

Unit- 4 Coding and Testing

Introduction:

- Coding is undertaken once the design phase is complete and the design documents have been successfully reviewed.
- In the coding phase, every module specified in the design document is coded and unit tested. During unit testing, each module is tested in isolation from other modules.
- After all the modules of a system have been coded and unit tested, the integration and system testing phase is undertaken
- Integration and testing of modules is carried out according to an integration plan.
- The full product takes shape only after all the modules have been integrated together. System testing is conducted on the full product. During system testing, the product is tested against its requirements as recorded in the SRS document.
- Testing is an important phase in software development, requires the maximum effort and requires the maximum effort.

Coding:

- The input to the coding phase is the design document produced at the end of the design phase.
- The design document contains not only the high-level design of the system in the form of a module structure (e.g., a structure chart), but also the detailed design.
- The detailed design is usually documented in the form of module specifications where the data structures and algorithms for each module are specified.
- The objective of the coding phase is to transform the design of a system into code in a high-level language, and then to unit test this code.
- Good software development organisations require their programmers to adhere to some well-defined and standard style of coding which is called their coding standard.
- Organisations formulate their own coding standards and require their developers to follow the standards rigorously.
- The main advantages of adhering to a standard:
 - A coding standard gives a uniform appearance to the codes written by different engineers.
 - It facilitates code understanding and code reuse.
 - It promotes good programming practices

What is the difference between a coding guideline and a coding standard?

- It is mandatory for the programmers to follow the coding standards. Compliance of their code to coding standards is verified during code inspection. Any code that does not conform to the coding standards is rejected during code review and the code is reworked by the concerned programmer.
- In contrast, coding guidelines provide some general suggestions regarding the coding style to be followed but leave the actual implementation of these guidelines to the discretion of the individual developers. Usually code review is carried out to ensure that the coding standards are followed and also to detect as many errors as possible before testing. Reviews are an efficient way of removing errors from code

Coding Standards and Guidelines:

Good software development organisations usually develop their own coding standards and guidelines.

Representative coding standards:

- **Rules for limiting the use of globals:** These rules list what types of data can be declared global and what cannot, with a view to limit the data that needs to be defined with global scope.
- **Standard headers for different modules:** The header of different modules should have standard format and information for ease of understanding and maintenance.
- **Naming conventions for global variables, local variables, and constant identifiers:**
A popular naming convention is that variables are named using mixed case lettering. Example :GlobalData, localData, CONSTDATA
- **Conventions regarding error return values and exception handling mechanisms:**
The way error conditions are reported by different functions in a program should be standard within an organisation.

Representative Coding Guidelines:

- **Do not use a coding style that is too clever or too difficult to understand:** Code should be easy to understand. Many inexperienced engineers actually take pride in writing cryptic and incomprehensible code.

- **Avoid obscure side effects:** The side effects of a function call include modifications to the parameters passed by reference, modification of global variables, and I/O operations. An obscure side effect is one that is not obvious from a casual examination of the code. Obscure side effects make it difficult to understand a piece of code.
- **Do not use an identifier for multiple purposes**Programmers often use the same identifier to denote several temporary entities. There are several things wrong with this approach and hence should be avoided.
- **Code should be well-documented:** As a rule of thumb, there should be at least one comment line on the average for every three source lines of code.
- Length of any function should not exceed 10 source lines: A lengthy function is usually very difficult to understand as it probably has a large number of variables and carries out many different types of computations.
- Do not use GOTO statements: Use of GOTO statements makes a program unstructured. This makes the program very difficult to understand, debug, and maintain.

Code Review:

- Testing is an effective defect removal mechanism. However, testing is applicable to only executable code.
- Review is a very effective technique to remove defects from source code. In fact, review has been acknowledged to be more cost-effective in removing defects as compared to testing.
- Code review for a module is undertaken after the module successfully compiles. That is, all the syntax errors have been eliminated from the module.
- Code review does not target to design syntax errors in a program, but is designed to detect logical, algorithmic, and programming errors.
- Code review has been recognised as an extremely cost-effective strategy for eliminating coding errors and for producing high quality code.
- Reviews directly detect errors, whereas testing only helps detect failures.
- Eliminating an error from code involves three main activities—testing, debugging, and then correcting the errors. Testing is carried out to detect if the system fails to work satisfactorily for certain types of inputs and under certain circumstances. Once a failure

is detected, debugging is carried out to locate the error that is causing the failure and to remove it. Of the three testing activities, debugging is possibly the most laborious and time consuming activity.

- In code inspection, errors are directly detected, thereby saving the significant effort that would have been required to locate the error. Normally, the following two types of reviews are carried out on the code:
 - Code Inspection
 - Code Walkthrough

Code inspection:-

- During code inspection, the code is examined for the presence of some common programming errors.
- The principal aim of code inspection is to check for the presence of some common types of errors that usually creep into code due to programmer mistakes and oversights and to check whether coding standards have been adhered to.
- The inspection process has several beneficial side effects, other than finding errors. The programmer usually receives feedback on programming style, choice of algorithm, and programming techniques. The other participants gain by being exposed to another programmer's errors.
- Good software development companies collect statistics regarding different types of errors that are commonly committed by their engineers and identify the types of errors most frequently committed.
- Such a list of commonly committed errors can be used as a checklist during code inspection to look out for possible errors.
- Following is a list of some classical programming errors which can be checked during code inspection:
 - ✓ Use of uninitialised variables.
 - ✓ Jumps into loops.
 - ✓ Non-terminating loops.
 - ✓ Incompatible assignments.
 - ✓ Array indices out of bounds.
 - ✓ Improper storage allocation and deallocation.

- ✓ Mismatch between actual and formal parameters in procedure calls.
- ✓ Use of incorrect logical operators or incorrect precedence among operators.
- ✓ Improper modification of loop variables. ○ Comparison of equality of floating point values.
- ✓ Dangling reference caused when the referenced memory has not been allocated.

Code walkthrough:-

- Code walkthrough is an informal code analysis technique.
- In this technique, a module is taken up for review after the module has been coded, successfully compiled, and all syntax errors have been eliminated.
- A few members of the development team are given the code a couple of days before the walkthrough meeting.
- Each member selects some test cases and simulates execution of the code by hand.
- The main objective of code walkthrough is to discover the algorithmic and logical errors in the code.
- Even though code walkthrough is an informal analysis technique, several guidelines have evolved over the years. Guidelines are based on personal experience, common sense, and several other subjective factors.
 - The team performing code walkthrough should not be either too big or too small. Ideally, it should consist of between three to seven members.
 - Discussions should focus on discovery of errors and avoid deliberations on how to fix the discovered errors.
 - In order to foster cooperation and to avoid the feeling among the engineers that they are being watched and evaluated in the code walkthrough meetings, managers should not attend the walkthrough meetings.

Software Documentation:

When a software is developed, in addition to the executable files and the source code, several kinds of documents such as users' manual, software requirements specification (SRS) document, design document, test document, installation manual, etc., are developed as part of the software engineering process.

All these documents are considered a vital part of any good software development practice.

Good documents are helpful in the following ways:

- Good documents help enhance understandability of code.
- Documents help the users to understand and effectively use the system.
- Good documents help to effectively tackle the manpower turnover problem
- Production of good documents helps the manager to effectively track the progress of the project

Different types of software documents can broadly be classified into the following:

Internal documentation:

- These are provided in the source code itself. Internal documentation can be provided in the code in several forms. The important types of internal documentation are the following:
 - Comments embedded in the source code.
 - Use of meaningful variable names.
 - Module and function headers.
 - Code indentation.
 - Code structuring (i.e., code decomposed into modules and functions).
 - Use of enumerated types.
 - Use of constant identifiers.
 - Use of user-defined data types.
- Even when a piece of code is carefully commented, meaningful variable names have been found to be the most helpful in understanding the code

External documentation:

- These are the supporting documents such as SRS document, installation document, user manual, design document, and test document.
- A systematic software development style ensures that all these documents are of good quality and are produced in an orderly fashion.
- An important feature that is required of any good external documentation is consistency with the code.
- If the different documents are not consistent, a lot of confusion is created for somebody trying to understand the software.

- Every change made to the code should be reflected in the relevant external documents.
- Another important feature required for external documents is proper understandability by the category of users for whom the document is designed

Testing:-

- The aim of program testing is to help realise/identify all defects in a program.
- However, in practice, even after satisfactory completion of the testing phase, it is not possible to guarantee that a program is error free.
- This is because the input data domain of most programs is very large, and it is not practical to test the program exhaustively with respect to each value that the input can assume.
- We must remember that careful testing can expose a large percentage of the defects existing in a program

Testing terminology:-

Mistake: A mistake is essentially any programmer action that later shows up as an incorrect result during program execution. A programmer may commit a mistake in almost any development activity.

Error: An error is the result of a mistake committed by a developer in any of the development activities. Among the extremely large variety of errors that can exist in a program. The terms error, fault, bug, and defect are considered to be synonyms.

Failure: A failure of a program essentially denotes an incorrect behaviour exhibited by the program during its execution. An incorrect behaviour is observed either as an incorrect result produced or as an inappropriate activity carried out by the program.

Test-case: A test case is a triplet [I , S, R], where I is the data input to the program under test, S is the state of the program at which the data is to be input, and R is the result expected to be produced by the program. The state of a program is also called its execution mode.

- A positive test case is designed to test whether the software correctly performs a required functionality
- A negative test case is designed to test whether the software carries out something that is not required of the system.

Test scenario: A test scenario is an abstract test case in the sense that it only identifies the aspects of the program that are to be tested without identifying the input, state, or output. A test case can be said to be an implementation of a test scenario.

Test script: A test script is an encoding of a test case as a short program. Test scripts are developed for automated execution of the test cases.

Test suite: A test suite is the set of all tests that have been designed by a tester to test a given program.

Testability: Testability of a requirement denotes the extent to which it is possible to determine whether an implementation of the requirement conforms to it in both functionality and performance.

Failure mode: A failure mode of a software denotes an observable way in which it can fail.

Equivalent faults: Equivalent faults denote two or more bugs that result in the system failing in the same failure mode

Validation vs Verification:-

- The objectives of both verification and validation techniques are very similar since both these techniques are designed to help remove errors in a software.
- The underlying principles of these two bug detection techniques and their applicability are very different.

Verification:-

- Verification is the process of determining whether the output of one phase of software development conforms to that of its previous phase;
- Verification is to check if the work products produced after a phase conform to that which was input to the phase.
- Techniques used for verification include review, simulation, formal verification, and testing.

Validation:-

- Validation is the process of determining whether a fully developed software conforms to its requirements specification

- Validation is applied to the fully developed and integrated software to check if it satisfies the customer's requirements.
- System testing can be considered as a validation step where it is determined whether the fully developed code is as per its requirements specification.

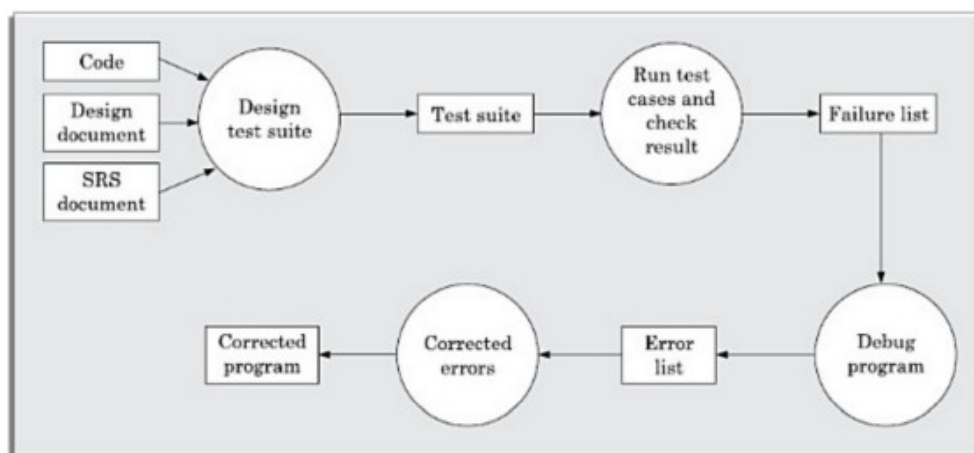
Error detection techniques = Verification techniques + Validation techniques

How to test a Program: -

- Testing a program involves executing the program with a set of test inputs and observing if the program behaves as expected.
- If the program fails to behave as expected, then the input data and the conditions under which it fails are noted for later debugging and error correction.
- Unless the conditions under which a software fails are noted down, it becomes difficult for the developers to reproduce a failure observed by the testers.

Testing Activities:-

- **Test suite design:** The set of test cases using which a program is to be tested is designed possibly using several test case design techniques.
- **Running test cases and checking the results to detect failures:** Each test case is run and the results are compared with the expected results. A mismatch between the actual result and expected results indicates a failure.
- **Locate error:** In this activity, the failure symptoms are analysed to locate the errors.
- **Error correction:** After the error is located during debugging, the code is appropriately changed to correct the error



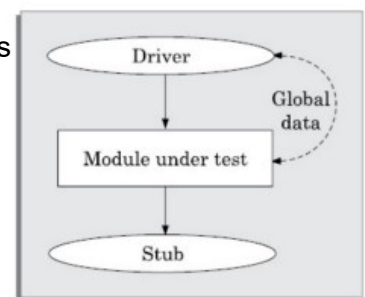
Unit Testing:-

- Unit testing is undertaken after a module has been coded and reviewed.
- This activity is typically undertaken by the coder of the module himself in the coding phase.
- Before carrying out unit testing, the unit test cases have to be designed and the test environment for the unit under test has to be developed.
- In order to test a single module, we need a complete environment to provide all relevant code that is necessary for execution of the module.
- That is, besides the module under test, the following are needed to test the module:
 - The procedures belonging to other modules that the module under test calls.
 - Non-local data structures that the module accesses.
 - A procedure to call the functions of the module under test with appropriate parameters.
- Modules required to provide the necessary environment (which either call or are called by the module under test) are usually not available until they too have been unit tested.
- In this context, stubs and drivers are designed to provide the complete environment for a module so that testing can be carried out.

Stub: A stub procedure is a dummy procedure that has the same I/O parameters as the function called by the unit under test but has a highly simplified behaviour.

Driver: A driver module should contain the non-local data structures accessed by the module under test. Additionally, it should also have the code to call the different functions of the unit under test with appropriate parameter values for testing.

- Unit testing is referred to as testing in the small, whereas integration and system testing are referred to as testing in the large



Black-Box testing: -

- In black-box testing, test cases are designed from an examination of the input/output values only and no knowledge of design or code is required. This method of test can be applied virtually to every level of software testing: unit, integration, system and acceptance. It is sometimes referred to as specification-based testing.



- The following are the two main approaches available to design black box test cases:
 - Equivalence class partitioning
 - Boundary value analysis

Equivalence class partitioning:

- In the equivalence class partitioning approach, the domain of input values to the program under test is partitioned into a set of equivalence classes.
- The partitioning is done such that for every input data belonging to the same equivalence class, the program behaves similarly.
- The main idea behind defining equivalence classes of input data is that testing the code with any one value belonging to an equivalence class is as good as testing the code with any other value belonging to the same equivalence class.
- Equivalence classes for a unit under test can be designed by examining the input data and output data.
- The following are two general guidelines for designing the equivalence classes:
 1. If the input data values to a system can be specified by a range of values, then one valid and two invalid equivalence classes need to be defined.

For example, if the equivalence class is the set of integers in the range 1 to 10 (i.e., $[1,10]$), then the invalid equivalence classes are $[-\infty,0]$, $[11,+\infty]$.
 2. If the input data assumes values from a set of discrete members of some domain, then one equivalence class for the valid input values and another equivalence class for the invalid input values should be defined.

For example, if the valid equivalence classes are {A,B,C}, then the invalid equivalence class is $U - \{A,B,C\}$ where U is the universe of possible input values.

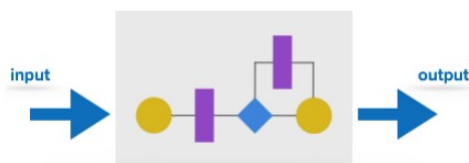
Boundary Value Analysis:

- A type of programming error that is frequently committed by programmers is missing out on the special consideration that should be given to the values at the boundaries of different equivalence classes of inputs.
- Boundary value analysis-based test suite design involves designing test cases using the values at the boundaries of different equivalence classes.
- To design boundary value test cases, it is required to examine the equivalence classes to check if any of the equivalence classes contains a range of values. For those equivalence classes that are not a range of values no boundary value test cases can be defined.
- For an equivalence class that is a range of values, the boundary values need to be included in the test suite. For example, if an equivalence class contains the integers in the range 1 to 10, then the boundary value test suite is {0,1,10,11}.

White Box Testing:-

WHITE BOX TESTING (also known as Clear Box Testing, Open Box Testing, Glass Box Testing, Transparent Box Testing, Code-Based Testing or Structural Testing) is a software testing method in which the internal structure/design/implementation of the item being tested is known to the tester. Our goal is to ensure that all statements and conditions have been executed atleast once.

It is the testing based on an analysis of the internal structure of the component or system.



- White-box testing is an important type of unit testing. A large number of white-box testing strategies exist.

- Each testing strategy essentially designs test cases based on analysis of some aspect of source code and is based on some heuristic.

Basic Concepts: A white-box testing strategy can either be coverage-based or fault based.

Fault-based testing: A fault-based testing strategy targets to detect certain types of faults.

These faults that a test strategy focuses on constitutes the fault model of the strategy..

Coverage-based testing: A coverage-based testing strategy attempts to execute (or cover) certain elements of a program

Using white-box testing methods, the software engineer can derive test cases that

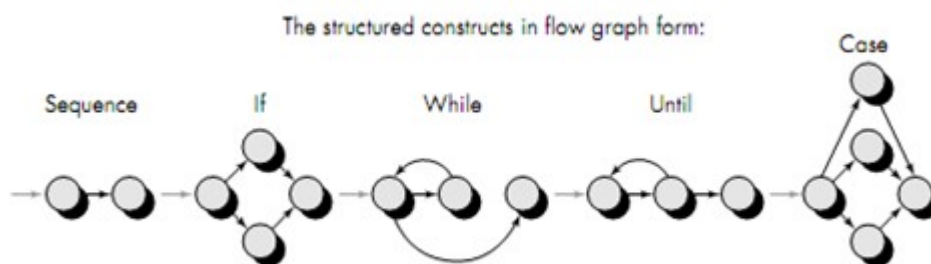
- 1) Guarantee that all independent paths within a module have been exercised at least once
- 2) Exercise all logical decisions on their true and false sides,
- 3) Execute all loops at their boundaries and within their operational bounds
- 4) Exercise internal data structures to ensure their validity.

Basis Path Testing

Basis path testing is a white-box testing technique first proposed by Tom McCabe. Test cases derived to exercise the basis set are guaranteed to execute every statement in the program at least one time during testing.

Flow Graph Notation

Before the basis path method can be introduced, a simple notation for the representation of control flow, called a flow graph (or program graph) must be introduced.



Each circle, called a flow graph node, represents one or more procedural statements. The arrows on the flow graph, called edges or links, represent flow of control

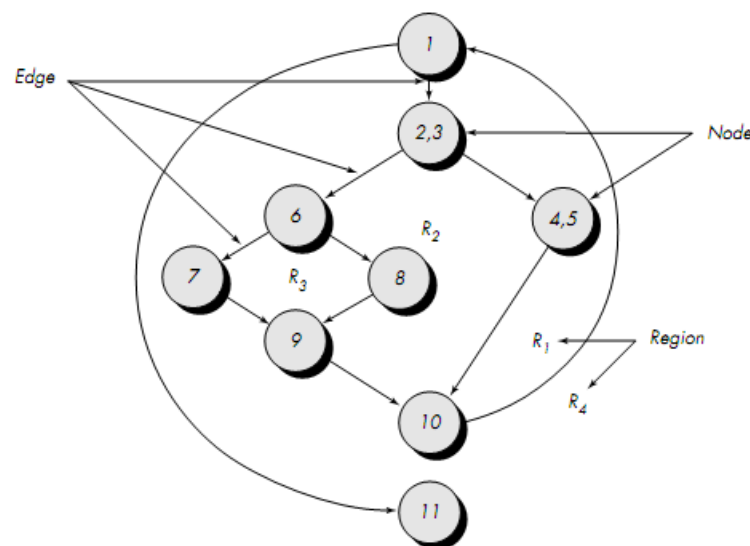
Each node that contains a condition is called a **predicate node** and is characterized by two or more edges emanating from it.

Cyclomatic Complexity

Cyclomatic complexity is software metric that provides a quantitative measure of the logical complexity of a program.

In the context of the basis path testing method, the value computed for Cyclomatic complexity defines the number of independent paths in the basis set of a program and provides us with an upper bound for the number of tests that must be conducted to ensure that all statements have been executed at least once.

An independent path is any path through the program that introduces at least one new set of processing statements or a new condition. When stated in terms of a **flow graph**, an independent path must move along at least one edge that has not been traversed before the path is defined.



In the above fig the set of independent paths are as follows:

Path 1: 1-11

Path 2: 1-2-3-4-5-10-1-11

Path 3: 1-2-3-6-8-9-10-1-11

Path 4: 1-2-3-6-7-9-10-1-11

Note that each new path introduces a new edge. The path

1-2-3-4-5-10-1-2-3-6-8-9-10-1-11

Is not considered an independent path because it is simply a combination of already specified paths and does not traverse any new edges.

Paths 1, 2, 3, and 4 constitute a basis set for the above flow graph

How is Cyclomatic complexity computed?

1. The number of regions of the flow graph corresponds to the Cyclomatic complexity.

2. Cyclomatic complexity, $V(G)$, for a flow graph, G , is defined as $V(G) = E - N + 2$

Where E is the number of flow graph edges, N is the number of flow graph nodes.

3. Cyclomatic complexity, $V(G)$, for a flow graph, G , is also defined as $V(G) = P + 1$

where P is the number of predicate nodes contained in the flow graph G .

Referring once more to the above flow graph the Cyclomatic complexity can be computed using each of the algorithms just noted:

1. The flow graph has four regions.

2. $V(G) = 11 \text{ edges} - 9 \text{ nodes} + 2 = 4$.

3. $V(G) = 3 \text{ predicate nodes} + 1 = 4$. cyclomatic complexity for above flow graph

Debugging:

After a failure has been detected, it is necessary to first identify the program statement(s) that are in error and are responsible for the failure, the error can then be fixed.

Debugging Approaches: The following are some of the approaches that are popularly adopted by the programmers for debugging:

1. Brute force method:

- This is the most common method of debugging but is the least efficient method.
- In this approach, print statements are inserted throughout the program to print the intermediate values with the hope that some of the printed values will help to identify the statement in error.
- This approach becomes more systematic with the use of a symbolic debugger, because values of different variables can be easily checked and breakpoints and watchpoints can be easily set to test the values of variables effortlessly.

2. Backtracking:

- This is also a fairly common approach. In this approach, starting from the statement at which an error symptom has been observed, the source code is traced backwards until the error is discovered.

3. Cause elimination method:

- In this approach, once a failure is observed, the symptoms of the failure are noted.
- Based on the failure symptoms, the causes which could possibly have contributed to the symptom is developed and tests are conducted to eliminate each.

4. Program slicing:

- This technique is similar to back tracking. In the backtracking approach, one often has to examine a large number of statements.
- However, the search space is reduced by defining slices.
- A slice of a program for a particular variable and at a particular statement is the set of source lines preceding this statement that can influence the value of that variable.

Debugging guidelines:

Debugging is often carried out by programmers based on their ingenuity and experience. The following are some general guidelines for effective debugging:

- Many times debugging requires a thorough understanding of the program design. Trying to debug based on a partial understanding of the program design may require an inordinate amount of effort to be put into debugging even for simple problems.
- Debugging may sometimes even require full redesign of the system. In such cases, a common mistake that novice programmers often make is attempting not to fix the error but its symptoms.
- One must beware of the possibility that an error correction may introduce new errors. Therefore after every round of error-fixing, regression testing must be carried out.

Integration Testing:-

- Integration testing is carried out after all (or at least some of) the modules have been unit tested.
- Successful completion of unit testing, to a large extent, ensures that the unit (or module) as a whole works satisfactorily.
- In this context, the objective of integration testing is to detect the errors at the module interfaces (call parameters).
- The objective of integration testing is to check whether the different modules of a program interface with each other properly.
- During integration testing, different modules of a system are integrated in a planned manner using an integration plan.
- The integration plan specifies the steps and the order in which modules are combined to realise the full system.
- After each integration step, the partially integrated system is tested.
- By examining the structure chart, the integration plan can be developed.
- Any one (or a mixture) of the following approaches can be used to develop the test plan:

1. Big-bang approach to integration testing:-

- Big-bang testing is the most obvious approach to integration testing. In this approach, all the modules making up a system are integrated in a single step.
- In simple words, all the unit tested modules of the system are simply linked together and tested.
- However, this technique can meaningfully be used only for very small systems.
- The main problem with this approach is that once a failure has been detected during integration testing, it is very difficult to localise the error as the error may potentially lie in any of the modules.

2. Bottom-up approach to integration testing:-

- Large software products are often made up of several subsystems.
- A subsystem might consist of many modules which communicate among each other through well-defined interfaces.
- In bottom-up integration testing, first the modules for each subsystem are integrated.
- Thus, the subsystems can be integrated separately and independently.
- The primary purpose of carrying out the integration testing of a subsystem is to test whether the interfaces among various modules making up the subsystem work satisfactorily.
- In a pure bottom-up testing no stubs are required, and only test-drivers are required.

3. Top-down approach to integration testing:-

- Top-down integration testing starts with the root module in the structure chart and one or two subordinate modules of the root module.
- After the top-level 'skeleton' has been tested, the modules that are at the immediately lower layer of the 'skeleton' are combined with it and tested.
- Top-down integration testing approach requires the use of program stubs to simulate the effect of lower-level routines that are called by the routines under test.
- A pure top-down integration does not require any driver routines.

4. Mixed approach to integration testing:-

- The mixed (also called sandwiched) integration testing follows a combination of top-down and bottom-up testing approaches.
- In a top-down approach, testing can start only after the top-level modules have been coded and unit tested.
- Similarly, bottom-up testing can start only after the bottom level modules are ready.
- The mixed approach overcomes this shortcoming of the top-down and bottom-up approaches.
- In the mixed testing approach, testing can start as and when modules become available after unit testing.
- Therefore, this is one of the most commonly used integration testing approaches.
- In this approach, both stubs and drivers are required to be designed.

Regression Testing

Each time a new module is added as part of integration testing, the software changes. New data flow paths are established, new I/O may occur, and new control logic is invoked. **Regression testing** is the re-execution of some subset of tests that have already been conducted to ensure that changes have not propagated unintended side effects.

The regression test suite (the subset of tests to be executed) contains three different classes of test cases:

- A representative sample of tests that will exercise all software functions.
- Additional tests that focus on software functions that are likely to be affected by the change.
- Tests that focus on the software components that have been changed.

System Testing:

The sub-systems are integrated to the entire system. The testing process is concerned with finding errors which results from anticipated interactions between sub-systems and system

components. It is also concerned with validating that the system meets its functional and non-functional requirements.

Acceptance testing:

This is the final stage in the testing process before the system is accepted for operational use. The system is tested with data supplied by the system producer rather than stimulated test data. Acceptance testing may reveal errors and omissions in the system requirements definition because the real data exercise the system in different ways from the test data. Acceptance testing may also reveal requirements problems where the systems facilities do not really meet the user's needs or the system performance is unacceptable.

i) Alpha testing:

- Acceptance testing is sometimes called alpha testing. The alpha testing process continues until the system developer and the client agree with the delivered system is an acceptable implementation of the system requirements.

ii) Beta testing:

- When a system is to be marketed as a software product, a testing process called beta testing is often used. Beta testing involves delivering a system to a number of potential customers to agree to use that system. They report problems to the system developers. This exposes the product to real use and detects errors which may not have been anticipated by the system builders. After this feedback, the system is modified and either released or further beta testing or for general sale.