Coding Standards and Guidelines

- **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.
- Over the years, the general perception of testing as **monkeys** typing in random data and trying to crash the system has changed. Now testers are looked upon as masters of specialised concepts, techniques, and tools.

CODING: 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.

Coding Standards and Guidelines

- Normally, good software development organisations require their programmers to adhere to some well-defined and standard style of coding which is called their **coding standard**.
- It is mandatory for the programmers to follow the coding standards.
- Compliance of their code to coding standards is verified during code inspection. Any code that does not conform to the coding standards is **rejected** during code review and the code is reworked by the concerned programmer.
- Good software development organisations 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.

Coding Standards

1. **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.
2. **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:
   a. Name of the module.
   b. Date on which the module was created.
   c. Author’s name.
   d. Modification history.
   e. Synopsis of the module.
   f. Different functions supported in the module, along with their input/output parameters.
   g. Global variables accessed/modified by the module
3. **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).

4. **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.

Coding Guidelines

1. **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.
2. **Avoid obscure side effects**: The side effects of a function call include modifications to the parameters passed by reference, modification of global variables, and I/O operations. An obscure side effect is one that is not obvious from a casual examination of the code.
3. **Do not use an identifier for multiple purposes**: Programmers often use the same identifier to denote several temporary entities.
4. Each variable should be given a descriptive name indicating its purpose.
5. Use of variables for multiple purposes usually makes future enhancements more difficult.
6. **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.
7. **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.
8. **Does 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

- 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.
- Testing is an effective defect removal mechanism. However, testing is applicable to only **executable code**.
- The reason behind why code review is a much more **cost-effective** strategy to eliminate errors from code compared to testing is that reviews directly **detect errors**. On the other hand, testing only helps detect **failures** and significant effort is needed to locate the error during debugging.
- Normally, the following two types of reviews are carried out on the code of a module:
    - Code walkthrough.
    - Code inspection.

Code walkthrough:

1. The main objective of code walkthrough is to discover the algorithmic and logical errors in the code.
2. Code walkthrough is an informal code analysis technique.
3. 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.

4. A few members of the development team are given the code a couple of days before the walkthrough meeting.
5. Each member selects some test cases and simulates execution of the code by hand (i.e., traces the execution through different statements and functions of the code).
6. The members note down their findings of their walkthrough and discuss those in a walkthrough meeting where the coder of the module is present.

Code Inspection:

1. The principal aim of code inspection is to check for the presence of some common types of errors that usually creep into code due to programmer mistakes and oversights and to check whether coding standards have been adhered to.
2. The programmer usually receives feedback on programming style, choice of algorithm, and programming techniques.

Following is a list of some classical programming errors which can be checked 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.
- ✓ Use of incorrect logical operators or incorrect precedence among operators.
- ✓ Dangling reference caused when the referenced memory has not been allocated.

SOFTWARE DOCUMENTATION

When software is developed, in addition to the executable files and the source code, several kinds of documents such as users' manual, software requirements specification (SRS) document, design document, test document, installation manual, etc., are developed as part of the software engineering process. All these documents are considered a vital part of any good software development practice. Good documents are helpful in the following ways:

- Good documents help enhance understandability of code
- Documents help the users to understand and effectively use the system.
- Good documents help to effectively tackle the manpower turnover1 problem.
- Production of good documents helps the manager to effectively track the progress of the project.

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

Internal documentation:

1. These are provided in the source code itself.
2. Internal documentation can be provided in the code in several forms.
3. The important types of internal documentation are the following:
    a. Comments embedded in the source code.
    b. Use of meaningful variable names.
    c. Module and function headers.
    d. Code indentation.
    e. Code structuring (i.e., code decomposed into modules and functions).
    f. Use of enumerated types.
    g. Use of constant identifiers.
    h. Use of user-defined data types

External documentation: These are the supporting documents such as SRS document, installation document, user manual, design document, and test document

Gunning's fog index:

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 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.

$$\text{fog}(D) = 0.4 \times \left(\frac{\text{words}}{\text{sentences}}\right) + \text{per cent of words having 3 or more syllables}$$

Example 10.1 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. The fog index of the above example sentence is

0.4 □ (23/1) + (4/23) □ 100 = 26

If a users' manual is to be designed for use by factory workers whose 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

Definition: 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.

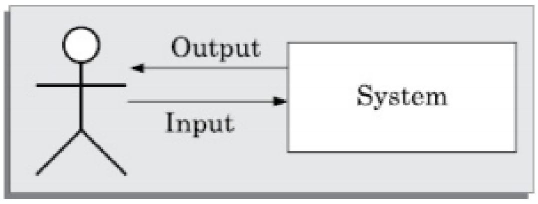


Figure 10.1: A simplified view of program testing.

Terminologies

Few important terminologies that have been standardised by the IEEE Standard Glossary of Software Engineering Terminology [IEEE90]:

1. **Mistake**: A mistake is essentially any programmer action that later shows up as an incorrect result during program execution. A programmer may commit a mistake in almost any development activity.
   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.
2. **Error:** An error is the result of a mistake committed by a developer in any of the development activities. 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
3. **Failure:** A failure of a program essentially denotes an incorrect behaviour exhibited by the program during its execution. Every failure is caused by some bugs present in the program.
   Example: A program crashes on an input.

Note: It may be noted that mere presence of an error in a program code may not necessarily lead to a failure during its execution.

```
int markList[1:10]; /* mark list of 10 students*/
int roll;           /* student roll number*/
       ...
if(roll>0)
       markList[roll]=mark;
else
       markList[roll]=0;
```

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.

4. **Test Case**: A test case is a specification of the inputs, execution conditions, testing procedure, and expected results that define a single test to be executed to achieve a particular software testing objective.
5. **Test suite:** A test suite is the set of all test that have been designed by a tester to test a given program.
6. **Testability:** 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.

Verification vs Validation

Barry Boehm described verification and validation as the following:

- ✓ **Verification:** Are we building the product right?
- ✓ **Validation:** Are we building the right product?

1. **Verification:**
   Verification is the process of checking that a software achieves its goal without any bugs. It is the process to ensure whether the product that is developed is right or not. It verifies whether the developed product fulfils the requirements that we have. Verification is **Static Testing**.
   Activities involved in verification:

   - Inspections

- Reviews
- Walkthroughs

2. **Validation**:
   Validation is the process of checking whether the software product is up to the mark or in other words product has high level requirements. It is the process of checking the validation of product i.e. it checks what we are developing is the right product. it is validation of actual and expected product. Validation is the **Dynamic Testing**.

   Activities involved in validation:

   1. Black box testing
   2. White box testing
   3. Unit testing
   4. Integration testing

   Error detection techniques = Verification techniques + Validation techniques

Testing Activities

1. Test suite design
2. Running test cases and checking the results to detect failures
3. Locate error
4. Error correction

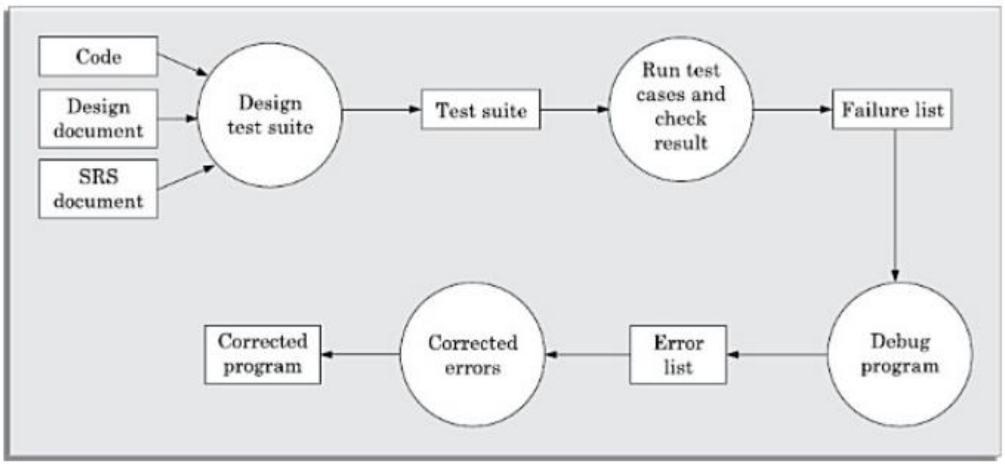


Figure 10.2: Testing process.

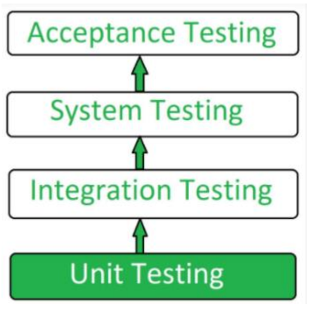
Levels of Testing

A software product is normally tested in three levels or stages

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

1. **Unit testing**: During unit testing, the individual functions (or units) of a program are tested.
2. **Integration testing:** After testing all the units individually, the units are slowly integrated and tested after each step of integration (integration testing).
3. **System testing:** Finally, the fully integrated system is tested (system testing).

Unit Testing



Unit testing is defined as a type of software testing where individual components of software are tested.

Unit testing of software product is carried out during the development of an application. An individual component may be either an individual function or a procedure. Unit testing is typically performed by the developer.

Objective of Unit Testing:

1. To isolate a section of code.
2. To verify the correctness of code.
3. To test every function and procedure.
4. To fix bug early in development cycle and to save costs

Integration testing

Integration testing is the process of testing the interface between two software units or module. It's focus on determining the correctness of the interface. There are four types of integration testing approaches. Those approaches are the following:

a. **Big-Bang Integration Testing**: It is the simplest integration testing approach, where all the modules are combining and verifying the functionality after the completion of individual module testing.
b. **Bottom-Up Integration Testing**: In bottom-up testing, each module at lower levels is tested with higher modules until all modules are tested.
c. **Top-Down Integration Testing**: Top-down integration testing technique used in order to simulate the behavior of the lower-level modules that are not yet integrated.
d. **Mixed Integration Testing:** A mixed integration testing is also called **sandwiched** integration testing. A mixed integration testing follows a combination of top down and bottom-up testing approaches

System Testing

System Testing is carried out on the whole system in the context of either system requirement specifications or functional requirement specifications or in the context of both. System testing tests the design and behavior of the system and also the expectations of the customer. It is performed to test the system beyond the bounds mentioned in the software requirements specification (SRS).

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 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.

Example 1 : For a software that computes the square root of an input integer that can assume values in the range of 0 and 5000. Determine the equivalence classes and the black box 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. Therefore, the test cases must include representatives for each of the three equivalence classes. A possible test suite can be: **{–5,500,6000**}.

Example 2: Design equivalence class partitioning test suite for a function that reads a character string of size less than five characters and displays whether it is a palindrome.

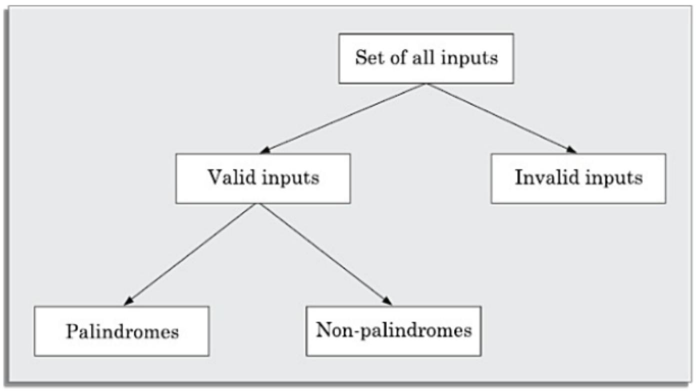


Figure 10.4: Equivalence classes for Example 10.6.

Answer: The equivalence classes are the leaf level classes shown in Figure 10.4. The equivalence classes are palindromes, non-palindromes, and invalid inputs. Now, selecting one representative value from each equivalence class, we have the required test suite: **{abc,aba,abcdef}.**

❖ Boundary Value Analysis

Boundary value analysis-based test suite design involves designing test cases using the values at the boundaries of different equivalence classes.

Example, programmers may improperly use < instead of <=, or conversely <= for <, etc.

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}.**

Important steps in the black-box test suite design approach:

1. Examine the input and output values of the program.
2. Identify the equivalence classes.
3. Design equivalence class test cases by picking one representative value from each equivalence class.
4. Design the boundary value test cases as follows. Examine if any equivalence class is a range of values. Include the values at the boundaries of such equivalence classes in the test suite.

WHITE-BOX TESTING

White-box testing is an important type of unit testing. A large number of white-box testing strategies exist. A white-box testing strategy can either be (i) **coverage**-based or (ii) **fault** based**.**

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.

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. An example of a fault-based strategy is **mutation** testing (testers change specific components of an application's source code to ensure a software test suite will be able to detect the changes).

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.

- ❖ **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.
- ❖ If a stronger testing has been performed, then a weaker testing need not be carried out.

COVERAGE-BASED TESTING

Statement Coverage: The statement coverage strategy aims to design test cases so as to execute every statement in a program at least once.

Example Design statement coverage-based test suite for the following Euclid's GCD computation program:

```
int computeGCD(x,y)
   int x,y;
{
    while (x != y){
    if (x>y) then
    x=x-y;
    else y=y-x;
    }
    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.

Example 2: Design branch coverage -based test suite for the following Euclid's GCD computation program

Answer: The test suite {(x = 3, y = 3), (x = 3, y = 2), (x = 4, y = 3), (x =3, y = 4)} achieves branch coverage.

Note: Branch coverage-based testing is stronger than statement coverage-based testing.

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.

Consider the following C program segment:

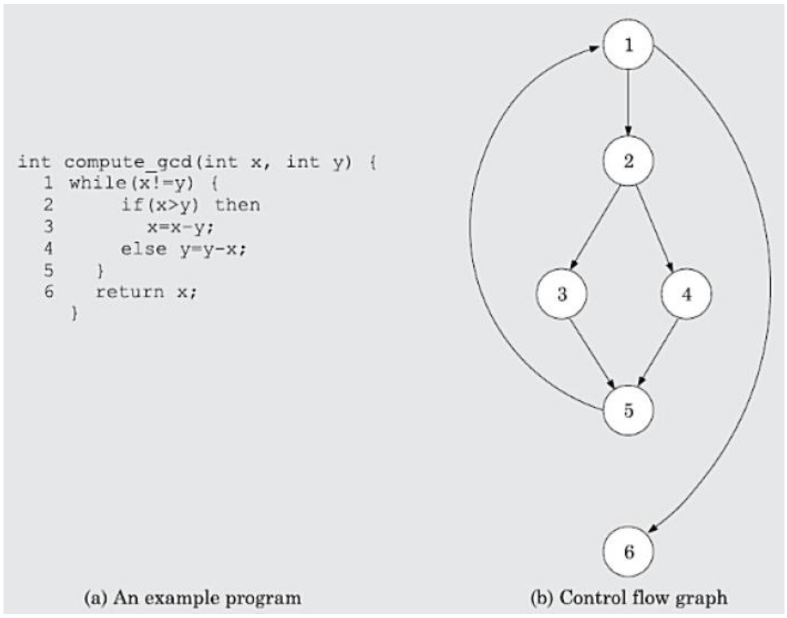
```
if(temperature>150 || temperature>50)
  setWarningLightOn();
```

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.

Control flow graph (CFG) : A control flow graph describes the sequence in which the different instructions of a program get executed. 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.



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

(a) An example program (b) Control flow graph

McCabe's Cyclomatic Complexity Metric:

- Cyclomatic complexity Metric is the quantitative measure of the number of linearly independent paths in it.
- Cyclomatic complexity of a program is a measure of the psychological complexity or the level of difficulty in understanding the program.
- It is a software metric used to indicate the complexity of a program.
- It is computed using the Control Flow Graph of the program.
- The nodes in the graph indicate the smallest group of commands of a program, and a directed edge in it connects the two nodes i.e. if second command might immediately follow the first command.
- McCabe's cyclomatic complexity defines an upper bound on the number of independent paths in a program.

For example, if source code contains no control flow statement then its cyclomatic complexity will be 1 and source code contains a single path in it.

Similarly, if the source code contains one **if condition** then cyclomatic complexity will be 2 because there will be two paths one for true and the other for false.

There are three different ways to compute the cyclomatic complexity.

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 example shown in the above Figure 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 aprogram 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

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.**

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**.

FAULT-BASED TESTING

Mutation Testing: Mutation test cases are designed to help detect specific types of faults in a program. 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.**

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.

DEBUGGING

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

Debugging Approaches

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.

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.

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.

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. We can classify various program analysis tools into the following two broad categories:

1. Static analysis tools
2. Dynamic analysis tools

Static Analysis Tools:

Static program analysis tools assess and compute various characteristics of a program without executing it.

Typically, static analysis tools analyse the **source code** to compute certain metrics characterising 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 uninitialised 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.

For example, the dynamic analysis tool can report the statement, branch, and path coverage achieved by a test suite. If the coverage achieved is not satisfactory more test cases can be designed, added to the test suite, and run. Further, dynamic analysis results can help eliminate redundant test cases from a test suite.

SYSTEM TESTING

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.

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 organisation.

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.

Functionality and Performance test cases

The system test cases can be classified into **functionality** and **performance** test cases. 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.

Smoke 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.

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

1. 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. All performance tests can be considered as **black-box** tests.
2. **Stress testing**: Stress testing is also known as endurance testing. Stress testing evaluates system performance when it is stressed for short periods of time.
   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.
3. **Volume testing:** Volume testing checks whether the data structures (buffers, arrays, queues, stacks, etc.) have been designed to successfully handle extraordinary situations.
4. **Configuration testing:** Configuration testing is used to test system behaviour in various hardware and software configurations specified in the requirements.
5. **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.
6. **Regression testing:** This type of testing is required when a software is maintained to fix some bugs or enhance functionality, performance, etc.
7. **Recovery testing**: Recovery testing tests the response of the system to the presence of faults, or loss of power, devices, services, data, etc.
8. **Maintenance testing:** This addresses testing the diagnostic programs, and other procedures that are required to help maintenance of the system.
9. **Documentation testing**: It is checked whether the required user manual, maintenance manuals, and technical manuals exist and are consistent.

10. **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.
11. **Security testing:** Security testing is essential for software that handle or process confidential data that is to be gurarded against stealing.

Regression Testing

Regression Testing is the process of testing the modified parts of the code and the parts that might get affected due to the modifications to ensure that no new errors have been introduced in the software after the modifications have been made.

When to do regression testing?

1. When a new functionality is added to the system and the code has been modified to absorb and integrate that functionality with the existing code.
2. When some defect has been identified in the software and the code is debugged to fix it.
3. When the code is modified to optimize it's working.

Process of Regression testing:

1. Firstly, whenever we make some changes to the source code for any reasons like adding new functionality, optimization, etc. then our program when executed fails in the previously designed test suite for obvious reasons.
2. After the failure, the source code is debugged in order to identify the bugs in the program.
3. After identification of the bugs in the source code, appropriate modifications are made.
4. Then appropriate test cases are selected from the already existing test suite which covers all the modified and affected parts of the source code.
5. We can add new test cases if required. In the end regression testing is performed using the selected test cases.

TESTING OBJECT-ORIENTED PROGRAMS

UNIT TESTING

- Traditional Techniques Considered **Not** Satisfactory for Testing Object-oriented Programs.
- Adequate testing of individual **methods does not** ensure that a class has been satisfactorily tested.
- An **object** is the basic unit of testing of object-oriented programs.

Grey-Box Testing of Object-oriented Programs

1. **Model-based** testing is important for object oriented programs,
2. 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.
3. 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 is 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 derived . c lass, the inherited methods must be retested.
- **Association testing**: All association relations are tested.
- **Aggregation testing**: Various aggregate objects are created and tested.

INTEGRATION TESTING

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.

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.