SOFTWARE ENGINEERING
UNIT-4

1. **Coding and Testing**: **Coding,**
2. Code Review,
3. Software Documentation,
4. **Testing,**
5. Unit Testing,
6. Black-Box Testing,
7. White-Box Testing,
8. Debugging,
9. Program Analysis Tool,
10. Integration Testing,
11. Testing Object-Oriented Programs,
12. System Testing,
13. Some General Issues Associated with Testing

1. CODING AND TESTING

- **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.
- That is, a module is tested independently as and when its coding is complete.
- 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 integration plan, according to which different modules are integrated together.
- During each integration step, a number of modules are added to the partially integrated system and the resultant system is tested.
- **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.
- Testing of a professional software is carried out using a large number of test cases.
- It is usually the case that many of the different test cases can be executed in parallel by different team members.
- Therefore, to reduce the testing time, during the testing phase the largest manpower (compared to all other life cycle phases) is deployed.
- In a typical development organisation, at any time, the maximum number of software engineers can be found to be engaged in testing activities.
- It is not very surprising then that in the software industry there is always a large demand for software test engineers.
- Now Software testers are looked upon as masters of specialized concepts, techniques, and tools.
- Testing a software product is as much challenging as initial development activities such as specifications, design, and coding.
- Moreover, testing involves a lot of creative thinking.

CODING

- The input to the coding phase is the design document.
- Design document contains both module structure (e.g., a structure chart) and 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.
- During the coding phase, different modules identified in the design document are coded according to their respective module specifications.
- 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.
- 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.

- A coding standard lists several rules to be followed during coding.
- Besides the coding standards, several coding guidelines are also prescribed by software companies.
- But, what is the Difference between a coding guideline and a coding standard? It is **mandatory for the programmers to follow the coding standards**.
- Coding guidelines provide some general suggestions regarding the coding style to be followed.
- After a module has been coded, 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.
- It is important to detect as many errors as possible during code reviews.

Coding Standards and Guidelines

- Good software development organisations usually develop their own coding standards and guidelines depending on what suits their organisation best and based on the specific types of software they develop.

Representative coding standards

- **Rules for limiting the use of globals:** These rules list what types of data can be declared global.
- **Standard headers for different modules:** The header of different modules should have standard format and information for ease of understanding and maintenance.
- The following is an example of header format that is being used in some companies:
  Name of the module.
  Date on which the module was created.
  Author's name.
  Modification history.
  Synopsis of the module.
- **Naming conventions for global variables, local variables, and constant identifiers:** A popular naming convention is that variables are named using mixed case lettering. Global variable names would always start with a capital letter (e.g., GlobalData) and local variable names start with small letters (e.g., localData). Constant names should be formed using capital letters only (e.g., 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. For example, all functions while encountering an error condition should either return a 0 or 1 consistently, independent of which programmer has written the code. This facilitates reuse and debugging.
- **Representative coding guidelines:** The following are some representative coding guidelines that are recommended by many software development organisations. Wherever necessary, the rationale behind these guidelines is also mentioned.
- **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. Clever coding can obscure meaning of the code and reduce code understandability; thereby making maintenance and debugging difficult and expensive.
- **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 (indistinct) 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. For example, some programmers make use of a temporary loop variable for also computing and storing the final result.
- **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. For the same reason, lengthy functions are likely to have disproportionately larger number of bugs.
- **Do not use GO TO statements:** Use of GO TO statements makes a program unstructured. This makes the program very difficult to understand, debug, and maintain.

2. 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.
- Obviously, 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 recognized as an extremely cost-effective strategy for eliminating coding errors and for producing high quality code.
- Reviews directly detect errors.
- Testing only helps detect failures
- Normally, the following two types of reviews are carried out on the code of a module: Code inspection.
  Code walkthrough.

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 for making this naive but useful analysis technique more effective.
- These guidelines are based on personal experience, common sense, several other subjective factors.

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 common types of errors that usually occur in the code due to **programmer mistakes.**
- 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.
- As an example of the type of errors detected during code inspection, consider the classic error of writing a procedure that modifies a formal parameter and then calls it with a constant actual parameter.
- It is more likely that such an error can be discovered by specifically looking for this kinds of mistakes in the code, rather than by simply hand simulating execution of the code.
- In addition to the commonly made errors, adherence to coding standards is also checked during code inspection.
- 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 parameter 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.

Clean Room Testing

- Clean room testing was implemented at IBM.
- This type of testing applies on walkthroughs, inspection, and formal verification.
- The programmers are not allowed to test any of their code by executing the code other than doing some syntax testing using a compiler.
- It is interesting to note that the term cleanroom was first coined at IBM by drawing design to the semiconductor fabrication units where defects are avoided by manufacturing in an ultra-clean atmosphere.
- This technique reportedly produces documentation and code that is more reliable and maintainable than other development methods relying heavily on code execution-based testing.
- The main problem with this approach is that testing effort is increased as walkthroughs, inspection, and verification are time consuming for detecting all simple errors.

3. SOFTWARE DOCUMENTATION

- When a software is developed,
  - 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.
  - Even when an engineer leaves the organisation, and a new engineer comes in, he can build up the required knowledge easily by referring to the documents.
  - Production of good documents helps the manager to effectively track the progress of the project. The project manager would know that some measurable progress has been achieved, if the results of some pieces of work has been documented and the same has been reviewed.
- Different types of software documents can broadly be classified into the following:
  - **Internal documentation:** These are provided in the source code itself.
  - **External documentation:** These are the supporting documents such as SRS document, installation document, user manual, design document, and test document. We discuss these two types of documentation in the next section.

Internal Documentation

- Internal documentation is the code comprehension features 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.
- Careful experiments suggest that out of all types of internal documentation, meaningful variable names is most useful while trying to understand a piece of code.
- A good style of code commenting is to write to clarify certain non-obvious aspects of the working of the code, rather than cluttering the code with trivial comments.

External Documentation

- External documentation is provided through various types of supporting documents such as users' manual, software requirements specification document, design document, test document, etc.
- 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 requierd 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.

4. TESTING

- **Testing** is an important phase in software development.
- The aim of program testing is to help realize identify all defects in a program.
- Testing of a professional software is carried out using a large number of test cases.
- It is usually the case that many of the different test cases can be executed in parallel by different team members.
- Therefore, to reduce the testing time, during the testing phase the largest manpower (compared to all other life cycle phases) is deployed.
- In a typical development organisation, at any time, the maximum number of software engineers can be found to be engaged in testing activities.
- It is not very surprising then that in the software industry there is always a large demand for software test engineers.
- Now Software testers are looked upon as masters of specialized concepts, techniques, and tools.
- Testing a software product is as much challenging as initial development activities such as specifications, design, and coding.
- Moreover, testing involves a lot of creative thinking.

Basic Concepts and Terminologies

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.
- A highly simplified view of program testing is schematically shown in Figure 10.1.

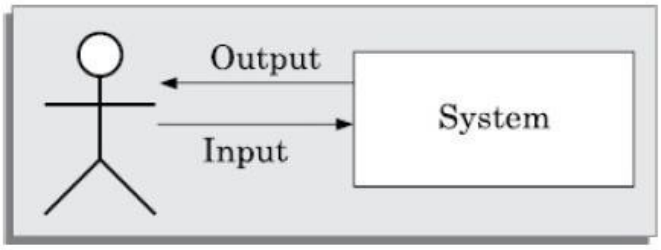

Figure 10.1: A simplified view of program testing.

- The tester has been shown as a stick icon, who inputs several test data to the system and observes the outputs produced by it to check if the system fails on some specific inputs.
- 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.
- For examples, a software might fail for a test case only when a network connection is enabled.

Terminologies

- As is true for any specialized domain, the area of software testing has come to be associated with its own set of terminologies. In the following, we discuss a few important terminologies that have been standardized by the IEEE Standard Glossary of Software Engineering Terminology :
- 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. For example, during coding a programmer might commit the mistake of

- not initializing a certain variable, or
- might overlook the errors that might arise in some exceptional situations such as division by zero in an arithmetic operation.
- Both these mistakes can lead to an incorrect result.

- 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. One example of an error is a call made to a wrong function. The terms error, fault, bug, and defect are considered to be synonyms in the area of program testing.
  Example 10.2 Can a designer's mistake give rise to a program error? Give an example of a designer's mistake and the corresponding program error.
  Answer: Yes, a designer's mistake give rise to a program error. For example, a requirement might be overlooked by the designer, which can lead to it being overlooked in the code as well.
- A **failure** of a program essentially denotes an incorrect behavior exhibited by the program during its execution. An incorrect behavior is observed either as an incorrect result produced or as an inappropriate activity carried out by the program. Every failure is caused by some bugs present in the program. In other words, we can say that every software failure can be traced to some bug or other present in the code. The number of possible ways in which a program can fail is extremely large. Out of the large number of ways in which a program can fail, in the following we give three randomly selected examples:
  – The result computed by a program is 0, when the correct result is 10.
  – A program crashes on an input.
  – A robot fails to avoid an obstacle and collides with it.
  It may be noted that mere presence of an error in a program code may not necessarily lead to a failure during its execution.
  Example 10.3 Give an example of a program error that may not cause any failure.
  Answer: Consider the following C program segment:
  In the above code, if the variable roll assumes zero or some negative value under some circumstances, then an array index out of bound type of error would result. However, it may be the case that for all allowed input values the variable roll is always assigned positive values. Then, the else clause is unreachable and no failure would occur. Thus, even if an error is present in the code, it does not show up as an error since it is unreachable for normal input values.
  Explanation: An array index out of bound type of error is said to occur, when the array index variable assumes a value beyond the array bounds.
- 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. As an example, consider the different execution modes of certain text editor software.
- 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. In the test case, the input, output, and the state at which the input would be applied is designed such that the scenario can be executed.
- 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.

- A test case is said to be a **positive test case** if it is designed to test whether the software correctly performs a required functionality.
- A test case is said to be **negative test case**, if it is designed to test whether the software carries out something that is not required of the system.
- As one example each of a positive test case and a negative test case, consider a program to manage user login.
- A positive test case can be designed to check if a login system validates a user with the correct user name and password.
- A negative test case in this case can be a test case that checks whether the login functionality validates and admits a user with wrong or bogus login user name or password.

- A **test suite** is the set of all test that have been designed by a tester to test a given program.
- **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. In other words, the testability of a requirement is the degree to which an implementation of it can be adequately tested to determine its conformance to the requirement.

  Example 10.4 Suppose two programs have been written to implement essentially the same functionality. How can you determine which of these is more testable?

  Answer: A program is more testable, if it can be adequately tested with less number of test cases. Obviously, a less complex program is more testable. The complexity of a program can be measured using several types of metrics such as number of decision statements used in the program. Thus, a more testable program should have a lower structural complexity metric.
- A **failure mode** of software denotes an observable way in which it can fail. In other words, all failures that have similar observable symptoms, constitute a failure mode. As an example of the failure modes of a software, consider a railway ticket booking software that has three failure modes—failing to book an available seat, incorrect seat booking (e.g., booking an already booked seat), and system crash.
- **Equivalent faults** denote two or more bugs that result in the system failing in the same failure mode. As an example of equivalent faults, consider the following two faults in C language—division by zero and illegal memory access errors. These two are equivalent faults, since each of these leads to a program crash.
- **Verification versus validation**
  (Follow the Running Notes – Material has been given to all the students)

Testing Activities

- Testing involves performing the following main 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. We discuss a few important test case design techniques later in this Chapter.
- **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. The test cases for which the system fails are noted down for later debugging.
- **Locate error:** In this activity, the failure symptoms are analysed to locate the errors. For each failure observed during the previous activity, the statements that are in error are identified.

- **Error correction:** After the error is located during debugging, the code is appropriately changed to correct the error. The testing activities have been shown schematically in Figure 10.2. As can be seen, the test cases are first designed, the test cases are run to detect failures. The bugs causing the failure are identified through debugging, and the identified error is corrected of all the above mentioned testing activities, debugging often turns out to be the most time-consuming activity.

Figure 10.2: Testing process.

Testing in the Large versus Testing in the Small

- A software product is normally tested in three levels or stages:

Unit testing
Integration testing
System testing

- During unit testing, the individual functions (or units) of a program are tested.
- Unit testing is referred to as testing in the small, whereas integration and system testing are referred to as testing in the large.
- After testing all the units individually, the units are slowly integrated and tested after each step of integration (integration testing).
- Finally, the fully integrated system is tested (system testing).
- Integration and system testing are known as testing in the large.

5. 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.

Driver and stub modules

- 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.

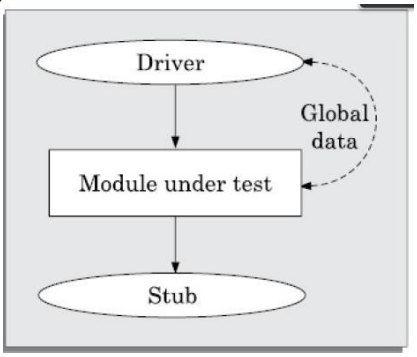

Figure 10.3: Unit testing with the help of driver and stub modules.

Stub: The role of stub and driver modules is pictorially shown in Figure 10.3.

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 behavior.

For example, a stub procedure may produce the expected behavior using a simple table look up mechanism.

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.

6. 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.
- The following are the two main approaches available to design black box test cases:
  - Equivalence class partitioning
  - Boundary value analysis

In the following subsections, we will elaborate these two test case design techniques.

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 $[-\infty,0]$, $[11,+\infty]$.

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.
- The reason behind programmers committing such errors might purely be due to psychological factors.
- Programmers often fail to properly address the special processing required by the input values that lie at the boundary of the different equivalence classes.
- For example, programmers may improperly use < instead of <=, or conversely <= for <, etc.
- 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 (i.e., consist of a discrete collection 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}.
- **Example 10.9** For a function that computes the square root of the integer values in the range of 0 and 5000, determine the boundary value test suite.
  Answer: There are three equivalence classes—The set of negative integers, the set of integers in the range of 0 and 5000, and the set of integers larger than 5000. The boundary value-based test suite is: {0,-1,5000,5001}.

7. WHITE-BOX TESTING

- 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 constitute the **fault model** of the strategy.
- **Coverage-based testing:** A coverage-based testing strategy attempts to execute (or cover) certain elements of a program. Popular examples of coverage-based testing strategies are statement coverage, branch coverage, multiple condition coverage, and path coverage-based testing.
- **Testing criterion for coverage-based testing :** A coverage-based testing strategy typically targets to execute (i.e., cover) certain program elements for discovering failures. The set of specific program elements that a testing strategy targets to execute is called the testing criterion of the strategy.
- **Stronger versus weaker testing:** compare two testing strategies by determining whether one is stronger, weaker, or complementary to the other.
  - A white-box testing strategy is said to be **stronger** than another strategy, if the stronger testing strategy covers **all program elements** covered by the weaker testing strategy, and the stronger strategy additionally covers at least one program element that is not covered by the weaker strategy.
  - When none of two testing strategies fully covers the program elements exercised by the other, then the two are called complementary testing strategies.
  - The concepts of stronger, weaker, and complementary testing are schematically illustrated in Figure 10.6. Observe in Figure 10.6(a) that testing strategy A is stronger than B since B covers only a proper subset of elements covered by B. On the other hand, Figure 10.6(b) shows A and B are complementary testing strategies since some elements of A are not covered by B and vice versa. If a stronger testing has been performed, then a weaker testing need not be carried out.

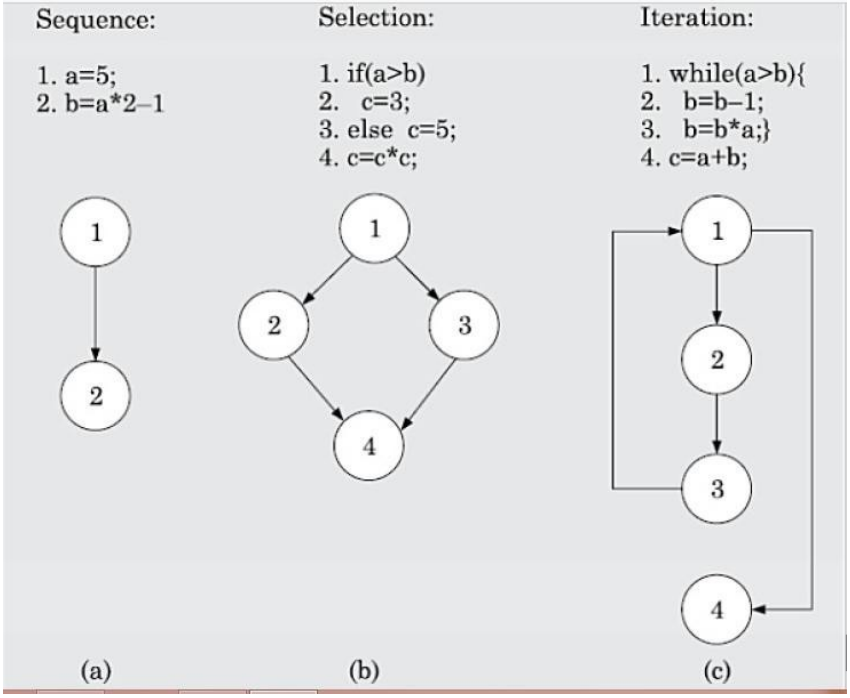

Figure 10.6: Illustration of stronger, weaker, and complementary testing strategies.

A test suite should, however, be enriched by using various complementary testing strategies.

Statement Coverage

- The statement coverage strategy aims to design test cases so as to execute every statement in a program at least once. The principal idea governing the statement coverage strategy is that unless a statement is executed, there is no way to determine whether an error exists in that statement.
- **Example 10.11** Design statement coverage-based test suite for the following Euclid's GCD computation program:

```
int computeGCD(x,y)
int x,y;
{
1 while (x != y)
{ 2 if (x>y) then
3 x=x-y;
4 else y=y-x;
5 }
6 return x;
}
```

Answer: To design the test cases for the statement coverage, the conditional expression of the while statement needs to be made true and the conditional expression of the if statement needs to be made both true and false. By choosing the test set {(x = 3, y = 3), (x = 4, y = 3), (x = 3, y = 4)}, all statements of the program would be executed at least once.

Branch Coverage

- A test suite satisfies branch coverage, if it makes each branch condition in the program to assume true and false values in turn. In other words, for branch coverage each branch in the CFG representation of the program must be taken at least once, when the test suite is executed.
- Branch testing is also known as edge testing, since in this testing scheme, each edge of a program's control flow graph is traversed at least once.

Example 10.12 For the program of Example 10.11, determine a test suite to achieve branch coverage.
Answer: The test suite {(x = 3, y = 3), (x = 3, y = 2), (x = 4, y = 3), (x = 3, y = 4)} achieves branch coverage. It is easy to show that branch coverage-based testing is a stronger testing than statement coverage-based testing. We can prove this by showing that branch coverage ensures statement coverage, but not vice versa.

Multiple Condition Coverage

- In the multiple condition (MC) coverage-based testing, test cases are designed to make each component of a composite conditional expression to assume both true and false values. For example, consider the composite conditional expression ((c1 .and.c2).or.c3).
- A test suite would achieve MC coverage, if all the component conditions c1, c2 and c3 are each made to assume both true and false values. Branch testing can be considered to be a simplistic condition testing strategy where only the compound conditions appearing in the different branch statements are made to assume the true and false values. It is easy to prove that condition testing is a stronger testing strategy than branch testing. For a composite conditional expression of n components, 2n test cases are required for multiple condition coverage. Thus, for multiple condition coverage, the number of test cases increases exponentially with the number of component conditions. Therefore, multiple condition coverage-based testing technique is practical only if n (the number of conditions) is small.

  Example 10.13 Give an example of a fault that is detected by multiple condition coverage, but not by branch coverage.
  Answer: Consider the following C program segment:

```
if(temperature>150 || temperature>50)
setWarningLightOn();
```

  The program segment has a bug in the second component condition, it should have been temperature<50. The test suite {temperature=160, temperature=40} achieves branch coverage. But, it is not able to check that setWarningLightOn(); should not be called for temperature values within 150 and 50.

Path Coverage

- A test suite achieves path coverage if it exeutes each linearly independent paths (o r basis paths) at least once. A linearly independent path can be defined in terms of the control flow graph (CFG) of a program. Therefore, to understand path coverage-based testing strategy, we need to first understand how the CFG of a program can be drawn.

Control flow graph (CFG)

- A control flow graph describes how the control flows through the program. We can define a control flow graph as the following: A control flow graph describes the sequence in which the different instructions of a program get executed. In order to draw the control flow graph of a program, we need to first number all the statements of a program. The different numbered statements serve as nodes of the control flow graph (see Figure 10.5).

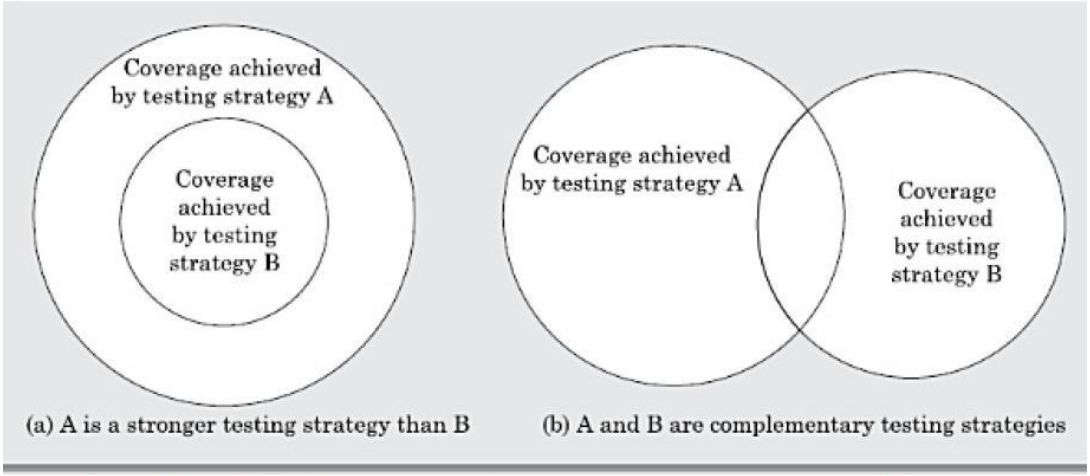

Figure 10.5: CFG for (a) sequence, (b) selection, and (c) iteration type of constructs.

- There exists an edge from one node to another, if the execution of the statement representing the first node can result in the transfer of control to the other node.
- More formally, we can define a CFG as follows. A CFG is a directed graph consisting of a set of nodes and edges (N, E), such that each node n N corresponds to a unique program statement and an edge exists between two nodes if control can transfer from one node to the other.
- We can easily draw the CFG for any program, if we know how to represent the sequence, selection, and iteration types of statements in the CFG. After all, every program is constructed by using these three types of constructs only. Figure 10.5 summarises how the CFG for these three types of constructs can be drawn.
- The CFG representation of the sequence and decision types of statements is straight forward. Please note carefully how the CFG for the loop (iteration) construct can be drawn.
- For iteration type of constructs such as the while construct, the loop condition is tested only at the beginning of the loop and therefore always control flows from the last statement of the loop to the top of the loop.
- That is, the loop construct terminates from the first statement (after the loop is found to be false) and does not at any time exit the loop at the last statement of the loop. Using these basic ideas, the CFG of the program given in Figure 10.7(a) can be drawn as shown in Figure 10.7(b).

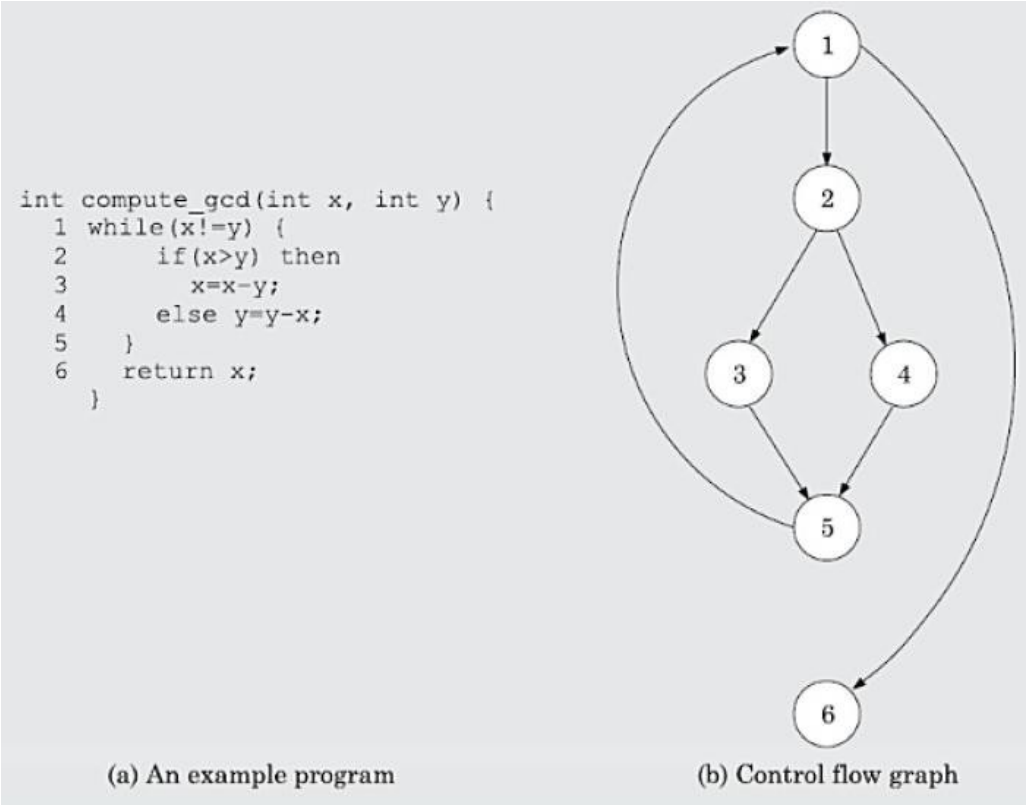

Figure 10.7: Control flow diagram of an example program.

Path

- A path through a program is any node and edge sequence from the start node to a terminal node of the control flow graph of a program. Please note that a program can have more than one terminal nodes when it contains multiple exit or return type of statements. Writing test cases to cover all paths of a typical program is impractical since there can be an infinite number of paths through a program in presence of loops.
- For example, in Figure 10.5(c), there can be an infinite number of paths such as 12314, 12312314, 12312312314, etc. If coverage of all paths is attempted, then the number of test cases required would become infinitely large. For this reason, path coverage testing does not try to cover all paths, but only a subset of paths called linearly independent p a t h s (o r basis paths). Let us now discuss what are linearly independent paths and how to determine these in a program.

Linearly independent set of paths (or basis path set)

- A set of paths for a given program is called linearly independent set of paths (or the set of basis paths or simply the basis set), if each path in the set introduces at least one new edge that is not included in any other path in the set.
- If a set of paths is linearly independent of each other, then no path in the set can be obtained through any linear operations (i.e., additions or subtractions) on the other paths in the set.
- According to the above definition of a linearly independent set of paths, for any path in the set, its subpath cannot be a member of the set.

McCabe's Cyclomatic Complexity Metric

- McCabe obtained his results by applying graph-theoretic techniques to the control flow graph of a program. McCabe's cyclomatic complexity defines an upper bound on the number of independent paths in a program. We discuss three different ways to

compute the cyclomatic complexity. For structured programs, the results computed by all the three methods are guaranteed to agree.

- **Method:** Given a control flow graph G of a program, the cyclomatic complexity V(G) can be computed as:

  $$V(G) = E - N + 2$$

  where, N is the number of nodes of the control flow graph and E is the number of edges in the control flow graph. For the CFG of example shown in Figure 10.7, E = 7 and N = 6. Therefore, the value of the Cyclomatic complexity = 7 – 6 + 2 = 3.

Steps to carry out path coverage-based testing

- The following is the sequence of steps that need to be undertaken for deriving the path coverage-based test cases for a program:

  1. Draw control flow graph for the program.
  2. Determine the McCabe's metric V(G).
  3. Determine the cyclomatic complexity. This gives the minimum number of test cases required to achieve path coverage.
  4. repeat

  Test using a randomly designed set of test cases. Perform dynamic analysis to check the path coverage achieved. until at least 90 per cent path coverage is achieved.

Data Flow-based Testing

- Data flow based testing method selects test paths of a program according to the definitions and uses of different variables in a program. Consider a program P . For a statement numbered S of P , let DEF(S) = {X /statement S contains a definition of X } and USES(S)= {X /statement S contains a use of X }
- For the statement S: a=b+c;, DEF(S)={a}, USES(S)={b, c}. The definition of variable X at statement S is said to be live at statement S1, if there exists a path from statement S to statement S1 which does not contain any definition of X. All definitions criterion is a test coverage criterion that requires that an adequate test set should cover all definition occurrences in the sense that, for each definition occurrence, the testing paths should cover a path through which the definition reaches a use of the definition.

Mutation Testing

- All white-box testing strategies that we have discussed so far are coverage-based testing techniques.
- In contrast, mutation testing is a fault-based testing technique in the sense that mutation test cases are designed to help detect specific types of faults in a program.
- In mutation testing, a program is first tested by using an initial test suite designed by using various white box testing strategies that we have discussed.
- After the initial testing is complete, mutation testing can be taken up.
- The idea behind mutation testing is to make a few arbitrary changes to a program at a time.
- Each time the program is changed, it is called a mutated program and the change effected is called a mutant.
- An underlying assumption behind mutation testing is that all programming errors can be expressed as a combination of simple errors.
- A mutation operator makes specific changes to a program. For example, one mutation operator may randomly delete a program statement.
- A mutant may or may not cause an error in the program.
- If a mutant does not introduce any error in the program, then the original program and the mutated program are called equivalent programs.

- A mutated program is tested against the original test suite of the program.
- If there exists at least one test case in the test suite for which a mutated program yields an incorrect result, then the mutant is said to be dead, since the error introduced by the mutation operator has successfully been detected by the test suite.
- If a mutant remains alive even after all the test cases have been exhausted, the test suite is enhanced to kill the mutant.

8. 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. In this Section, we shall summarise the important approaches that are available to identify the error locations. Each of these approaches has its own advantages and disadvantages and therefore each will be useful in appropriate circumstances. We also provide some guidelines for effective debugging.

Debugging Approaches

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

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 (also called a source code debugger), because values of different variables can be easily checked and break points and watch points can be easily set to test the values of variables effortlessly.
- Single stepping using a symbolic debugger is another form of this approach, where the developer mentally computes the expected result after every source instruction and checks whether the same is computed by single stepping through the program.

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.
- Unfortunately, as the number of source lines to be traced back increases, the number of potential backward paths increases and may become unmanageably large for complex programs, limiting the use of this approach.

Cause elimination method

- In this approach, once a failure is observed, the symptoms of the failure (i.e., certain variable is having a negative value though it should be positive, etc.) 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.
- A related technique of identification of the error from the error symptom is the software fault tree analysis.

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.

Program slicing makes use of the fact that an error in the value of a variable can be caused by the statements on which it is data dependent.

9. PROGRAM ANALYSIS TOOLS

- A program analysis tool usually is an automated tool that takes either the source code or the executable code of a program as input and produces reports regarding several important characteristics of the program, such as its size, complexity, adequacy of commenting, adherence to programming standards, adequacy of testing, etc.
- Program analysis tools are classified into the following two broad categories:
    - Static analysis tools
    - Dynamic analysis tools
- These two categories of program analysis tools are discussed in the following

Static Analysis Tools

- Static program analysis tools assess and compute various characteristics of a program without executing it. Typically, static analysis tools analyze the source code to compute certain metrics characterizing the source code (such as size, cyclomatic complexity, etc.) and also report certain analytical conclusions. These also check the conformance of the code with the prescribed coding standards. In this context, it displays the following analysis results:
- To what extent the coding standards have been adhered to?
- Whether certain programming errors such as uninitialized variables, mismatch between actual and formal parameters, variables that are declared but never used, etc., exist?
- A list of all such errors is displayed.

Dynamic Analysis Tools

- Dynamic program analysis tools can be used to evaluate several program characteristics based on an analysis of the run time behaviour of a program.
- These tools usually record and analyse the actual behaviour of a program while it is being executed.
- A dynamic program analysis tool (also called a dynamic analyser) usually collects execution trace information by instrumenting the code.
- Code instrumentation is usually achieved by inserting additional statements to print the values of certain variables into a file to collect the execution trace of the program.
- The instrumented code when executed, records the behaviour of the software for different test cases.
- An important characteristic of a test suite that is computed by a dynamic analysis tool is the extent of coverage achieved by the test suite.
- After a software has been tested with its full test suite and its behavior recorded, the dynamic analysis tool carries out a post execution analysis and produces reports which describe the coverage that has been achieved by the complete test suite for the program.

10. 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).
- For example, it is checked that no parameter mismatch occurs when one module invokes the functionality of another module.

- Thus, the primary objective of integration testing is to test the module interfaces, i.e., there are no errors in parameter passing, when one module invokes the functionality of another module.
- 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 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 realize the full system. After each integration step, the partially integrated system is tested.
- An important factor that guides the integration plan is the module dependency graph.
- Thus, 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:
    - Big-bang approach to integration testing.
    - Bottom-up approach to integration testing Mixed (also called sandwiched) approach to integration testing.
    - Top-down approach to integration testing

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.
- Therefore, debugging errors reported during big-bang integration testing are very expensive to fix.
- As a result, big-bang integration testing is almost never used for large programs.

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 the each subsystem are integrated. Thus, the subsystems can be integrated separately and independently.
- The primary purpose of carrying out the integration testing a subsystem is to test whether the interfaces among various modules making up the subsystem work satisfactorily.
- The test cases must be carefully chosen to exercise the interfaces in all possible manners.
- In a pure bottom-up testing no stubs are required, and only test-drivers are required. Large software systems normally require several levels of subsystem testing, lower-level subsystems are successively combined to form higher-level subsystems.
- The principal advantage of bottom- up integration testing is that several disjoint subsystems can be tested simultaneously.
- Another advantage of bottom-up testing is that the low-level modules get tested thoroughly, since they are exercised in each integration step. Since the low-level

modules do I/O and other critical functions, testing the low-level modules thoroughly increases the reliability of the system.

- A disadvantage of bottom-up testing is the complexity that occurs when the system is made up of a large number of small subsystems that are at the same level. This extreme case corresponds to the big-bang approach.

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.
- An advantage of top-down integration testing is that it requires writing only stubs, and stubs are simpler to write compared to drivers.
- A disadvantage of the top-down integration testing approach is that in the absence of lower-level routines, it becomes difficult to exercise the top-level routines in the desired manner since the lower level routines usually perform input/output (I/O) operations.

Mixed approach to integration testing

- The mixed (also called sandwiched) integration testing follows a combination of top-down and bottom-up testing approaches.
- In 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.

Phased versus Incremental Integration Testing

- Big-bang integration testing is carried out in a single step of integration. In contrast, in the other strategies, integration is carried out over several steps. In these later strategies, modules can be integrated either in a phased or incremental manner. A comparison of these two strategies is as follows:

  In incremental integration testing, only one new module is added to the partially integrated system each time.

  In phased integration, a group of related modules are added to the partial system each time.

- Obviously, phased integration requires less number of integration steps compared to the incremental integration approach. However, when failures are detected, it is easier to debug the system while using the incremental testing approach since the errors can easily be traced to the interface of the recently integrated module. Please observe that a degenerate case of the phased integration testing approach is big-bang testing.

11. TESTING OBJECT-ORIENTED PROGRAMS

- During the initial years of object-oriented programming, it was believed that object-orientation would, to a great extent, reduce the cost and effort incurred on testing.

- This thinking was based on the observation that object-orientation incorporates several good programming features such as encapsulation, abstraction, reuse through inheritance, polymorphism, etc., thereby chances of errors in the code is minimised.
- However, it was soon realised that satisfactory testing object-oriented programs is much more difficult and requires much more cost and effort as compared to testing similar procedural programs.
- The main reason behind this situation is that various object-oriented features introduce additional complications and scope of new types of bugs that are present in procedural programs.
- Therefore additional test cases are needed to be designed to detect these. We examine these issues as well as some other basic issues in testing object-oriented programs in the following subsections.

What is a Suitable Unit for Testing Object-oriented Programs?

- For procedural programs, we had seen that procedures are the basic units of testing.
- That is, first all the procedures are unit tested. Then various tested procedures are integrated together and tested.
- Thus, as far as procedural programs are concerned, procedures are the basic units of testing.
- Since methods in an object-oriented program are analogous to procedures in a procedural program, can we then consider the methods of object-oriented programs as the basic unit of testing?

Do Various Object-orientation Features Make Testing Easy?

- The implications of different object-orientation features in testing are as follows.
- **Encapsulation:** The encapsulation feature helps in data abstraction, error isolation, and error prevention.
    - Encapsulation prevents the tester from accessing the data internal to an object.
    - The encapsulation feature though makes testing difficult, the difficulty can be overcome to some extent through use of appropriate state reporting methods.
- **Inheritance:** The inheritance feature helps in code reuse and was expected to simplify testing.
    - It was expected that if a class is tested thoroughly, then the classes that are derived from this class would need only incremental testing of the added features.
    - However, this is not the case. Even if the base class class has been thoroughly tested, the methods inherited from the base class need to be tested again in the derived class.
    - The reason for this is that the inherited methods would work in a new context (new data and method definitions).
    - As a result, correct behaviour of a method at an upper level, does not guarantee correct behaviour at a lower level. Therefore, retesting of inherited methods needs to be followed as a rule, rather as an exception.
- **Dynamic binding:** Dynamic binding was introduced to make the code compact, elegant, and easily extensible. However, as far as testing is concerned all possible bindings of a method call have to be identified and tested. This is not easy since the bindings take place at run-time.
- **Object states:** In contrast to the procedures in a procedural program, objects store data permanently. As a result, objects do have significant states. The behaviour of an

object is usually different in different states. That is, some methods may not be active in some of its states. Also, a method may act differently in different states.

Grey-Box Testing of Object-oriented Programs

- Model-based testing is important for object oriented programs, as these test cases help detect bugs that are specific to the object-orientation constructs.
- For object-oriented programs, several types of test cases can be designed based on the design models of object-oriented programs.
- These are called the grey-box test cases.
- The following are some important types of grey-box testing that can be carried on based on UML models:
- **State-model-based testing**
  State coverage: Each method of an object are tested at each state of the object.
  State transition coverage: It is tested whether all transitions depicted in the state model work satisfactorily.
  State transition path coverage: All transition paths in the state model are tested.
- **Use case-based testing**
  Scenario coverage: Each use case typically consists of a mainline scenario and several alternate scenarios. For each use case, the mainline and all alternate sequences are tested to check if any errors show up.
- **Class diagram-based testing**
  Testing derived classes: All derived classes of the base class have to be instantiated and tested. In addition to testing the new methods defined in the derivec. lass, the inherited methods must be retested.
  Association testing: All association relations are tested.
  Aggregation testing: Various aggregate objects are created and tested.
  Sequence diagram-based testing
  Method coverage: All methods depicted in the sequence diagrams are covered.
  Message path coverage: All message paths that can be constructed from the sequence diagrams are covered.

11.5 Integration Testing of Object-oriented Programs

There are two main approaches to integration testing of object-oriented programs:

• Thread-based
• Use based

- **Thread-based approach:** In this approach, all classes that need to collaborate to realise the behaviour of a single use case are integrated and tested. After all the required classes for a use case are integrated and tested, another use case is taken up and other classes (if any) necessary for execution of the second use case to run are integrated and tested. This is continued till all use cases have been considered.
- **Use-based approach:** Use-based integration begins by testing classes that either need no service from other classes or need services from at most a few other classes. After these classes have been integrated and tested, classes that use the services from the already integrated classes are integrated and tested. This is continued till all the classes have been integrated and tested.

12. SYSTEM TESTING

- After all the units of a program have been integrated together and tested, system testing is taken up. System tests are designed to validate a fully developed system to assure that it meets its requirements. The test cases are therefore designed solely based on the SRS document.

- The system testing procedures are the same for both object-oriented and procedural programs, since system test cases are designed solely based on the SRS document and the actual implementation (procedural or object oriented) is immaterial. There are essentially three main kinds of system testing depending on who carries out testing:
- 1. **Alpha Testing:** Alpha testing refers to the system testing carried out by the test team within the developing organization.
- 2. **Beta Testing:** Beta testing is the system testing performed by a select group of friendly customers.
- 3. **Acceptance Testing:** Acceptance testing is the system testing performed by the customer to determine whether to accept the delivery of the system.

Smoke Testing

- Smoke testing is carried out before initiating system testing to ensure that system testing would be meaningful, or whether many parts of the software would fail.
- The idea behind smoke testing is that if the integrated program cannot pass even the basic tests, it is not ready for a vigorous testing.
- For smoke testing, a few test cases are designed to check whether the basic functionalities are working.
- For example, for a library automation system, the smoke tests may check whether books can be created and deleted, whether member records can be created and deleted, and whether books can be loaned and returned.

Performance Testing

- Performance testing is an important type of system testing.
- Performance testing is carried out to check whether the system meets the nonfunctional requirements identified in the SRS document.
- There are several types of performance testing corresponding to various types of non-functional requirements. For a specific system, the types of performance testing to be carried out on a system depends on the different non-functional requirements of the system documented in its SRS document.
- All performance tests can be considered as black-box tests.

Stress testing

- Stress testing is also known as endurance testing.
- Stress testing evaluates system performance when it is stressed for short periods of time.
- Stress tests are black-box tests which are designed to impose a range of abnormal and even illegal input conditions so as to stress the capabilities of the software.

Volume testing

- Volume testing checks whether the data structures (buffers, arrays, queues, stacks, etc.) have been designed to successfully handle extraordinary situations.
- For example, the volume testing for a compiler might be to check whether the symbol table overflows when a very large program is compiled.

Configuration testing

- Configuration testing is used to test system behaviour in various hardware and software configurations specified in the requirements.
- Sometimes systems are built to work in different configurations for different users.
- For instance, a minimal system might be required to serve a single user, and other extended configurations may be required to serve additional users during configuration testing.

- The system is configured in each of the required configurations and depending on the specific customer requirements, it is checked if the system behaves correctly in all required configurations.

Compatibility testing

- This type of testing is required when the system interfaces with external systems (e.g., databases, servers, etc.).
- Compatibility aims to check whether the interfaces with the external systems are performing as required. For instance, if the system needs to communicate with a large database system to retrieve information, compatibility testing is required to test the speed and accuracy of data retrieval.

Regression testing

- This type of testing is required when a software is maintained to fix some bugs or enhance functionality, performance, etc. Regression testing is also discussed in Section 10.13.

Recovery testing

- Recovery testing tests the response of the system to the presence of faults, or loss of power, devices, services, data, etc. The system is subjected to the loss of the mentioned resources (as discussed in the SRS document) and it is checked if the system recovers satisfactorily. For example, the printer can be disconnected to check if the system hangs. Or, the power may be shut down to check the extent of data loss and corruption.

Maintenance testing

- This addresses testing the diagnostic programs, and other procedures that are required to help maintenance of the system. It is verified that the artifacts exist and they perform properly.

Documentation testing

- It is checked whether the required user manual, maintenance manuals, and technical manuals exist and are consistent. If the requirements specify the types of audience for which a specific manual should be designed, then the manual is checked for compliance of this requirement.

Usability testing

- Usability testing concerns checking the user interface to see if it meets all user requirements concerning the user interface. During usability testing, the display screens, messages, report formats, and other aspects relating to the user interface requirements are tested. A GUI being just being functionally correct is not enough. Therefore, the GUI has to be checked against the checklist we discussed in Sec. 9.5.6.

Security testing

- Security testing is essential for software that handle or process confidential data that is to be guarded against pilfering. It needs to be tested whether the system is fool-proof from security attacks such as intrusion by hackers. Over the last few years, a large number of security testing techniques have been proposed, and these include password cracking, penetration testing, and attacks on specific ports, etc.

Error Seeding

- Sometimes customers specify the maximum number of residual errors that can be present in the delivered software. These requirements are often expressed in terms of maximum number of allowable errors per line of source code. The error seeding technique can be used to estimate the number of residual errors in a software. Error seeding, as the name implies, it involves seeding the code with some known errors. In other words, some artificial errors are introduced (seeded) into the program. The

number of these seeded errors that are detected in the course of standard testing is determined. These values in conjunction with the number of unseeded errors detected during testing can be used to predict the following aspects of a program:

- The number of errors remaining in the product.
- The effectiveness of the testing strategy.
- Let N be the total number of defects in the system, and let n of these defects be found by testing.
- Let S be the total number of seeded defects, and let s of these defects be found during testing.

- Therefore, we get: Defects still remaining in the program after testing can be given by: Error seeding works satisfactorily only if the kind seeded errors and their frequency of occurrence matches closely with the kind of defects that actually exist. However, it is difficult to predict the types of errors that exist in a software. To some extent, the different categories of errors that are latent and their frequency of occurrence can be estimated by analyzing historical data collected from similar projects. That is, the data collected is regarding the types and the frequency of latent errors for all earlier related projects.
- This gives an indication of the types (and the frequency) of errors that are likely to have been committed in the program under consideration. Based on these data, the different types of errors with the required frequency of occurrence can be seeded.

13. SOME GENERAL ISSUES ASSOCIATED WITH TESTING

- In this section, we shall discuss two general issues associated with testing. These are —how to document the results of testing and how to perform regression testing.

Test documentation

- A piece of documentation that is produced towards the end of testing is the test summary report.
- This report normally covers each subsystem and represents a summary of tests which have been applied to the subsystem and their outcome.
- It normally specifies the following: What is the total number of tests that were applied to a subsystem. Out of the total number of tests how many tests were successful.
- How many were unsuccessful, and the degree to which they were unsuccessful, e.g., whether a test was an outright failure or whether some of the expected results of the test were actually observed.

Regression testing

- Regression testing spans unit, integration, and system testing. Instead, it is a separate dimension to these three forms of testing.
- Regression testing is the practice of running an old test suite after each change to the system or after each bug fix to ensure that no new bug has been introduced due to the change or the bug fix.
- However, if only a few statements are changed, then the entire test suite need not be run — only those test cases that test the functions and are likely to be affected by the change need to be run.
- Whenever software is changed to either fix a bug, or enhance or remove a feature, regression testing is carried out.