Unit V: Coding & Testing

Coding & Code Review:

- 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 it's coding is complete.
- After all the modules of a system have been **coded and unit tested**, the **integration and system testing phase is undertaken.**
- **Integration and testing of modules** is carried out according to an **integration plan**. The integration plan, according to which **different modules are integrated together** 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.**
- Testing of professional software is carried out using a large number of **test cases**.

5.1 CODING

- The input to the coding phase is the **design document** produced at the end of the design phase.
- Design document contains not only the **high-level design** of the system in the form of a module structure (e.g., a structure chart), but also the **detailed design**.
- The **detailed design** is usually documented in the form of module specifications where the **data structures and algorithms** for each **module** are specified.
- During the **coding phase**, different **modules** identified in the **design document** are **coded** according to their respective **module specifications**.
- The objective of the coding phase is to **transform the design of a system** into **code** in a **high-level language**, and then to **unit test this code**.
- **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; way variables are to be named, the way the code is to be laid out, the error return conventions,

Coding standard vs coding guideline

- It is mandatory for the **programmers to follow the coding standards.**
- **Compliance** of their code to **coding standards** is verified during **code inspection.**
- **Any** code that does not conform to the coding standards is rejected **during code review** and the code is reworked by the concerned programmer.
- In contrast, **coding guidelines** provide some ***general suggestions*** regarding the coding style to be followed but leave the **actual implementation of** these guidelines to the discretion of the individual developers.

5.1.1 Coding Standards and Guidelines

General coding standards and guidelines that are commonly adopted by many software development organisations, rather than trying to provide an exhaustive list.

A.Representative coding standards

1. **Rules for limiting the use of globals**: These rules list **what types of data** can be **declared global and what cannot**, with a view to **limit the data** that needs to be defined with global scope.

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:

1. Name of the module.
2. Date on which the module was created.
3. Author's name.
4. Modification history.
5. Synopsis of the module. This is a small writeup about what the module does.
6. Different functions supported in the module, along with their
7. input/output parameters.
8. 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 organisation.
- For example, all functions while **encountering an error condition** should either return a **0 or 1**
- Consistently, independent of which programmer has written the code. This facilitates **reuse and debugging.**

B. Representative coding guidelines:

The following are some representative coding guidelines that are recommended by many software development organisations.

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 a 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. Obscure side effects make it difficult to understand a piece of code.
- For example, suppose the value of a **global variable** is changed or some file I/O is performed obscurely in a called module. That is, this is difficult to infer from the function's name and header information. Then, it would be really hard to understand the code.

3. 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.

4. 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.

5. 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.

6. Do not use GO TO statements:

- Use of GO TO statements makes u**nstructured**. This makes the program very difficult to understand, debug, and maintain.

5.2 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.
- Code review does not target to **design syntax errors** in a program, but is designed to detect **logical, algorithmic, and programming errors**.
- Code review is a much more **cost-effective strategy** to **eliminate errors** from code compared to testing is that reviews directly detect errors.
- Eliminating **an error from code** involves three main activities—testing [**to detect if the system fails to work satisfactorily**] , debugging[**Locate the error that is causing the failure and to remove it**]., and then **correcting the errors.**

  Following two types of reviews are carried out on the code of a module:
  - Code inspection.
  - Code walkthrough.

5.2.1 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.
- The main objective of code walkthrough is to **discover the algorithmic and logical errors in the code.**

Some of these guidelines are following:

- The **team performing code walkthrough** should not be either **too big or too small.**
- it should consist of between three to seven **members.**
- Discussions should focus on **discovery of errors** and avoid **deliberations on how to fix the discovered errors**.
- In order to foster 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**.

5.2.2 Code Inspection

- During code inspection, the code is **examined for the presence of some common programming errors.**
- The aim of CI 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.

Following is a list of some **classical programming errors** which can be checked during **code inspection:**

1. Use of uninitialised variables.
2. Jumps into loops.
3. Non-terminating loops.
4. Incompatible assignments.
5. Array indices out of bounds.
6. Improper storage allocation and deallocation.
7. Mismatch between actual and formal parameter in procedure calls.
8. Use of incorrect logical operators or incorrect precedence among operators.
9. Improper modification of loop variables.
10. Comparison of equality of floating point values.
11. Dangling reference caused when the referenced memory has not been allocated

5.2.3 Clean Room Testing:

- Clean room testing was pioneered at IBM.
- This type of testing relies heavily on **walkthroughs, inspection, and formal verification.**
- This technique reportedly produces **documentation and code** that is more reliable and maintainable than other development methods relying heavily on code execution-based testing.

5.3 TESTING

The aim of program testing is to help realiseidentify all defects in a program.

5.3.1 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 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 5.1.

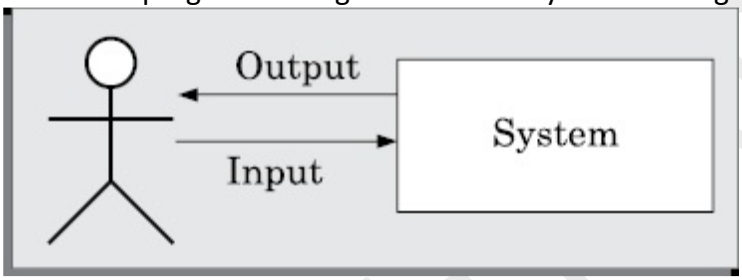

Figure 5.1: A simplified view of program testing.

- The tester has been shown as a stick icon, who inputs several test data to the system and observes the outputs produced by it to check if the system fails on some specific inputs.

Terminologies

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

1. **A mistake**
   - is essentially any programmer action that later shows up as an **incorrect result** during program execution.
   - For example,during coding a programmer might commit the mistake of **not initializing a certain variable**

2**. 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. A failure
   - of a program essentially denotes an **incorrect behaviour** exhibited by the program during its execution.
   - An incorrect behaviour is observed either as **an incorrect result** produced or as an **inappropriate activity** carried out by the program.
   - Every failure is caused by some bugs present in the program.

- **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.

4. A test case

- is a triplet [I , S, R], where I is the **data input** to the **program under test**, S is the state of the program at which the data is to be input, and R is the result expected to be produced by the program.
- A n example of a test case is—[input: "abc", state: edit, result: abc is displayed], which essentially means that the input abc needs to be applied in the edit mode, and the expected result is that the string a b c would be displayed.

5. A test scenario is an abstract test case in the sense that it only

- Identifies the aspects of the program that are to be tested **without identifying the input, state, or output**.
- A test case can be said to be an **implementation of a test scenario**. In the test case, the input, output, and the state at which the input would be applied is designed such that the scenario can be executed.
- An important automatic test case design strategy is to first design test scenarios through an analysis of some program abstraction (model) and then implement the test scenarios as test cases.

6.A test script

- is an encoding of a **test case** as a short program.
- Test scripts are developed for automated execution of the test cases.
- There are positive as well as negative test cases

7. **A test suite**

- is **the set of all test** that have been designed by a tester to test a given program.

8**. Testability**

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

Verification versus validation

Sr.no	Verification	validation
1.	process of determining whether the output of one phase of software development conforms to that of its previous phase	process of determining whether a fully developed software conforms to its requirements specification
2	objective of verification is to check if the work products produced after a phase	validation is applied to the fully developed and integrated software to

	conform to that which was input to the phase.	check if it satisfies the customer's requirements.
3.	primary techniques used for verification include review, simulation, formal verification, and testing. unit and integration testing can be considered as verification steps where it is verified whether the code is a s per the module and module interface specifications.	system testing can be considered as a validation step where it is determined whether the fully developed code is as per its requirements specification.
4	Verification does not require execution of the software	Validation requires execution of the software.
5	Verification is carried out during the development process to check if the development activities are proceeding alright	validation is carried out to check if the right as required by the customer has been developed.
6	verification is concerned with phase containment of errors	the aim of validation is to check whether the deliverable software is error free.

5.3.2 Testing Activities

Testing involves performing the following main activities:

1. Test suite design:
- The set of test cases using which a program is to be tested is designed possibly using several **test case design techniques**.

2. 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.

3. Locate error:
- In this activity, the failure symptoms are analysed to locate the errors.
- For each failure observed during the previous activity, the statements that are in error are identified.

4. Error correction:
- After the error is located during debugging, the code is appropriately changed to correct the error.

The testing activities have been shown schematically in Figure 5.2. As can be seen, the test cases are first designed, the test cases are run to detect failures. The bugs causing the failure are identified through debugging, and the identified error is corrected

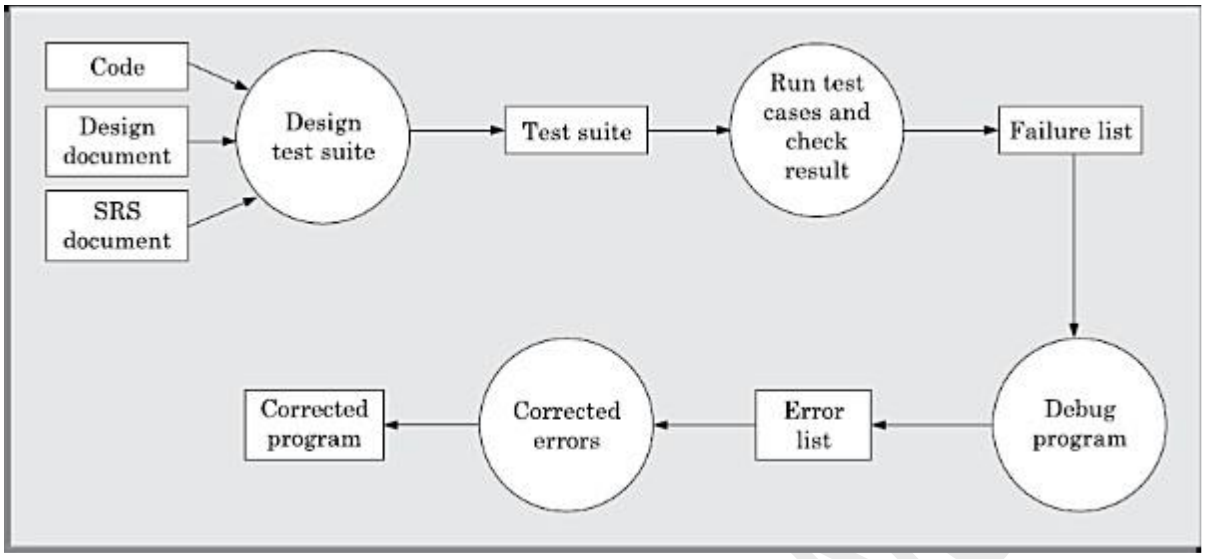

Figure 5.2: Testing process.

5.3.3 Why Design Test Cases?

- When test cases are designed based on **random input data**, many of the test cases do **not contribute to the significance of the test suite**, That is, they do not help detect any additional defects not already being detected by other test cases in the suite.
- A minimal test suite is a carefully designed **set of test cases** such that each test case helps detect different errors.

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

1. Black-box approach
2. White-box (or glass-box) approach

Sr.no	Black-box approach	White-box (or glass-box) approach
1	test cases are designed solely based on an analysis of the input/out behaviour(functional specification)	white-box test cases are based on an analysis of the code.
2	does not require any knowledge of the internal structure of a program.	designing white-box test cases requires a thorough knowledge of the internal structure of a program
3	black-box testing is also known as functional testing	white-box testing is also called structural testing.

5.3.4 Testing in the Large versus Testing in the Small

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

1. Unit testing
2. Integration testing
3. System testing

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

5.4 UNIT TESTING

- Unit testing is undertaken **after a module has been coded and reviewed.**
- This activity is typically undertaken by the **coder** of the module himself in the **coding phase**
- Before carrying out unit testing, the **unit test cases** have to be **designed** and the **test environment** for the unit under test has to **be developed**.

Driver and stub modules

- In order to test a **single module**, we need a complete environment to provide all relevant code that is necessary for execution of the module.
- 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.

- **Stub:** The role of stub and driver modules is pictorially shown in Figure 5.3.
- A stub procedure is a **dummy procedure** that has the **same I/O parameters** as the function called by the unit under test but has a highly simplified
- **Example**, a stub procedure may produce the expected behaviour using a simple table look up mechanism.
- **Driver:** A driver module should contain the **non-local data structures** accessed by the **module under test**.
- it should also have the code to call the different functions of the unit under test with appropriate parameter values for testing.

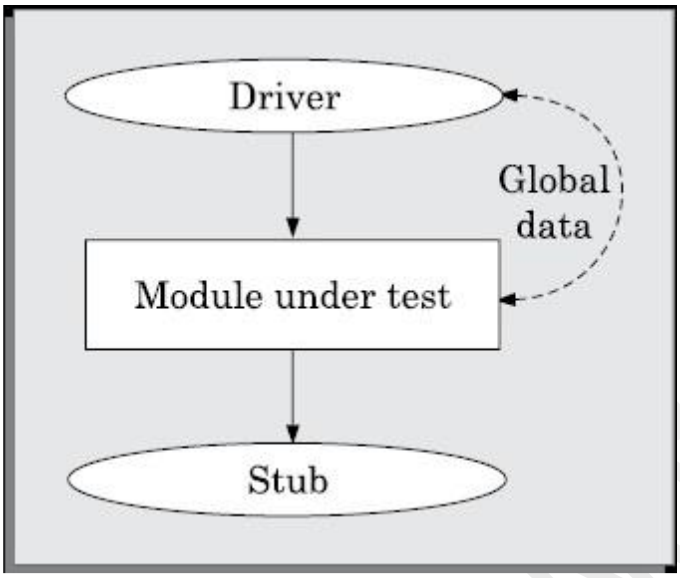

Figure 5.3: Unit testing with the help of driver and stub modules

5.5 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:

a. Equivalence class partitioning
b. Boundary value analysis

5.5.1 Equivalence class partitioning

- The main idea behind defining equivalence classes of input data is that testing the code with any one value belonging to an **equivalence class** is as good as testing the code with any other value belonging to the same equivalence class.

Equivalence classes for a unit under test can be designed by examining the input data and output data. The following are two general guidelines for designing the equivalence classes:

1. If the input data values to a system can be specified by a range of values, then one valid and two invalid equivalence classes need to be defined.

For example, if the equivalence class is the set of integers in the range 1 to 10 (i.e., [1,10]), then the invalid equivalence classes are $[-\infty,0]$, $[11,+\infty]$.

2. If the input data assumes values from a set of discrete members of some domain, then one equivalence class for the valid input values and another equivalence class for the invalid input values should be defined.

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

Example 5.6 for 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 5.8: 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**.

Answer: The equivalence classes are the leaf level classes shown in Figure 5.4. 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}.

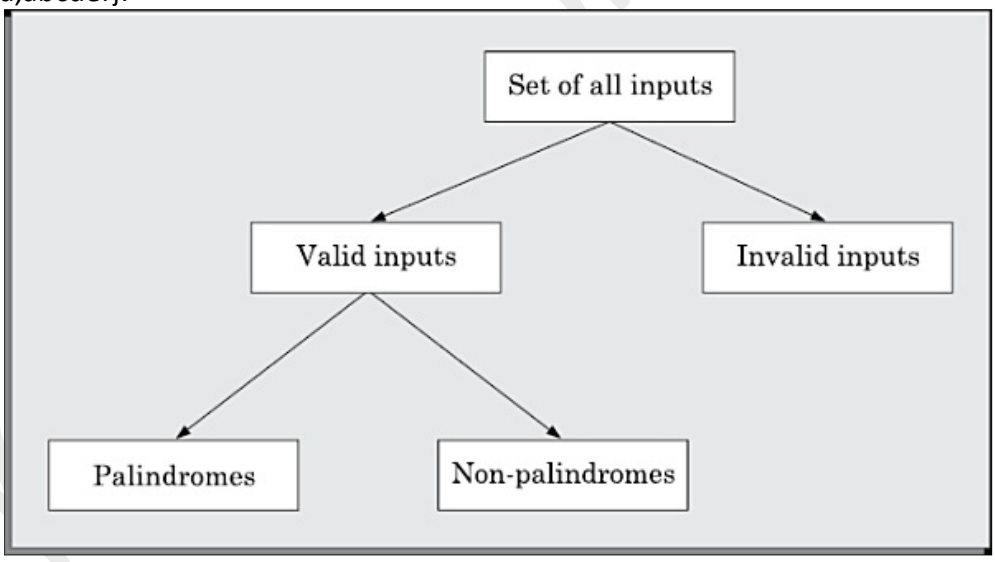

Figure 5.4: Equivalence classes for Example 5.6

5.5.2. 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
- Programmers often fail to properly address the special processing required by the input values that lie at the boundary of the different equivalence classes.
- For example, programmers may improperly use < instead of <=, or conversely <= for <, etc.
- Boundary value analysis-based **test suite design** involves **designing test cases** using the values at the boundaries of **different equivalence classes.**
- To design boundary value test cases, it is required to examine the equivalence classes to check if any of the equivalence classes contains a range of values
- For an equivalence class that is a **range of values**, the boundary values need to be included in the test suite.
- For example, if an equivalence class contains the integers in the range 1 to 10, then the boundary value test suite is {0,1,10,11}.
- **Example 5.9: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}.***

- **Example 5.10:Design boundary value test suite for the function described in Example 5.6.**
- Answer: The equivalence classes have been showed in Figure 5.5. There is a boundary between the valid and invalid equivalence classes. Thus, the boundary value test suite is {abcdefg, abcdef}.

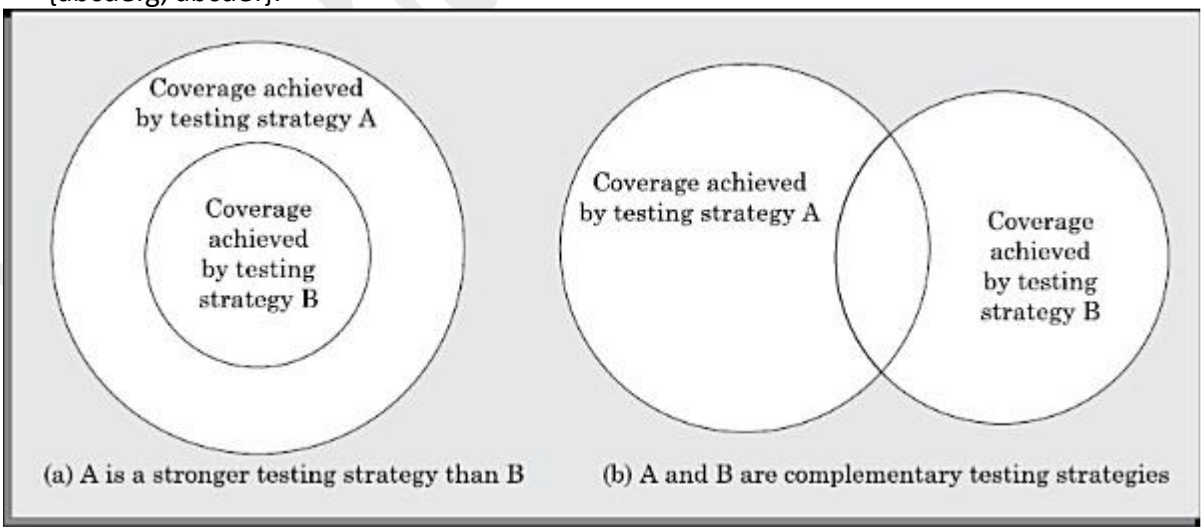

Figure 5.5: CFG for (a) sequence, (b) selection, and (c) iteration type of constructs.

5.5.3 Summary of the Black-box Test Suite Design Approach

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.
    a. Examine if any equivalence class is a range of values.
    b. Include the values at the boundaries of such equivalence classes in the test suite.

5.6 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.

5.6.1 Basic Concepts

- A white-box testing strategy can either be coverage-based or faultbased.

1. 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**

2. 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

- The set of specific program elements that a **testing strategy targets** to execute is called the testing criterion of the strategy.
- 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 elements of the domain defined by that criterion.**

Stronger versus weaker testing

- white-box testing strategy is said to be **stronger than another strategy**, if the stronger testing strategy covers all **program elements covered by the weaker testing strategy**, and the stronger strategy additionally covers at least one program element that is not covered by the weaker strategy.

- When **none of two testing strategies fully covers the program** elements exercised by the other, then the two are called **complementary testing** strategies.
- The concepts of stronger, weaker, and complementary testing are schematically illustrated in Figure 5.6.
- Observe in Figure 5.6(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 5.6(b) shows A and B are complementary testing strategies since some elements of A are not covered by B and vice versa.
- A test suite should, however, be enriched by using various complementary testing strategies.

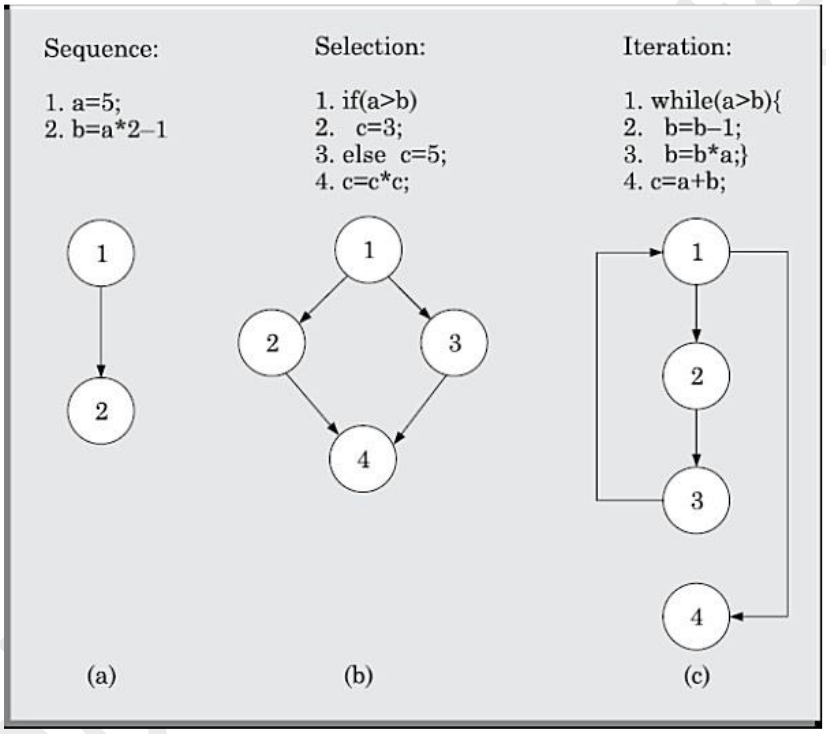

Figure 5.6: Illustration of stronger, weaker, and complementary testing strategies.

5.6.2 Statement Coverage

- The statement coverage strategy aims to design **test cases** so as to execute every statement in a program at least once.
- The principal idea governing the statement coverage strategy is that ***unless a statement is executed, there is no way to determine whether an error exists in that statement***.
- 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.

In the following, we illustrate a test suite that achieves statement coverage.

Example 5.11 Design statement coverage-based test suite for the following Euclid's GCD computation program:

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

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

5.6.3 Branch Coverage

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

- **Example 5.12 For the program of Example 5.11, determine a test suite to achieve branch coverage.**
- Answer: The test suite {(x = 3, y = 3), (x = 3, y = 2), (x = 4, y = 3), (x = 3, y = 4)} achieves branch coverage.
- It is easy to show that branch coverage-based testing is a stronger testing than statement coverage-based testing.

5.6.4 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.**
- Answer: ***Example 5.13 Give an example of a fault that is detected by multiple condition coverage, but not by branch coverage.***
- 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} achievesbranch coverage. But, it is not able to check that setWarningLightOn(); should not be called for temperature values within 150 and 50.

5.6.5 Path Coverage

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

Control flow graph (CFG)

- A control flow graph describes how the control flows through the program.
- We can define a control flow graph as the following: A control flow graph **describes the sequence in which the different instructions of a program get executed.**

Steps to draw the control flow graph of a program

1. First number all the statements of a program. The different numbered statements serve as nodes of the control flow graph (see Figure 5.5).
2. Draw 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.

- A control flow graph describes the sequence in which the different instructions of a program get executed.
- A CFG is a directed graph consisting of a set of nodes and edges (N, E).
- the CFG of the program given in Figure 5.7(a) can be drawn as shown in Figure 5.7(b).

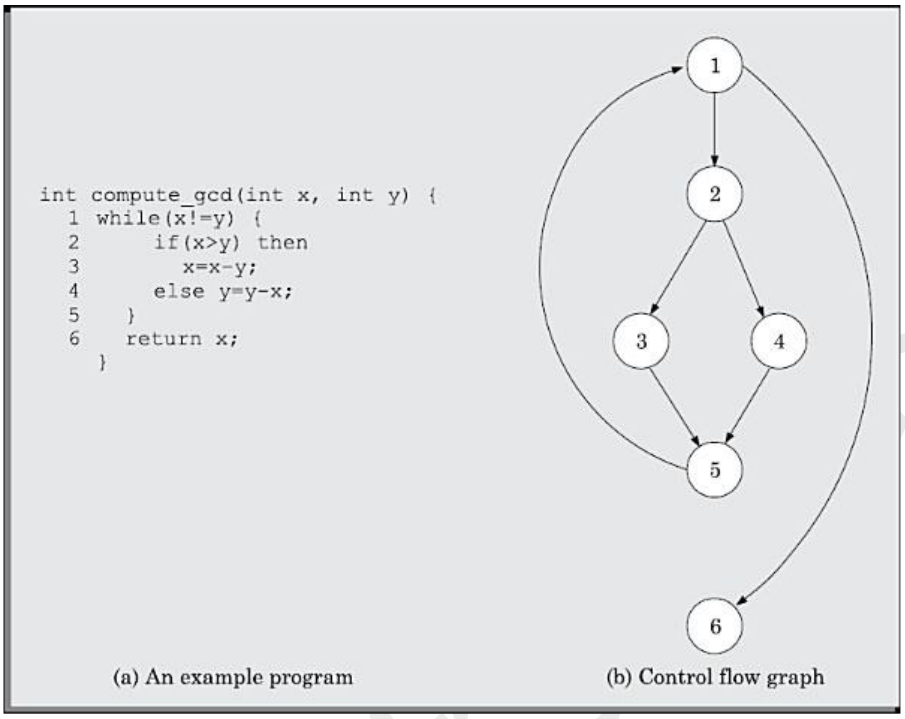

Figure 5.7: Control flow diagram of an example program

Path

- A path through a program is **any node and edge sequence** from the start node to a terminal node of the **control flow graph** of a program.

- Path coverage testing does not try to cover all paths, but only a subset of paths called linearly independent paths (or basis paths).

Linearly independent set of paths (or basis path set)

- A set of paths for a given program is called **linearly independent set of paths** (or the set of basis paths or simply the basis set), if **each path in the set introduces at least one new edge that is not included in any other path in the set.**
- **if 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
- If a set of paths is linearly independent of each other, **then no path in the set** can be obtained through any linear operations (i.e., additions or subtractions) on the other paths in the set.
- **McCabe's cyclomatic complexity** metric used to compute the number of linearly independent paths for any arbitrary program.

5.6.6 McCabe's Cyclomatic Complexity Metric

McCabe's cyclomatic complexitydefines an upper bound on the number of independent paths in a program.

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
E is the number of edges in the control flow graph.

For the CFG of example shown in Figure 10.7, E = 7 and N = 6. Therefore,
the value of the **Cyclomatic complexity = 7 – 6 + 2 = 3.**

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.**
- The number of bounded areas in a CFG increases with the number of decision statements and loops

- CFG example shown in Figure 5.7. 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.**

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

Uses of McCabe's cyclomatic complexity metric

1. It is used in path testing
2. **Estimation of structural complexity of code**: Cyclomatic complexity of a program is a measure of the psychological complexity or the level of difficulty in understanding the program
3. **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.
4. **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.** Larger is the structural complexity, the more difficult it is to test and debug the code.

5.6.7 Data Flow-based Testing

- Data flow based testing method selects **test paths of a program** according to the **definitions and uses of different variables** in a program.
- Consider a program P . For a statement numbered S of P , let

  DEF(S) = {X /statement S contains a definition of X } and
  USES(S)= {X /statement S contains a use of X }

  For the statement **S: a=b+c;, DEF(S)={a}, USES(S)={b, c}.** The definition of **variable X** at **statement S is said to be live at statement S1** , if there exists a path from statement S to statement S1 which does not contain any **definition of X** .

- The **definition-use chain (or DU chain) of a variable X** is of the form **[X, S, S1],** where S and S1 are statement numbers, such that X belongs to DEF(S) and belongs to USES(S1), and the **definition of X in the statement S is live at statement S1 .**
- One simple **data flow testing strategy** is to require that **every DU chain** be covered **at least once.**
- Data flow testing strategies are especially useful for testing programs containing nested if and loop statements

5.6.8 Mutation Testing

- Mutation testing is a **fault-based testing technique** in the sense that mutation test cases are designed to help **detect specific types of faults** in a program.
- In mutation testing, a **program is first tested** by using an **initial test suite** designed by using various **white box testing strategies.**
- After the initial testing is complete, mutation testing can be taken up.
- The idea behind mutation testing is to make a **few arbitrary changes** to a program at a time.Each time the **program is changed**, it is called a **mutated program** and the change effected is called a **mutant**.
- **For example**, one mutation operator may randomly **delete a program statement**.

- A **mutant** may or may not **cause an error in the program**. If a mutant does not introduce any error in the program, then the **original program and the mutated program** are called **equivalent programs.**
- A mutated program is tested against the original test suite of the program. If there exists at least one test case in the test suite for which a mutated program yields an incorrect result, then the mutant is said to be dead, since the error introduced by the mutation operator has successfully been detected by the test suite.
- **Advantage of mutation testing** is that it can be automated to a great extent. The process of generation of mutants can be automated by predefining a **set of primitive changes** that can be applied to the program.
- These **primitive changes** can be **simple program alterations** such as—
  - deleting a statement
  - deleting a variable definition
  - changing the type of an arithmetic operator (e.g., + to -)
  - changing a logical operator (and to or)
  - changing the value of a constant
  - Changing the data type of a variable, etc.
- Mutation-based testing approach is computationally **very expensive**, since a large number of possible mutants can be generated.
- Mutation testing is **not suitable for manual testing.** some testing tool that should automatically generate the mutants and run the test suite automatically on each mutant.

5.7 PROGRAM ANALYSIS TOOLS

- A program analysis tool usually is an **automated tool** that takes either the **source code or the executable code** of a program as **input** and **produces reports** regarding several important **characteristics of the program**, such as
  - its size, complexity, adequacy of commenting,adherence to programming standards, adequacy of testing, etc.

  program analysis tools are categorised into the following
  1. Static analysis tools
  2. Dynamic analysis tools

5.7.1 Static Analysis Tools

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

- 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.
- **Example:** Code review techniques such as code walkthrough and code inspection, compiler
- **Limitation :** inability to analyse **run-time information** such as dynamic memory references using pointer variables and pointer arithmetic, etc

5.7.2 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.
- A dynamic program analysis tool (also called a dynamic analyser) usually **collects execution trace information by instrumenting the code.** Code instrumentation is usually achieved by inserting additional statements to print the values of certain variables into a file to collect the execution trace of the program. The instrumented code when executed, records the behaviour of the software for different test cases.
- After software has been tested with its **full test suite** and its behaviour recorded, the dynamic analysis tool carries out a **post execution analysis** and **produces reports** which describe the coverage that has been achieved by the complete test suite for the program.
- **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.
- Dynamic analysis results are reported in the form of a **histogram or pie chart** to describe the structural coverage achieved for different modules of the program.

5.8 INTEGRATION TESTING

- Integration testing is carried out after all (or at least some of) the modules have been **unit tested**.

- 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.
- The objective of integration testing is to **check whether the different modules** of a **program interface** with each other properly.

- Any one (or a mixture) of the following approaches can be used to develop the test plan:
    1. Big-bang approach to integration testing
    2. Top-down approach to integration testing
    3. Bottom-up approach to integration testing
    4. Mixed (also called sandwiched) approach to integration testing

1. Big-bang approach to integration testing

- All the unit tested modules making up a system are integrated in a single step & tested.
- Used only for very small systems.
- **Problem** : once a failure has been detected during integration testing, it is very difficult to localise the error as the error may potentially lie in any of the modules.
- Debugging errors reported during big-bang integration testing **are very expensive to fix**.
- Never used for **large programs**.

2. Bottom-up approach to integration testing

- Large software products are often made up of **several subsystems**.
- A subsystem might consist of **many modules** which communicate among each other through well-defined interfaces.
- In this approach first the **modules for the each subsystem are integrated**. Thus, the subsystems can be **integrated separately and independently.**
- Purpose is to test whether the interfaces among various modules making up the Subsystem work satisfactorily.
- Large software systems normally require several levels of **subsystem testing; lower-level subsystems are successively combined to form higher-level subsystems.**
- **Advantage**:
    - Several disjoint subsystems can be tested simultaneously.
    - Low-level modules get tested thoroughly, since they are exercised in each integration step so increases the reliability of the system.
- **Disadvantage:**

- The **complexity that occurs** when the system is made up of a large number of small subsystems that are at the same level.

3. Top-down approach to integration testing

- Top-down integration testing starts **with the root module in the structure chart** and **one or two subordinate modules** of the root module.
- After the top-level 'skeleton' has been tested, the modules that are at the immediately lower layer of the 'skeleton' are combined with it and tested.
- Requires the use of program stub but does not require any driver routines.
- **Advantage:**
  - It requires writing only **stubs, and stubs** are simpler to write compared to drivers.
- **Disadvantage**:
  - 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.**

4. Mixed approach to integration testing

- **shortcoming of the top-down and bottom-up approaches:**
  - In topdown 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.
- most commonly used integration testing approaches.

5.8.1 Phased versus Incremental Integration Testing

- In incremental integration testing, only one new module is added to the partially integrated system each time.
- In phased integration, a group of related modules are added to the partial system each time.

5.9 SYSTEM TESTING

- After all the units of a program have been integrated together and tested, **system testing is taken up.**
- System tests are designed **to validate a fully developed system** to assure that **it meets**

its requirements.

- The test cases are 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.**

- The system test cases can be classified into
  a. **Functionality**:
     i. The functionality tests are designed to check whether the software satisfies the **functional requirements** as documented in the SRS document.
     ii. **Smoke testing** is done to check whether at least the main functionalities of the software are working properly.
  b. **Performance test cases:**
     i. Test the conformance of the system with the non functional requirements of the system.

5.9.1 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**.**
- **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.

5.9.2 Performance Testing

- Performance testing is carried out **to check whether the system meets** the **nonfunctional requirements** identified in the SRS document
- All performance tests can be considered as black-box tests.

1. 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 i**mpose 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.
- 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.**

2. 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.

3. Configuration testing

- Configuration testing is used to **test system behaviour** in various **hardware and software configurations** specified in the requirements.
- Sometimes systems are built to work in different configurations for different users.
- For instance, **a minimal system** might be required to serve a single user, and other extended configurations may be required to serve additional users during configuration testing.
- The system is configured in each of the required configurations and depending on the specific customer requirements, it is checked if the system behaves correctly in all required configurations.

4. 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.**

5. Regression testing

- This type of testing is required when software is maintained **to fix some bugs or enhance functionality, performance, etc.**

6. 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**.

7. Maintenance testing

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

8. 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.

9. 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.
- Functionally correct GUI is not enough. Therefore, the GUI has to be checked against the checklist.

10. Security testing

- Security testing is essential for software that handles or **process confidential data** that is to be gurarded against pilfering.
- It needs to be tested whether the system is **fool-proof from security attacks** such as intrusion by hackers.
- Security testing techniques include **password cracking, penetration testing, and attacks on specific ports, etc.**