> 4 -> 5 is a def-clear path with respect to variable *s* and *n* because *s* and *n* are not defined in node 4 but these are live at node 5.
- Global definition
  - A node *i* has a global definition of variable *x* if node *i* has a definition of *x* and there is a def-clear path with respect to *x* from node *i* to some node containing a global c-use, or edge containing a p-use of variable *x*.
- Def-use pair
  - There are two types of def-use pairs; *dcu* and *dpu.*
  - The dcu is corresponding to a definition and its c-use and the dpu is corresponding to a definition and its p-use.

- Suppose a variable *x* is in def (*i*) at node *i*, then the dcu (*x*, *i*) is the set of nodes such that each node has *x* in its c-use and there is def-clear path from *i* to *j*.
- In Figure 9.10, the dcu (*s*, 3) has the c-use at node 6 and there is def-clear path 3 -> 4 -> 6. Thus, we get dcu(*s*, 3) = {6}.
- Similarly, dpu (*x*, *i*) is the set of edges such that each edge has *x* in its p-use and there is a def-clear path from *i* to (*j*, *k*).
- In Figure 9.10, the dpu (*s*, 1) has p-use at node 2 and there is a def-clear path from 1 -> 2 -> 6. Thus, we get dpu (*s*, 1) = {(2, 6)}.

2. Select data flow testing criteria

- There are various data flow based testing criteria, such as *all-defs*, *all-c-uses*, *all-p-uses*, *all-p-uses/some-c-uses*, *all-c-uses/some-p-uses*, *all-uses*, *and all-du-paths*.
- It is difficult to say a particular criterion is sufficient.
- Therefore, two criteria are compared. Let c1 and c2 are the two criteria. c2 is said to be included in c1 if for every def-use graph, any set of complete paths of the graph that satisfies c_1 also satisfies c_2.
- Also, it is difficult to trace all the test criteria during data flow testing. It includes relationship -all the criteria are related to each other.

The ordering of different data flow based criterions for test case design is shown in Figure 9.11.



3. Determine feasible paths

- A feasible path should be selected on a criterion that can trace out all the paths on some test data.
- To satisfy the all-def criteria for variables *s* and *n*, there will be the following three paths drawn from the Figure 9.10:

{1 -> 2 -> 3, 3 -> 4 -> 6, 1 -> 2 -> 6}

Table 9.5: Occurrences of variable in def, c-use, and p-use

Line	Definition	c-use	p-use
1			
2	n, s		
3	n		
4			n
5	s	s, n	
6	n	N	
7			n, s
8		S	
9	n	S	
10	s		
11			
12			
13		S	
14			

4. Design test cases

- On the selected data flow based criteria, the test cases are generated to ensure that the criteria are satisfied by it.
- For example, if all-edges criteria are selected then the test case should be designed to cover all edges in the execution of the paths.
- The execution of these three paths for the definitions *s* and *n* with test data (*s*=0, *n*=5), all the lines of the program will be executed at least once.

Mutation Testing

- Mutation testing is another powerful white-box testing that takes a different approach for testing programs, where control flow and data flow based testing are performed by exercising test cases in different paths to find errors.
- *Mutation* is the act of slightly changing a program.
- The changed program is known as *mutated program* and it is called a *mutant* of original program.

- The process of creating mutant is called *mutation*.

- The mutations are tested with test cases of original program to determine whether the test case is capable of detecting the change between original program and its mutants.

- In mutation testing, mutants are created by using mutation operator. A *mutation operator* is an operator whose availability depends on a particular programming language.

- Example 9.7: The following program is written in C language to calculate the sum of numbers and their square values.

```
main()
    {
    int i, sum=0, sqsum=0;
    for (i=1;i<5;i++)
    {
       sum+=i;
       sqsum+=i*i;
       printf("%2d", i,i*i);
     }
     printf("The sum is: %d and square sum is: %d", sum, sqsum);
    }
```

The mutant program is shown below with high order mutants:

```
main(){
        int i, sum=0, sqsum=0;
        for (i=1;i<=5;i++)              Mutated statement 1
        {
           sum+=i;
           sqsum=i*i;                   Mutated statement 2
           printf("%2d", i,i*i);
         }
    printf("The sum is: %d and square sum is: %d", sum, sqsum);
    }
```

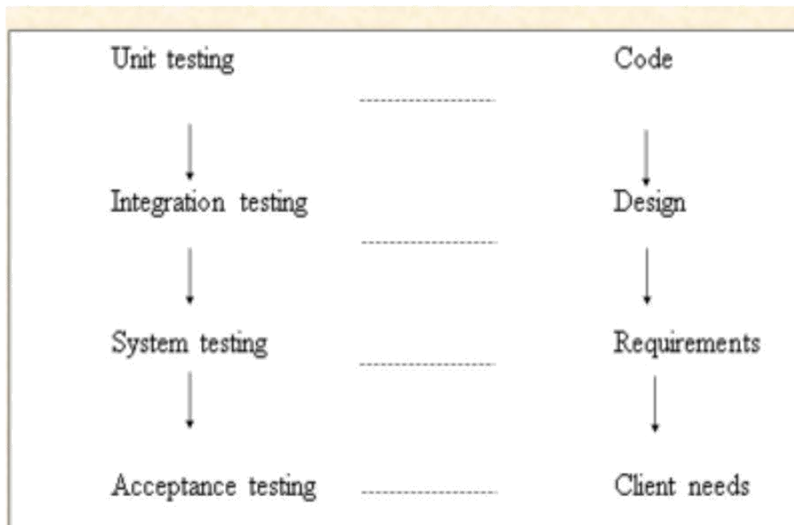
- The mutant can be first order or higher order mutants.
- If the mutant is created by single change in the program then it is known as first order mutant. The higher order mutants are produced by making several changes in a program.
- If a mutant program is distinguished from the parent or original program then it is known as *dead* or *killed mutant*.
- The dead mutants are helpful to detect and fix problems. If a test case does not distinguish the mutant program from original program then it is called *live mutant*.
- In case if the test case is not able to detect and fix problems, then killed mutant and original programs are said to *equivalent.*
- A programmer tries to compute the mutation score after mutation testing.
- Let *T* be a test case, *L* live mutants and *D* the dead mutants and *E* the equivalent mutant and *N* the total mutants generated.
- The mutation score on test case *T*, i.e., *M* (*T*), is computed as follows:

$$M(T) = \frac{|D|}{|N| - |E|}$$

- ✓ Testing is a defect detection technique that is performed at various levels. Testing begins once a module is fully constructed.
- ✓ Although software engineers test source codes after it is written, but it is not an appealing way that can satisfy customer’s needs and expectations.
- ✓ Software is developed through a series of activities, i.e., customer needs, specification, design, and coding.
- ✓ Each of these activities has different aims. Therefore, testing is performed at various levels of development phases to achieve their purpose.

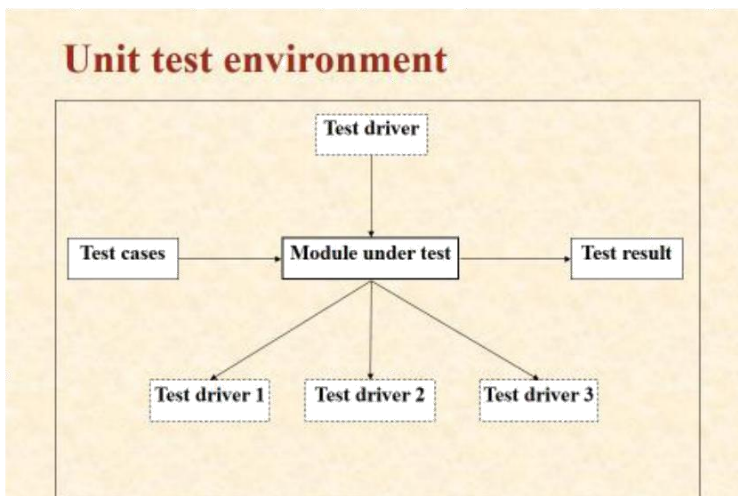
LEVELS OF TESTING:

- ✓ Testing is a defect detection technique that is performed at various levels. Testing begins once a module is fully constructed.
- ✓ Although software engineers test source codes after it is written, but it is not an appealing way that can satisfy customer’s needs and expectations.
- ✓ Software is developed through a series of activities, i.e., customer needs, specification, design, and coding.
- ✓ Each of these activities has different aims. Therefore, testing is performed at various levels of development phases to achieve their purpose.



Unit Testing

- ✓ Unit means a program unit, module, component, procedure, subroutine of a system developed by the programmer.
- ✓ The aim of unit testing is to find bugs by isolating an individual module using test stub and test drivers and by executing test cases on it.
- ✓ During module testing, test stub and test drivers are designed for proper testing in a test environment.
- ✓ The unit testing is performed to detect both structural and functional errors in the module.
- ✓ Therefore, test cases are designed using white-box and black-box testing strategies for unit testing.
- ✓ Most of the module errors are captured through white-box testing.



Integration Testing

- Integration testing is another level of testing, which is performed after unit testing of modules.
- It is carried out keeping in view the design issues of the system into subsystems.
- The main goal of integration testing is to find interface errors between modules.
- There are various approaches in which the modules are combined together for integration testing.
  - *Big-bang approach*
  - *Top-down approach*
  - *Bottom-up approach*
  - *Sandwich approach*

Big-bang approach

- The big-bang is a simple and straightforward integration testing.
- In this approach, all the modules are first tested individually and then these are combined together and tested as a single system.
- This approach works well where there is less number of modules in a system.
- As all modules are integrated to form a whole system, the chaos may occur. If there is any defect found, it becomes difficult to identify where the defect has occurred.
- Therefore, big-bang approach is generally avoided for large and complex systems.

Top-down approach

- Top-down integration testing begins with the main module and move downwards integrating and testing its lower level modules.
- Again the next lower level modules are integrated and tested.
- Thus, this incremental integration and testing is continued until all modules up to the concrete level are integrated and tested.
- The top-down integration testing approach is as follows:
- *main system -> subsystems -> modules at concrete level.*
- In this approach, the testing of a module may be delayed if its lower level modules (i.e., test stubs) are not available at this time.
- Thus, writing test stubs and simulating to act as actual modules may be complicated and time-consuming task.

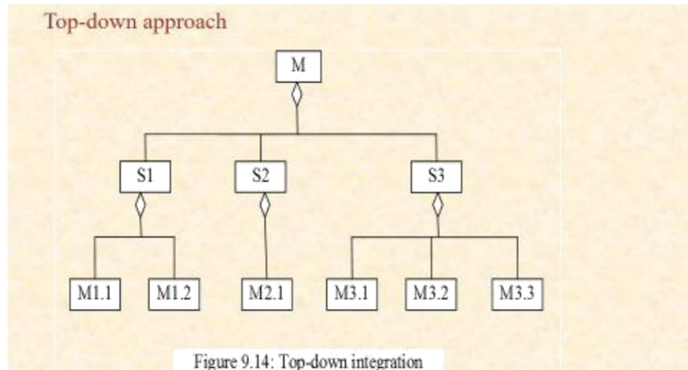


Figure 9.14: Top-down integration

Bottom-up approach

- ✓ As the name implies, bottom-up approach begins with the individual testing of bottom-level modules in the software hierarchy.
- ✓ Then lower level modules are merged function wise together to form a subsystem and then all subsystems are integrated to test the main module covering all modules of the system.
- ✓ The approach of bottom-up integration is as follows:
  - ✓ *concrete level modules -> subsystem –> main module*.
- ✓ The bottom-up approach works opposite to the top-down integration approach.

Sandwich approach

- ✓ The sandwich testing combines both top-down and bottom-up integration approaches.
- ✓ During sandwich testing, top-down approach force to the lower level modules to be available and bottom-up approach requires upper level modules.
- ✓ Thus, testing a module requires its top and bottom level modules.
- ✓ It is the most preferred approach in testing because the modules are tested as and when these are available for testing.

System Testing

- ✓ The unit and integration testing are applied to detect defects in the modules and the system as a whole. Once all the modules have been tested, system testing is performed to check whether the system satisfies the requirements (both functional and non-functional).
- ✓ To test the functional requirements of the system, functional or black-box testing methods are used with appropriate test cases.
- ✓ System testing is performed keeping in view the system requirements and system objectives.
- ✓ The non-functional requirements are tested with a series of tests whose purpose is to check the computer-based system.
- ✓ A single test case cannot ensure all the system non-functional requirements.
- ✓ For specific non-functional requirements, special tests are conducted to ensure the system functionality.

- Some of the non-functional system tests are
  - Performance testing
  - Volume testing
  - Stress testing
  - Security testing
  - Recovery testing
  - Compatibility testing
  - Configuration testing
  - Installation testing
  - Documentation testing

Performance testing

- A performance testing is carried out to check the run time outcomes of the system, such as efficiency, accuracy, etc.
- Each system performs differently in different environment.
- During performance testing, both hardware and software are taken into consideration to observe the system behavior.
- For example, a testing tool is tested to check if it tests the specified codes in a defined time duration.

Volume testing

- It deals with the system if heavy amount of data are to be processed or stored in the system.
- For example, an operating system will be checked to ensure that the job queue will handle when a large number of processes enter into the computer.
- It basically checks the capacity of the data structures.

Stress testing

- In stress testing, the behavior of the system is checked when it is under stress.
- The stress may come due to load increases at peak time for a short period of time.
- There are several reasons of stress, such as the maximum number of users increased, peak demand, number of operations extended, etc.
- For example, the network server is checked if the number of concurrent users and nodes are increased to use the network resources in the evening time.
- The stress test is the peak time of the volume testing.

Security testing

- Due to the increasing complexity of software and its applications for variety of users in different technologies, it becomes necessary to provide sufficient security to the society.
- It is conducted to ensure the security checks at different levels in the system.
- For example, testing of e-payment system is done to ensure that the money transaction is happening in a secure manner in e-commerce applications.
- There a lot of confidential data are transferred and used in the system that must be protected from its leakage, alteration, and modification by illegal people.

Recovery testing

- Most of the systems now have the recovery policies if the there is any loss of data.
- Therefore, recovery testing is performed to check that it will recover the losses caused by data error, software error, or hardware problems.
- For example, the Windows operating system recovers the currently running files if any hardware/software problem occurs in the system.

Compatibility testing

- Compatibility testing is performed to ensure that the new system will be able to work with the existing system.
- Sometimes, the data format, report format, process categories, databases, etc., differ from system to system.
- For example, compatibility testing checks whether Windows 2007 files can be opened in the Windows 2003 if it is installed in the system.

Configuration testing

- Configuration testing is performed to check that a system can run on different hardware and software configurations.
- Therefore, system is configured for each of the hardware and software.
- For example, suppose you want to run your program on other machine then you are required to check the configuration of its hardware and software.

Documentation testing

- Once the system becomes operational, problems may be encountered in the system.
- A systematic documentation or manual can help to recover such problems.
- The system is verified whether its proper documentation is available.

Installation testing

- Installation testing is conducted to ensure that all modules of the software are installed properly.

- ✓ The main purpose of installation testing is to find errors that occur during the installation process.
- ✓ Installation testing covers various issues, such as automatic execution of the CD, files and libraries must be allocated and loaded; appropriate hardware configurations must be present; proper network connectivity; compatible with the operating system platform, etc.
- ✓ The installers must be familiar with the installation technologies and their troubleshooting mechanisms.

Acceptance Testing

- ✓ Acceptance testing is a kind of system testing, which is performed before the system is released into the market.
- ✓ It is performed with the customer to ensure that the system is acceptable for delivery.
- ✓ Once all system testing have been exercised, the system is now tested from the customer's point of view.
- ✓ Acceptance testing is conducted because there is a difference between the actual user and the simulated users considered by the development organization.
- ✓ The user involvement is important during acceptance testing of the software as it is developed for the end-users.
- ✓ Acceptance testing is performed at two levels, i.e.,
    - ✓ *Alpha testing*
    - ✓ *Beta testing*.
- ✓ *Alpha testing* is a pilot testing in which customers are involved in exercising test cases.
    - ✓ In alpha testing, customer conducts tests in the development environment. The users perform alpha test and tries to pinpoint any problem in the system.
    - ✓ The alpha test is conducted in a controlled environment.
    - ✓ After alpha testing, system is ready to transport the system at the customer site for deployment
- ✓ *Beta testing* is performed by a limited and friendly customers and end-users.
- ✓ Beta testing is conducted at the customer site, where the software is to be deployed and used by the end-users.
    - ✓ The developer may or may not be present during beta testing.
    - ✓ The end-users operate the system under testing mode and note down any problem observed during system operation.
    - ✓ The defects noted by the end-users are corrected by the developer.
    - ✓ If there are any major changes required, then these changes are sent to the configuration management team.

- ✓ The configuration management team decides whether to approve or disapprove the changes for modification in the system.

- ✓ Shadow testing
- ✓ Benchmark testing.

Shadow testing

- ✓ Shadow testing is conducted in case of maintenance or reengineering type of projects.
- ✓ In this testing, the new system and the legacy system are run side-by-side and their results are compared.
- ✓ Any unusual results noted by end-user are informed to the developers that they take corrective actions to remove the problems.

Benchmark testing

- ✓ In benchmark test, client prepares test cases to test the system performance. The benchmark test is conducted either by end-users or testers.
- ✓ Before performing benchmark test, tester or end-users must be familiar with the functional and nonfunctional requirements of the system.
- ✓ Benchmark testing helps to assess product's performance against other products in a number of areas including functionality, durability, quality, etc.

USABILITY TESTING

- ✓ Usability refers to the ease of use and comfort that users have while working with software.
- ✓ It is also known as *user-centric testing*. Nowadays, usability has become a wider aspect of software development and testing.
- ✓ Usability testing is conducted to check usability of the system which mainly focuses on finding the differences between quality of developed software and user's expectations of what it should perform.
- ✓ Poor usability may affect the success of the software. If the user finds that the system is difficult to understand and operate, then ultimately it will lead to unsuccessful product.
- ✓ The usability testing concentrates on the testing of user interface design, such as look and feel of the user interface, format of reports, screen layouts, hardware and user interactions, etc.
- ✓ Usability testing is performed by potential end-users in a controlled environment.
- ✓ The development organization calls selected end-users to test the product in terms of ease of use, functionality as expected, performance, safety and security; and the outcomes.

REGRESSION TESTING

- ✓ Regression testing is also known as program revalidation.
- ✓ Regression testing is performed whenever new functionality is added or the existing functionality is modified in the program.
- ✓ If the existing system is working correctly, then new system should work correctly after making changes because the code may have been changed.
- ✓ It is required when the new version of a program is obtained by changing the existing version.
- ✓ Regression testing is also needed when a subsystem is modified to get the new version of the system.

SMOKE TESTING

- ✓ Smoke testing is also sometimes known as *sanity testing*.
- ✓ In smoke testing, software module is tested to verify "build" activity in an informal and non-exhaustive manner.
- ✓ It is done to check that major functionalities of the system are working properly before performing detailed black-box and white-box testing.
- ✓ It should be able to expose errors in the system. The concept of smoke testing is taken from the hardware devices in which a new hardware device is attached to the system first time and it is assumed to be successful if it does not start smoking.

DEBUGGING

- ✓ Debugging is a post-testing mechanism of locating and fixing errors.
- ✓ A successful test case aims to prove that the program is incorrect. Such kind of behavior of the program is observed by symptoms or known errors.
- ✓ Once errors are reported by testing methods, these are isolated and removed during debugging. Debugging has two important steps:
    - *Identifying the location and nature of errors.*
    - *Correcting or fixing errors.*
- The identification of location and nature of an error is an important task in debugging.
- The error correction is done after errors have been identified. There are various methods of finding and correcting errors in a program.

DEBUGGING APPROACHES

- The most popular debugging approaches are as follows:
  - *Brute force*
  - *Backtracking*
  - *Breakpoint*
  - *Debugging by induction*
  - *Debugging by deduction*
  - *Debugging by testing*

Brute force

- ✓ It is the simplest method of debugging but it is inefficient. It uses memory dumps or output statements for debugging.
- ✓ The memory dump is a machine level representation of the corresponding variables and statements.
- ✓ It represents the static structure of the program at a particular snapshot of execution sequence.
- ✓ The memory dump rarely establishes correspondence to show errors at a particular time.
- ✓ Also, one should have good understanding of dynamics of the program.
- ✓ Therefore, instead of using brute force for debugging, a debugger should be used for better results.

Backtracking

- ✓ Backtracking is the refinement of brute force method and it is one of the successful methods of debugging.
- ✓ Debugging begins from where the bug is discovered and the source code is traced out backward though different paths until the exact location of the cause of bug is reached or the cause of bug has disappeared.
- ✓ This process is performed with the program logic in reverse direction of the flow of control.
- ✓ This method is effective for small size problems.
- ✓ Backtracking should be used when all other methods of debugging are not able to locate errors. The reason is the effort spent in backtracking if there is no error exists in the source code.

Breakpoint

- Breakpoint debugging is a method of tracing programs with a breakpoint and stopping the program execution at the breakpoint.
- A breakpoint is a kind of signal that tells the debugger to temporarily suspend execution of program at a certain point.
- Each breakpoint is associated with a particular instruction of the program.
- The program execution continues before the breakpoint statement. If any error is reported, its location is marked and then the program execution resumes till the next breakpoint.
- This process is continued until all errors are located in the program.
- Breakpoint is also performed with watch values.
- A watch value is a value of a variable or expression, which is set and shown along with the program execution.
- The watch values change as the program executes.
- The incorrect or unexpected values can be observed with the watch values.

Debugging by induction:

- It is based on pattern matching and a thought process on some clue.
- The process begins from collecting information about pertinent data where the bug has discovered.
- The patterns of successful test cases are observed and data items are organized.
- Thereafter, hypothesis is derived by relating the pattern and the error to be debugged.
- On successful hypothesis, the devised theory is proved the occurrence of bugs.
- Otherwise, more data are collected to derive causes of errors. Finally, causes are removed and errors are fixed in the program.

Debugging by deduction

- This is the kind of cause elimination method.
- On the basis of cause hypothesis, lists of possible causes are enumerated for the observed failure.
- Now the tests are conducted to eliminate causes to remove errors in the system.
- If all the causes are eliminated then errors are fixed. Otherwise, hypothesis is refined to eliminate errors.

- ✓ Finally, hypothesis is proved to ensure that all causes have been eliminated and the system is bug free.

Debugging by testing

- ✓ It uses test cases to locate errors.
- ✓ Test cases designed during testing are used in debugging to collect information to locate the suspected errors.
- ✓ Test case in testing focuses on covering many conditions and statements, whereas test case in debugging focuses on small number of conditions and statements.
- ✓ Test cases of debugging are the refined test cases of testing.
- ✓ Test cases during debugging concentrate on locating error situations in a program. Conclusion:

- ✓ Software testing is one of the important phases of software life cycle that aims to make the program error-free and ensures the product quality.
- ✓ The cost of testing is generally higher than the cost of remaining activities in the software development life cycle.
- ✓ There are two types of test cases design strategies, i.e., black-box and white-box testing.
- ✓ Testing is performed at various level, i.e., unit testing, integration testing, system testing, and acceptance testing.
- ✓ System testing is performed to check whether the system satisfies the requirements (both functional and non-functional).