**UNIT-IV**

**CHAPTER-I**

**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. Th at is, a module is tested independently as and when its coding is complete.

Integration and testing of modules is carried out according to an integration plan. The integration plan, according to which different modules are integrated together, usually envisages integration of modules through a number of steps. 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

**Coding**

* The input to the coding phase is the design document produced at the end of the design phase. Please recollect that the design documents contain not only the high-level design of the software in the form of a structure charts (representing the module call relationships), 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. During the coding phase, different modules identified in the design document are coded according to their respective module specifications.
* We can describe the overall objective of the coding phase to be the following. Normally, good software development organizations require their programmers to adhere to some well-defined and standard style of coding which is called their coding standard.
* Software development organizations usually formulate their own coding standards that suit them the most, and require their developers to follow the standards rigorously because of the significant business advantages it offers.

The main advantages of adhering to a standard style of coding are the following:

* 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, such as the way variables are to be named, the way the code is to be laid out, the error return conventions, etc. 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? 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, because reviews are efficient way of removing errors from code as compared to defect elimination using testing.

**Coding Standards and Guidelines**

Good software development organizations usually develop their own coding standards and guidelines depending on what suits their organization best and based on the specific types of software they develop. To give an idea about the types of coding standards that are being used, we shall only list some general coding standards and guidelines that are commonly adopted by many software development organizations, rather than trying to provide an exhaustive list.

**Representative coding standards**

**Rules for limiting the use of global:**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. 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. This is a small write-up about what the module does.
* Different functions supported in the module, along with their input/output parameters
* Global variables accessed/modified by 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., Global Data) and local variable names start with small letters (e.g., local Data). 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 organization. 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 organizations. 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 that are not obvious behavior of the function. An obscure side effect is hard to understand from a casual examination of the code. For example, suppose the value of a global variable is changed or some file I/O is performed, which may be difficult to infer from the function’s name and header information.

**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. The rationale that they give for such multiple use of variables is memory efficiency, e.g., three variables use up three memory locations, whereas when the same variable is used for three different purposes, only one memory location is used. However, there are several things wrong with this approach and hence should be avoided. Some of the problems caused by the use of a variable for multiple purposes are as follows:

* Each variable should be given a descriptive name indicating its purpose. This is not possible if an identifier is used for multiple purposes. Use of a variable for multiple purposes can lead to confusion and make it difficult for somebody trying to read and understand the code.
* Use of variables for multiple purposes usually makes future enhancements more difficult. For example, while changing the final computed result from integer to float type, the programmer might subsequently notice that it has also been used as a temporary loop variable that cannot be a float type.

**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.

**CODE REVIEW**

* Code review and testing are both effective defect removal mechanisms. However, review has been acknowledged to be more cost-effective in removing defects as compared to testing. Over the years, review techniques have become extremely popular and have been generalized for use with other work products such as design documents and the SRS document.
* Code review for a module (that is, a unit) 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 detect 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.
* The rationale behind the above statement is explained as follows. Eliminating an error from code involves three main activities: testing (detecting failures), debugging (locating the errors), 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 considerable efforts that would have been required to locate the errors.
* The procedures for conduction and the final objectives of these two review techniques are very different. In the following two subsections, we discuss these two types of code review techniques.

**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. That is, the reviewer mentally traces the execution through different statements and functions of the code.
* The members note down their findings of their code walkthrough and discuss those in a walkthrough meeting where the coder of the module is present. 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, and several other subjective factors. Therefore, these guidelines should be considered as example rather than as accepted rules to be applied dogmatically. Some of these guidelines are following:
* The size of the team performing code walkthrough should not be either too large or too small. Ideally, it should consist of between three to seven members.
* Discussions in the walkthrough meeting should focus on discovery of errors and avoid deliberations on how to fix the discovered errors.
* In order to foster co-operation 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.

**Code Inspection**

* During code inspection, the code is examined to check for the presence of some common programming errors. This is in contrast to the hand simulation of code execution carried out during code walkthroughs.
* 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 different participants gain by being exposed to another programmer’s errors which they can then consciously try to avoid.
* 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 these 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
* 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.

The following is a checklist of some classical programming errors which can be used during code inspection:

* Use of uninitialized 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.
* Dangling reference caused when the referenced memory has not been allocated.

**Cleanroom Technique**

* Cleanroom technique was pioneered at IBM. This technique relies heavily on walkthroughs, inspection, and formal verification for bug removal. 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 analogy 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 simple errors. Also testing-based error detection is eﬃcient for detecting certain errors that escape manual inspection.

**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 several ways:

* Good documents help to enhance understandability of a piece of code. Code understanding is an important part of any maintenance activity. Availability of good documents help to reduce the effort and time required for maintenance.
* Documents help the users to understand and effectively use the system.
* Good documents help to effectively tackle the manpower turnover1 problem. Even when an engineer leaves the organization, 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 and external documentation

**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
* Out of these different types of internal documentation, which one is the most valuable for understanding a piece of code?
* The above inference, of course, is in contrast to the common expectation that code commenting would be the most useful. The research finding can be far from the truth when comments are written without much thought. For example, the following style of code commenting is not much of a help in understanding the code.
* a=10; /\* a made 10 \*/
* 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.
* Good software development organizations usually ensure good internal documentation by appropriately formulating their coding standards and coding guidelines. Even when a piece of code is carefully commented, meaningful variable names has been found to be the most helpful in understanding a piece of code.

**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 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 code. In other words, all the documents developed for a product should be up-to-date and every change made to the code should be reflected in the relevant external documents.
* Even if only a few documents are not up-to-date, they create inconsistency and lead to confusion. Another important feature required for external documents is proper understandability by the category of users for whom the document is designed. For achieving this, Gunning’s fog index is very useful.

**Gunning’s fog index**

* Gunning’s fog index (developed by Robert Gunning in 1952) is a metric that has been designed to measure the readability of a text document. The computed metric value (fog index) of a document indicates the number of years of formal education that a person should have, in order to be able to comfortably understand that document. That is, if a certain document has a fog index of 12, anyone who has completed his 12th class would not have much diﬃculty in understanding that document.
* The Gunning’s fog index of a document *D* can be computed as follows:
* fog(D ) = 0.4 X (Words/Sentences)+Percent of words having 3 or more syllables
* Observe that the fog index is computed as the sum of two different factors. The first factor computes the average number of words per sentence (total number of words in the document divided by the total number of sentences). This factor therefore accounts for the common observation that long sentences are diﬃcult to understand.
* The second factor measures the percentage of complex words in the document. The complex words are considered to be those with three or more syllabi. Note that a syllable is a group of words that can be independently pronounced. For example, the word “sentence” has three syllables (“sen”, “ten”, and “ce”). Words having more than three syllables are complex words and presence of many such words hamper readability of a document.

**PROBLEM**

Consider the following sentence: “The Gunning’s fog index is based on the premise that use of short sentences and simple words makes a document easy to understand.” Calculate its fog index.

***Solution:*** The given sentence has 23 words. Four of the words have three or more syllabi.

* The fog index of the problem sentence is therefore
* 0.4 × (23/1) + (4/23) × 100 = 26.5
* If a users’ manual is to be designed for use by factory workers who's educational qualification is class 8, then the document should be written such that the Gunning’s fog index of the document does not exceed 8.

**TESTING**

The aim of program testing is to help in identifying 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.

Consider a function taking a floating-point number as argument. If a tester takes 1 second to type in an integer value, then even a million testers would not be able to exhaustively test it after trying for a million years. Even with this obvious limitation of the testing process, we should not underestimate the importance of testing. We must remember that careful testing can expose a large percentage of the defects existing in a program, and therefore, testing provides a practical way of reducing defects in a system.

**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. 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. When the system fails, it is necessary to note down the specific input values for which the failure occurs.
* However, 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 example, a software might fail for a test case only when a network connection is enabled. Unless this condition is documented in the failure report, it becomes difficult to reproduce the failure.



**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 [IEEE, 1990]:
* 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 of the development activities. 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 during program execution.
* An **error** is the result of a mistake committed by a developer in any of the development activities. Mistakes can give rise to an extremely large variety of errors.
* One example error is a call made to a wrong function. The terms *error, fault, bug,* and *defect* are used interchangeably by the program testing community.
* Please note that in the domain of hardware testing, the term *fault* is used with a slightly different connotation [IEEE, 1990] as compared to the terms *error* and *bug*.

Can a designer’s mistake give rise to a program error? Give an example of a designer’s mistake and the corresponding program error.

* A **failure** of a program essentially denotes an incorrect behavior exhibited by the program during its execution. An incorrect behavior is observed either as production of an incorrect result or as an inappropriate activity carried out by the program. Every failure is caused by one or more bugs present in the program.
* In other words, we can say that every software failure can be traced to one or more bugs present in the code. The number of possible bugs that can cause a program failure is extremely large. Out of the large number of the bugs that can cause program failure, 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.

**Verification *versus* validation**

* The objectives of both verification and validation techniques are very similar since both these techniques are designed to help remove errors in a software. In spite of the apparent similarity between their objectives, the underlying principles of these two-bug detection techniques and their applicability are very different.
* We can therefore say that the primary objective of verification is to determine whether the different steps of product development are being carried out correctly, whereas validation is carried out towards the end of the development process to determine whether the right product has been developed.
* Verification techniques can be viewed as an attempt to achieve phase containment of errors. Phase containment of errors has been acknowledged to be a cost-effective way to eliminate program bugs, and is an important software engineering principle. The principle of detecting errors as close to their points of commitment as possible is known as *phase containment of errors*. Phase containment of errors can reduce the effort required for correcting bugs.
* For example, if a design problem is detected in the design phase itself, then the problem can be taken care of much more easily than if the error is identified, say, at the end of the testing phase. In the later case, it would be necessary not only to rework the design, but also to appropriately redo the relevant coding as well as the system testing activities, thereby incurring higher cost.
* We can consider the verification and validation techniques to be different types of bug filters. To achieve high product reliability in a cost-effective manner, a development team needs to perform both verification and validation activities.
* The activities involved in these two types of bug detection techniques together are called the “V and V” activities.
* Error detection techniques = Verification techniques + Validation techniques

**Testing Activities**

Testing involves performing the following major activities:

* **Test suite design:** The test suite 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. The test cases for which the system fails are noted down for later debugging.
* **Locate error:** In this activity, the failure symptoms are analyzed 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.

A typical testing process in terms of the activities that are carried out has been shown schematically in Figure 10.2. As can be seen, the test cases are first designed. Subsequently, 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.



**Why Design Test Cases?**

* Testing a software using a large collection of randomly selected test cases does not guarantee that all (or even most) of the errors in the system will be uncovered. Let us try to understand why the number of random test cases in a test suite, in general, does not indicate of the effectiveness of testing. Consider the following example code segment which determines the greater of two integer values x and y. This code segment has a simple programming error:

if (x>y) max = x;

else max = x; /\* should be max = y \*/

* For the given code segment, the test suite {(x=3,y=2);(x=2,y=3)} can detect the error, whereas a larger test suite {(x=3,y=2);(x=4,y=3);(x=5,y=1)} does not detect the error. All the test cases in the larger test suite essentially test the same statement, while the other statement remains undetected. So, it would be incorrect to say that a larger test suite would always detect more errors than a smaller one, unless of course the larger test suite has also been carefully designed. This implies that for effective testing, the test suite should be carefully designed rather than picked randomly.
* We have already pointed out that exhaustive testing of almost any non-trivial system is impractical due to the fact that the domain of input data values to most practical software systems is extremely large. Therefore, to satisfactorily test a software with minimum cost, we must design a *minimal test suite* that is of reasonable size and can uncover as many existing errors in the system as possible. To reduce testing cost and at the same time to make testing more effective, systematic approaches have been developed to design a small test suite that can detect most, if not all failures.

There are essentially two main approaches to systematically design test cases:

* Black-box approach
* White-box (or glass-box) approach

In the black-box approach, test cases are designed using only the functional specification of the software. That is, test cases are designed solely based on an analysis of the input/ out behavior (that is, functional behavior) and does not require any knowledge of the internal structure or the code of a program. For this reason, black-box testing is also known as *functional testing*.

 On the other hand, designing white-box test cases requires a thorough knowledge of the code structure of a program, and therefore white-box testing is also called *structural testing*. Black-box test cases are designed solely based on the input-output behavior of a program. In contrast, white-box test cases are based on an analysis of the code. These two approaches to test case design are complementary. That is, a program has to be tested using the test cases designed by both the approaches, and one testing using one approach does not substitute testing using the other.

**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. After testing all the units individually, the units are incrementally integrated and tested after each step of integration (integration testing). Finally, the fully integrated system is tested (system testing). Integration and system testing are usually referred to as *testing in the large.*
* Often beginners ask the question—“Why test each module (unit) in isolation first, then integrate these modules and test, and again test the integrated set of modules—why not just test the integrated set of modules once thoroughly?” The answer to this question is the following
* —There are two main reasons to it. First while testing a module, other modules with which this module needs to interface may not be ready. Moreover, it is always a good idea to first test the module in isolation before integration because it makes debugging easier. If a failure is detected when an integrated set of modules is being tested, it would be diﬃcult to determine which module exactly has the error, and the code of a large number of modules may have to be examined to localize the error.
* In the following sections, we discuss the different levels of testing. It should be borne in mind in all our subsequent discussions that unit testing is carried out in the coding phase itself as soon as coding of a module is complete. On the other hand, integration and system testing are carried out during the testing phase.

**UNIT TESTING**

* Unit testing is undertaken after coding of a module is complete, all syntax errors have been removed, and the code has been 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, provide the required global data, 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. The role of stub and driver modules is pictorially shown in Figure 10.3. We briefly discuss the stub and driver modules that are required to provide the necessary environment for carrying out unit testing are briefly discussed in the following.
* **Stub:** A stub module consists of several stub procedures that are called by the module under test. A *stub* procedure is a dummy procedure that takes the same 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, rather than performing actual computations.
* **Driver:** A driver module contains the non-local data structures that are 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.



**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
* **Equivalence Class Partitioning**
* In the equivalence class partitioning approach, the domain of input values to the unit 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.
* 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:

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

* 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 can 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 two invalid equivalence classes are [−∞,0], [11,+∞], and the valid equivalence class is [1,10].
* 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 ∪-*{*A,B,C*}*, where ∪ is the universe of all 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. The reason behind programmers committing such errors might purely be due to psychological factors. Programmers often fail too properly address the special processing required by the input values that lie at the boundary of the different equivalence classes.
* For example, a programmer may improperly use *<* instead of *<*=, or conversely *<*= for *<*, etc.
* 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}.

**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. We first discuss some basic concepts associated with white-box testing, and follow it up with a discussion on specific testing strategies.
* **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. An example of a fault-based strategy is mutation testing, which is discussed later in this section.
* **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.
* For example, if a testing strategy requires all the statements of a program to be executed at least once, then we say that the testing criterion of the strategy is *statement coverage*. We say that a test suite is adequate with respect to a criterion, if it covers all program elements of the domain defined by that criterion.
* **Stronger *versus* weaker testing**
* We have mentioned that a large number of white-box testing strategies have been proposed. It therefore becomes necessary to compare the effectiveness of different testing strategies in detecting faults. We can compare two testing strategies by determining whether one is stronger, weaker, or complementary to the other.
* 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*.

 

**Statement Coverage**

* Statement coverage is a metric to measure the percentage of statements that are executed by a test suite in a program at least once.
* It is obvious that without executing a statement, it is diﬃcult to determine whether it causes a failure due to illegal memory access, wrong result computation due to improper arithmetic operation, etc. It must however be pointed out that an important weakness of the statement coverage strategy is that executing a statement once and observing that it behaves properly for one input value is no guarantee that it will behave correctly for all input values.
* Never the less, statement coverage is a very intuitive and appealing testing technique. In the following, we illustrate a test suite that achieves statement coverage.
* Design a statement coverage-based test suite for the following Euclid’s GCD computation function:

int computeGCD(int x,int y){

 while (x != y){

 if (x>y) then

 x=x-y;

 else y=y-x;

}

return x;

}

***Solution:*** To design the test cases for achieving 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**

* Branch coverage is also called *decision coverage* (DC). It is also sometimes referred to as *all edge coverage*. A test suite achieves branch coverage, if it makes the decision expression in each branch in the program to assume both true and false values. 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 all *edge testing*, since in this testing scheme, each edge of a program’s control flow graph is required to be traversed at least once.
* For the program of Problem 10.13, determine a test suite to achievebranch coverage.
* ***Solution:*** 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*.
* Branch coverage-based testing is stronger than statement coverage-based testing.
* *Proof:* We need to show that (a) branch coverage ensures statement coverage, and (b) statement coverage does not ensure branch coverage.
* (a) Branch testing would guarantee statement coverage since every statement must belong to some branch (assuming that there is no unreachable code).
* (b) To show that statement coverage does not ensure branch coverage, it is suﬃcient to give an example of a test suite that achieves statement coverage, but does not cover at least one branch. Consider the following code, and the test suite {5}.
* if(x>2) x+=1;
* The test suite would achieve statement coverage. However, it does not achieve branch coverage, since the condition (*x >* 2) is not made false by any test case in the suite.

**Condition Coverage**

* Condition coverage testing is also known as *basic condition coverage* (BCC) *testing*. A test suite is said to achieve basic condition coverage (BCC), if each basic condition in every conditional expression assumes both true and false values during testing. For example, for the following decision statement: if(A||B && C) …; the basic conditions A, B, and C assume both true and false values. However, for the given expression, just two test cases can achieve condition coverage. For example, one test case, may assign A = True, B = True, and C = True and another test case may assign A = False, B = False, and C = False. It is easy to see that basic condition coverage may not achieve branch coverage.

**Condition and Decision Coverage**

A test suite is said to achieve condition and decision coverage, if it achieves condition coverage as well as decision (that is, branch) coverage. Obviously, condition and decision coverage is stronger than both condition coverage and decision coverage

**Multiple Condition Coverage**

* *Multiple condition coverage* (MCC) is achieved, if the test cases make the component conditions of a composite conditional expression to assume all possible combinations of true and false values. For example, consider the composite conditional expression [(*c*1 *and c*2) *or c*3]. A test suite would achieve MCC, if all the component conditions *c*1, *c*2, and *c*3 are each made to assume all combinations of true and false values. Therefore, at least eight test cases would be required in this case to achieve MCC.
* It is easy to prove that condition testing is a stronger testing strategy than branch testing. For a composite conditional expression of *n* components, 2*n* 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 atomic conditions in the decision expression) is small.

**Path Coverage**

* A test suite achieves path coverage if it exeutes each *linearly independent paths* (or *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,

**Control flow graph (CFG)**

* A control flow graph describes how the control flows through the program. 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). 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.
* 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 10.8(a) can be drawn as shown in Figure 10.8(b).



**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. Therefore, we can say that *all path* testing is impractical. For this reason, path coverage testing does not try to cover all paths, but only a subset of paths called *linearly independent paths* (or *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. Please note that even if we find that a path has one new node compared to all other linearly independent paths, then this path should also be included in the set of linearly independent paths. This is because, any path having a new node would automatically have a new edge.
* According to the above definition of a linearly independent set of paths, for any path in the set, its sub path cannot be a member of the set. In fact, any arbitrary path of a program, can be synthesized by carrying out linear operations on the basis paths. Possibly, the name basis set comes from the observation that the paths in the basis set form the “basis” for all the paths of a program. Please note that there may not always exist a unique basis set for a program and several basis sets for the same program can usually be determined.
* Even though it is straight forward to identify the linearly independent paths for simple programs, for more complex programs it is not easy to determine the number of independent paths. In this context, McCabe’s cyclomatic complexity metric is an important result that lets us compute the number of linearly independent paths for any arbitrary program. McCabe’s cyclomatic complexity defines an upper bound for the number of linearly independent paths through a program. Also, the McCabe’s cyclomatic complexity is very simple to compute. Though the McCabe’s metric does not directly identify the linearly independent paths, but it provides us with a practical way of determining approximately how many paths to look for.

**McCabe’s Cyclomatic Complexity Metric**

* McCabe [1976] obtained his results by applying graph-theoretic techniques to the control flow of a program. McCabe’s cyclomatic complexity metric 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 1:** 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 the example program shown in Figure 10.8, *E* = 7 and *N* = 6. Therefore, the value of the Cyclomatic complexity = 7 − 6 + 2 = 3.
* **Method 2:** An alternate way of computing the cyclomatic complexity of a program is based on a visual inspection of the control flow graph is as follows—In this method, the cyclomatic complexity *V* (*G*) for a graph *G* is given by the following expression:
* V(G) = Total number of non-overlapping bounded areas + 1
* In the program’s control flow graph *G*, any region enclosed by nodes and edges can be called as a *bounded area*. This is an easy way to determine the McCabe’s cyclomatic complexity. But what if the graph *G* is not planar (i.e., however you draw the graph, two or more edges intersect). Actually, it can be shown that control flow representation of structured programs always yields planar graphs. But, presence of GOTO’s can easily add intersecting edges and make the CFG non-planar. Therefore, for non-structured programs, this way of computing the McCabe’s cyclomatic complexity does not apply.
* The number of bounded areas in a CFG increases with the number of decision statements and loops. Therefore, the McCabe’s metric provides a quantitative measure of the testing diﬃculty of a program and its ultimate reliability after testing. Consider the CFG example shown in Figure 10.8. From a visual examination of the CFG the number of bounded areas is 2. Therefore, the cyclomatic complexity, computed with this method is also 2+1=3. This method provides a very easy way of computing the cyclomatic complexity of CFGs, just from a visual examination of the CFG. On the other hand, the method for computing CFGs can easily be automated. That is, the McCabe’s metric computations methods 1 and 3 can be easily coded into a tool that can be used to automatically determine the cyclomatic complexities of arbitrary programs.
* **Method 3:** The cyclomatic complexity of a program can also be easily computed by computing the number of decision and loop statements of the program. If *N* is the number of decision and loop statements of a program, then the McCabe’s metric is equal to *N* + 1.

**How is path testing carried out by using computed McCabe’s cyclomatic metric value?**

* Knowing the number of basis paths in a program does not make it any easier to design test cases for path coverage, only it gives an indication of the minimum number of test cases required for path coverage. For the CFG of a moderately complex program segment of say 20 nodes and 25 edges, you may need several days of effort to identify all the linearly independent paths in it and to design the test cases. It is therefore impractical to require the test designers to identify all the linearly independent paths in a code, and then design the test cases to force execution along each of the identified paths. In practice, for path testing, usually the tester keeps on forming test cases with random data and executes those until the required coverage is achieved. A testing tool such as a dynamic program analyzer (see Section 10.8.2) is used to determine the percentage of linearly independent paths covered by the test cases that have been executed so far.
* If the percentage of linearly independent paths covered is below 90 per cent, more test cases (with random inputs) are added to increase the path coverage. Normally, it is not practical to target achievement of 100 per cent path coverage. The first reason is the presence of infeasible paths. Though the percentage of infeasible programs varies across programs, it is often in the range of 1–10%. An example of an infeasible path is the following:

 if(x==1) {…}

 if (x==2) {…}

* In the above code segment, it is not possible to have both the conditional expressions as true. Also, McCabe’s metric is only an upper bound and does not give the exact number of paths.

**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.

**Uses of McCabe’s cyclomatic complexity metric**

* Beside its use in path testing, cyclomatic complexity of programs has many other interesting applications such as the following:
* **Estimation of structural complexity of code:** McCabe’s cyclomatic complexity is a measure of the *structural complexity* of a program. The reason for this is that it is computed based on the code structure (number of decision and iteration constructs used). Intuitively, the McCabe’s complexity metric correlates with the diﬃculty level of understanding a program, since one understands a program by understanding the computations carried out along all independent paths of the program.
* In view of the above result, from the maintenance perspective, it makes good sense to limit the cyclomatic complexity of all functions to some reasonable value. Good software development organisations usually restrict the cyclomatic complexity of different functions to a maximum value of ten or so. This is in contrast to the computational complexity that is based on the execution of the program statements.
* **Estimation of testing effort:** Cyclomatic complexity is a measure of the maximum number of basis paths. Thus, it indicates the minimum number of test cases required to achieve path coverage. Therefore, the testing effort and the time required to test a piece of code satisfactorily is proportional to the cyclomatic complexity of the code. To reduce testing effort, it is considered necessary to restrict the cyclomatic complexity of every function to seven or so.

**Estimation of program reliability:** Experimental studies indicate there exists a clear relationship between the McCabe’s metric and the number of errors latent in the code after testing. This relationship exists possibly due to the correlation of cyclomatic complexity with the structural complexity of code. Usually, the larger is the structural complexity, the more diﬃcult it is to test and debug the code

**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 summaries 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, were the developer mentally computes the expected result after every source instruction and checks whether the same is actually computed by a statement 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, in the presence of decision statements and loops, this approach becomes cumbersome 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 identified 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

**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 mistakes that novice programmers often make is attempting not to fix the error but its symptoms.
* One must be beware of the possibility that an error correction may introduce new errors. Therefore, after every round of error-fixing, regression testing (see Section 10.13) 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). 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.
* 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. Any one (or a mixture) of the following approaches can be used to develop the test plan:
* Big-bang approach to integration testing
* Top-down approach to integration testing
* Bottom-up approach to integration testing
* Mixed (also called *sandwiched)* approach to integration testing

In the following subsections, we provide an overview of these approaches 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 localize the error as the error may potentially exist 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 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 usually implies single step integration. In contrast, in the other strategies, integration is carried out over a number of steps. In multi-step integration strategies, modules are 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 in each step of integration.
*  In phased integration, a group of related modules are added to the partial system in each step of integration.
* 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.

**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 minimized.
* However, it was soon realized 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.

**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
* A method operates in the scope of the data and other methods of its object. That is, all the methods share the data of the class. Therefore, it is necessary to test a method in the context of these. Moreover, objects can have significant number of states. The behavior of a method can be different based on the state of the corresponding object. Therefore, it is not enough to test all the methods and check whether they can be integrated satisfactorily.
* A method has to be tested with all the other methods and data of the corresponding object. Moreover, a method needs to be tested at all the states that the object can assume. As a result, it is improper to consider a method as the basic unit of testing an object-oriented program.
* Thus, in an object-oriented program, unit testing would mean testing each object in isolation. During integration testing (called *cluster testing* in the object-oriented testing literature) various unit tested objects are integrated and tested. Finally, system-level testing is carried out.

**Do Various Object-orientation Features Make Testing Easy?**

* **Encapsulation:** Encapsulation feature helps in data abstraction, error isolation, and error prevention. However, as far as testing is concerned, encapsulation is not an obstacle to testing, but leads to diﬃculty during debugging. Encapsulation prevents the tester from accessing the data internal to an object. Of course, it is possible that one can require classes to support state reporting methods to print out all the data internal to an object. Thus, 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 incrementally testing of the added features. However, this is not the case.
* The reason for this is that the inherited methods would work in a new context (new data and method definitions).As a result, correct behavior of a method at an upper level, does not guarantee correct behavior at a lower level. Therefore, retesting of inherited methods needs tube 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. For example, when a book has been issued out in a library information system, the book reaches the issuedOut state.

* In this state, if the issue method is invoked, then it may not exhibit its normal behaviour. In view of the discussions above, testing an object in only one of its states is not enough. The object has to be tested at all its possible states. Also, whether all the transitions between states (as specified in the object model) function properly or not should be tested.
* Additionally, it needs to be tested that no extra (sneak) transitions exist, neither are there extra states present other than those defined in the state model. For state-based testing, it is therefore beneficial to have the state model of the objects, so that the conformance of the object to its state model can be tested.

**Why are Traditional Techniques Considered Not Satisfactory for Testing Object-oriented Programs?**

* We have already seen that in traditional procedural programs, procedures are the basic unit of testing. In contrast, objects are the basic unit of testing for object-oriented programs. Besides this, there are many other significant differences as well between testing procedural and object-oriented programs. For example, statement coverage-based testing which is popular for testing procedural programs is not satisfactory for object-oriented programs.
* The reason is that inherited methods have to be retested in the derived class. Various object-oriented features (inheritance, polymorphism, dynamic binding, state-based behavior, etc.) require special test cases to be designed compared to the traditional testing
* The various object-orientation features are explicit in the design models, and it is usually diﬃcult to extract from and analysis of the source code. As a result, the design model is a valuable artifact for designing test cases for object-oriented programs. Therefore, this approach is considered to be intermediate between a fully white-box and a fully black-box approach, and is called a *grey-box* approach.
* Please note that grey-box testing is considered important for object-oriented programs. This is in contrast to testing procedural programs.

**Grey-Box Testing of Object-oriented Programs**

* As we have already mentioned, model-based testing is important for object-oriented programs, as these test cases help detect bugs that are specific to the object-orientation constructs.
* 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 a base class have to be instantiated and tested. In addition to testing the new methods defined in the derived class, 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. Each sequence diagram represents the message passing among objects that occurs for each use case. Each use case consists of a set of scenarios, and a message path represents the message exchanges that occur among concerned objects during execution of a s

**Integration Testing of Object-oriented Programs**

* In a procedural program, the module structure is typically represented using a structure chart, which is a rooted tree. Therefore, top-down, bottom-up, and mixed integration strategies are applicable However, in case of object-oriented programs, the class structure is an arbitrary graph and is not restricted to a tree structure. Therefore, the integration testing strategies for procedural programs are difficult to apply for object-oriented programs. Two approaches to integration testing of object-oriented programs have become popular:
* Thread-based
* Use based
* **Thread-based approach:** In this approach, all classes that need to collaborate to realise the behavior 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 do not need any 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.

**SYSTEM TESTING**

After all the units of a program have been integrated together and tested, system testing is taken up. 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 three main kinds of system testing. These are essentially similar tests, but differ in who carries out the 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. As can be observed from the above discussions, in the different types of system tests, the test cases can be the same, but the difference is with respect to who designs test cases and carries out testing.

Before a fully integrated system is accepted for system testing, *smoke testing* is performed. Smoke testing is done to check whether at least the main functionalities of the software are working properly. Unless the software is stable and at least the main functionalities are working satisfactorily, system testing is not undertaken.

The functionality tests are designed to check whether the software satisfies the functional requirements as documented in the SRS document. The performance tests, on the other hand, test the conformance of the system with the non-functional requirements of the system. We have already discussed how to design the functionality test cases by using a black-box approach

**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. 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. Input data volume, input data rate, processing time, utilisation of memory, etc., are tested beyond the designed capacity. For example, suppose an operating system is supposed to support fifteen concurrent transactions, then the system is stressed by attempting to initiate fifteen or more transactions simultaneously. A real-time system might be tested to determine the effect of simultaneous arrival of several high-priority interrupts.
* Stress testing is especially important for systems that under normal circumstances operate below their maximum capacity but may be severely stressed at some peak demand hours. For example, if the corresponding non-functional requirement states that the response time should not be more than twenty secs per transaction when sixty concurrent users are working, then during stress testing the response time is checked with exactly sixty users working simultaneously.
* **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 behavior 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 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.
* **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**

These 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.
* **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.