UNIT - 4

Coding and Testing: Coding, Code review, Software documentation, Testing, Black-box testing, White-Box testing, Debugging, Program analysis tools, Integration testing, Testing object-oriented programs, Smoke testing, and Some general issues associated with testing.
Software Reliability and Quality Management: Software reliability. Statistical testing, Software quality, Software quality management system, ISO 9000.SEI Capability maturity model. Few other important quality standards, and Six Sigma.

CODING AND TESTING

1. CODING:

The input to the coding phase is the design document produced at the end of the design phase. The objective of the coding phase is to transform the design of a system into code in a high-level language, and then to unit test this code. Good software development organisations require their programmers to adhere to some well-defined and standard style of coding which is called their coding standards. The main advantages of coding are:

- A coding standard gives a uniform appearance to the codes
- It promotes good programming practices.
- It facilitates code understanding and code reuse
- It is important to detect as many errors as possible during code reviews.

1.1 CODING STANDARDS AND GUIDELINES:

Good software development organisations usually develop their own coding standards and guidelines based on the specific types of software they develop.

Representative coding standards:-

- Contents of the headers,preceding codes for different modules Format specified
  - Name of the module.
  - Date on which the module was created
  - Author's name.
  - Modification history.
  - Synopsis of the module.
  - Different functions supported , input/output parameters.
  - Global variables accessed by the module.
- Naming conventions for global variables, local variables, and constant identifiers.
- Error return conversations and Exception handling mechanism.

Representative coding guidelines:-

- Do not use a coding style that is too clever or too difficult to understand.
- Avoid obscure side effects.
- Do not use an identifier for multiple purposes.
- Code should be well-documented.
- Length of any function should not exceed 10 source lines.
- Do not use GOTO statements.

2. CODE REVIEW:

- ❖ Code review undertaken after the module successfully compiles. If a failure is detected, it will be located and removed. Code reviews are of two types:
  - Code Walkthrough
  - Code Inspection.

2.1 CODE WALKTHROUGH :

- Code walkthrough is an informal code analysis technique**.**
- All syntax errors have been eliminated**.**
- The main objective of code walkthrough is to discover the algorithmic and logical errors in the code.
- The members note down their findings of their walkthrough and discuss those in a walkthrough meeting.

❖ **Guidelines are:**

- The team should not be either too big or too small, it consists of three to seven members.
- Discussions should be on errors.
- managers should not attend the walkthrough meetings.

These guidelines are based on personal experience, common sense and several other subjective factors.

2.2 CODE INSPECTION:

- The code execution is done in code walk through.
- Coding standards is checked-in during code inspection

The list of programming errors checked during Code Inspection :

- Use of uninitialised variables.
- Jumps into loops.
- Non-terminating loops.
- Array indices out of bounds.
- Improper storage allocation and deallocation.
- Mismatch between actual and formal parameter.
- Use of incorrect logical operators.
- Improper modification of loop variables.
- Comparison of equality of floating point values.

2.3 CLEAN ROOM TESTING:

1. The programmers are not allowed to test any of their code by executing other than doing some syntax testing using a compiler.
2. This produces documentation and code
3. The main problem with this approach is that testing effort is released because walkthroughs, inspection, and verification are time consuming.

3. SOFTWARE DOCUMENTATION :

- Developers develops the executable files and source code along with various kinds of documents like:
  - Users' manual
  - Software requirements specification (SRS) document

- Design document
- Test document
- Installation manual ,etc..

- Good documents are very usefull and serve :
  - Enhanced understandability and maintainability of software product.
  - Helps reduce the effort and time
  - Users can explore the system effectively
  - It tracks the progress of the project.

❖ **Different types of software documents:**

- Internal documentation
- External documentation

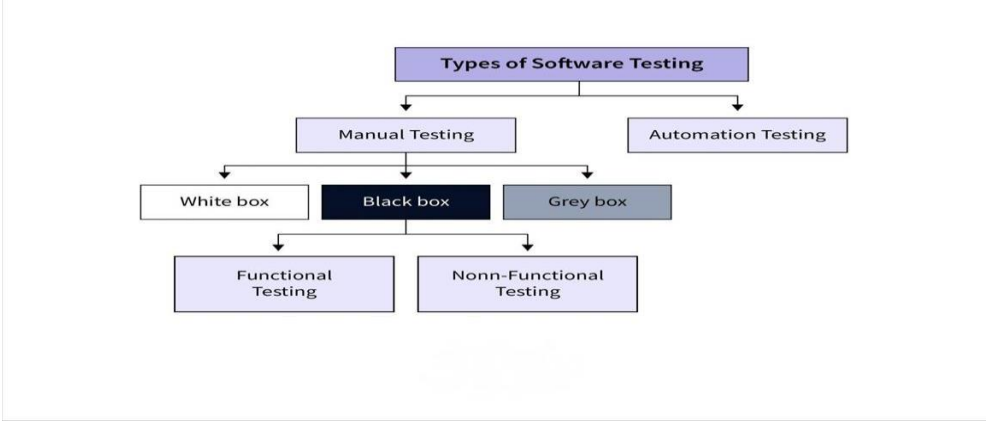
3.1 Internal Documentation: These are provided in the source code itself. The important types of internal documentation are the following:

- Comments embedded in the source code.
- Use of meaningful variable names.
- Module and function headers.
- Code indentation.
- Code structuring (i.e., code decomposed into modules and functions).
- Use of enumerated types.
- Use of constant identifiers.
- Use of user-defined data types.

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

4. TESTING :

- The aim of the testing process is to identify all the defects in software products**.**
- It is not possible to guarantee that the software is error free**.**
- Testing provide a practical way of reducing defects in a system and increasing the user's confidence.



4.1 How to test a program & Testing Activities :

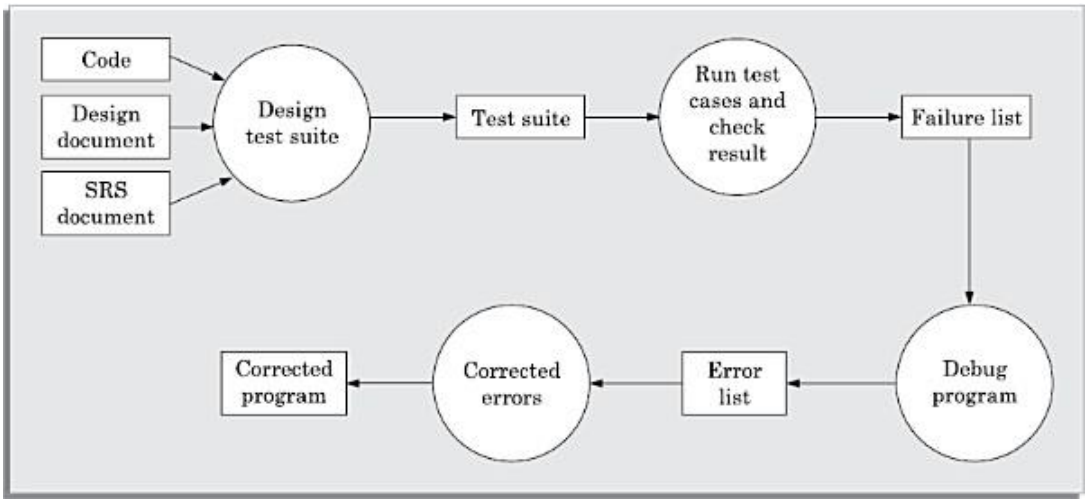


Figure 10.2. Testing Process

The testing activities have been shown schematically in Figure 10.2.

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

4.2 Verification versus Validation :

- ***Verification****:* Verification is the process of checking whether a product, service, or system meets specified requirements and ensures that it was built correctly according to design specifications. Are we building the product right.
- ***Validation****:* Validation is the process of evaluating a product, service, or system to ensure it meets the needs and expectations of the end-users and fulfills its intended purpose. Are we building the right product.
- Error detection techniques = Verification techniques + Validation techniques

4. 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.
- **E.g :** if (x>y) max = x;
  else max = x;
- The Test suite {(x=3,y=2);(x=2,y=3)} can detect the error, whereas a larger test suite {(x=3,y=2);(x=4,y=3); (x=5,y=1)} does not detect the error.

5. UNIT TESTING :

Unit testing is undertaken after a module has been coded and reviewed. Driver and stub modules :

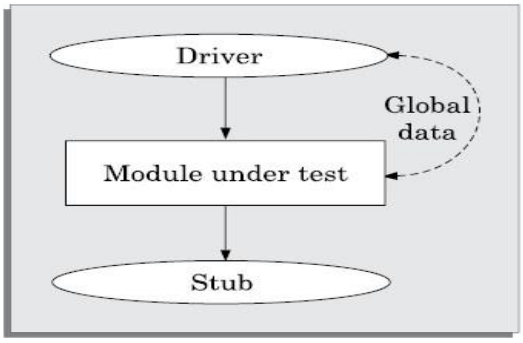


Figure 10.3: Unit testing with the help of driver and stub modules.

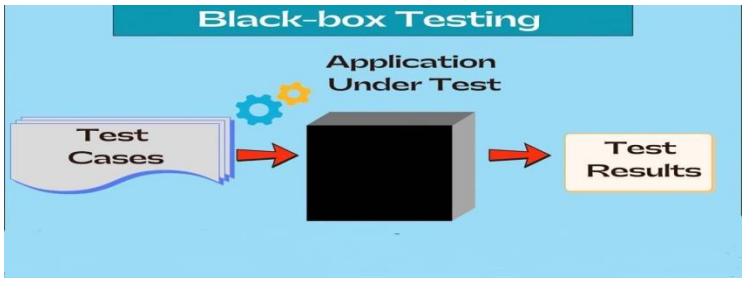
Stubs and **Drivers** is a small program designed to provide the complete environment for a module so that testing can be carried out.

Stub: A stub procedure is a dummy procedure that has the same I/O parameters as the function called by the unit under test but has a highly simplified behaviour. E.g: A stub procedure may produce the expected behaviour using a simple table look up mechanism. The mechanism used is top bottom intigration.

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

6. 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.



Two main approaches:

- Equivalence class partitioning .
- Boundary value analysis .

6.1 Equivalence Class Partitioning:

In this approach, the domain of input values to the program under test is partitioned into a set of equivalence classes. Some general guidelines are :

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.
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.

- **E.g** : 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 palindromes, non-palindromes, and invalid inputs. Now, selecting one representative value from each equivalence class, we have the required test suite: {abc,aba,abcdef}.

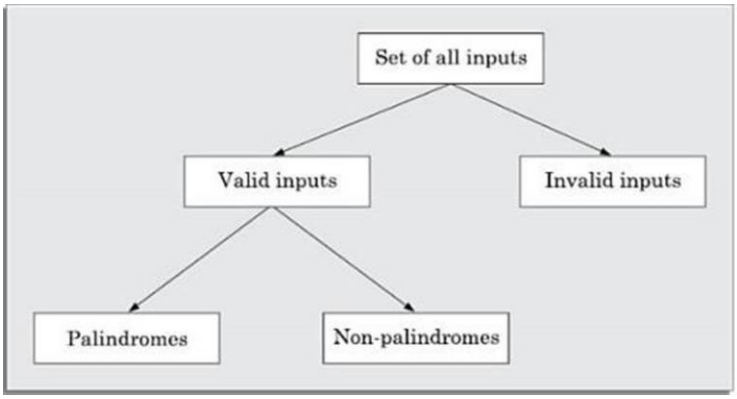


Figure: Equivalence classes for above Example .

6.2 Boundary Value Analysis (BVA):

- ❖ BVA-based test suite design involves designing test cases using the values at the boundaries of different equivalence classes**.** 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 may improperly use **<** instead of **<=**, or conversely **<=** instead of **<.**
- ❖ **E.g :** 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 suit
- ❖ **Answer**: BVA between the valid and invalid equivalence classes. Thus, the boundary value test suite is { 0, -1, 5000, 5001 }.

7. WHITE-BOX TESTING :

1. White-box testing is an important type of unit testing. A large number of white- box testing strategies exist.
2. A white-box testing strategy is said to be stronger than another strategy, if the stronger testing strategy covers all program elements covered by the weaker testing strategy, and the stronger strategy additionally covers at least one program element that is not covered by the weaker strategy

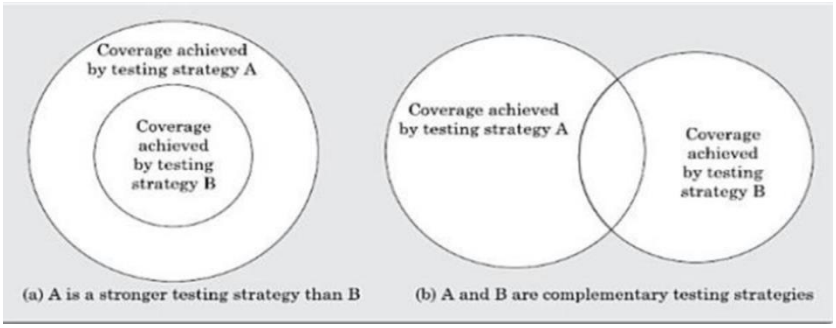
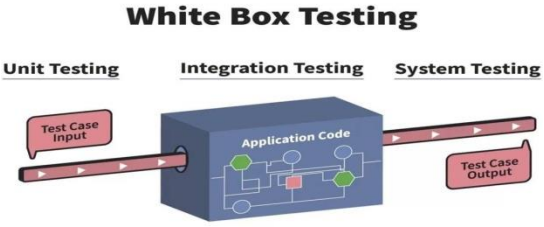


Figure 10.5: Stronger and Complementary testing strategies..



7.1 Basic Concepts:

1. **Fault-based testing**: A fault-based testing strategy targets to detect certain types of faults. These faults that a test strategy focuses on constitutes the fault model of the strategy.
2. **Coverage-based testing**: A coverage-based testing strategy attempts to execute (or cover) certain elements of a program.e.g :statement coverage, branch coverage, multiple condition coverage, and path coverage-based testing.
3. **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.
4. **Stronger versus weaker testing:** If a stronger testing has been performed, then a weaker testing need not be carried out.

7.2 Statement Coverage :

The statement coverage strategy aims to design test cases so as to execute every statement in a program at least once. Design statement coverage-based test suite for the following Euclid's GCD computation program:

E.g 0.6 : 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:** 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.

7.3 Branch Coverage (or) Edge Testing :

For branch coverage each branch in the CFG representation of the program must be taken at least once, when the test suite is executed. Each edge of a program's control flow graph is traversed at least once .

For the program of **E.g 0.6** 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.

7.4 Multiple Condition Coverage :

- Branch testing can be considered to be a simplistic condition testing strategy where only the compound conditions appearing in the different branch statements are made to assume the true and false values. It is easy to prove that condition testing is a stronger testing strategy than branch testing.

```
E.g:    if(temp>150 || temp>50)
        setWarningLightOn();
```

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

7.5 Path Coverage

- 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.
- A control flow graph describes the sequence in which the different instructions of a program get executed.

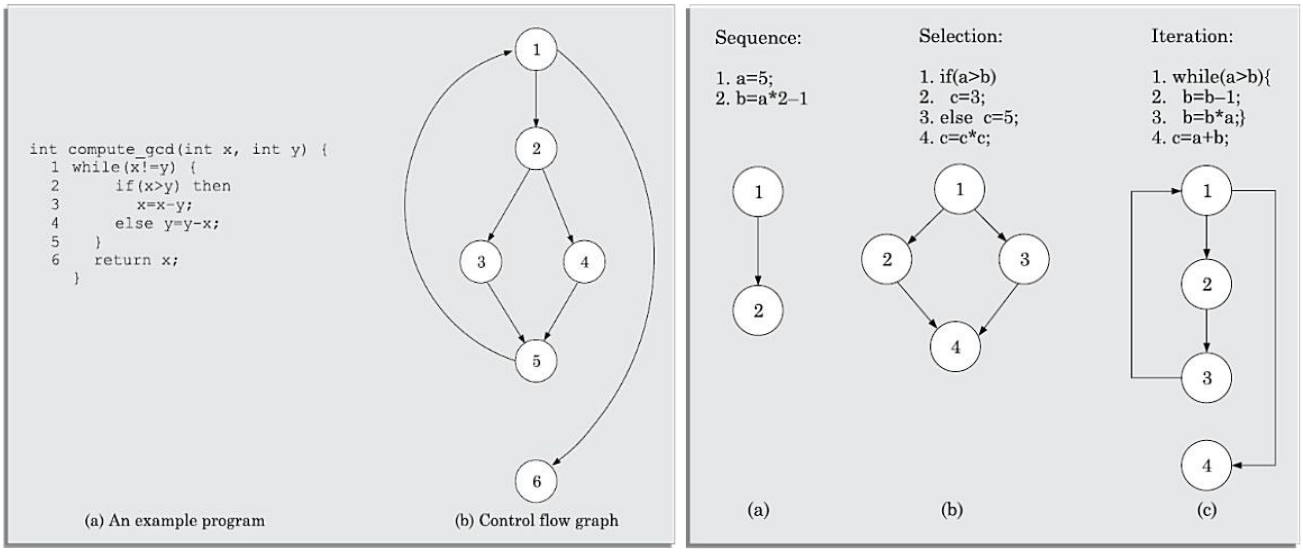


Figure 10.7: Control flow diagram of an example program.

Path:

- A path through a program is any node and edge sequence from the start node to a terminal node of the control flow graph of a program.
- Writing test cases to cover all paths of a typical program is impractical since there can be an infinite number of paths through a program in presence of loops. E.g, in Figure 10.5(c),

Linearly independent set of paths (or basis path set):

- A set of paths for a given program is called linearly independent set of paths, if each path in the set introduces at least one new edge that is not included in any other path in the set. This is because, any path having a new node would automatically have a new edge.

7.6 McCabe's Cyclomatic Complexity Metric :

- McCabe's cyclomatic complexity defines 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 no. of nodes and E is the no. of edges. For the CFG of example shown in Figure 10.7, E = 7 and N = 6. So, 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:
    - V(G) = Total no. of non-overlapping bounded areas + 1
- **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 are 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 no of test cases required to achieve path coverage.
4. repeat

7.7 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 S^1 , if there exists a path from statement S to statement S^1 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, S^1], where S and S^1 are statement numbers, such that X € DEF(S) and X € USES(S^1), and the definition of X in the statement S is live at statement S^1 . Every DU chain be covered at least once. Data flow testing strategies are especially useful for testing programs containing nested if and loop statements.

7.8 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.
- 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.
- The mutants can be automated by predefining a set of 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.

8. DEBUGGING :

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

8.1 Debugging Approaches:

1. **Brute force method:** This is least efficient method. 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.
2. **Backtracking:** In this approach, starting from the statement at which an error symptom has been observed, the source code is traced backwards until the error is discovered.
3. **Cause elimination method:** Based on the failure symptoms, the causes which could possibly have contributed to the symptom is developed and tests are conducted to eliminate each.
4. **Program slicing:** This technique is similar to back tracking. The search space is reduced by defining slices. Program slicing makes use of the fact that an error in the value of a variable can be caused by the statements on which it is data dependent.

8.2 Debugging Guidelines :

Debugging is often carried out by programmers based on their ingenuity and experience**.**

- Many times debugging requires a thorough understanding of the program design.
- Debugging may sometimes even require full redesign of the system
- One must be beware of the possibility that an error correction may introduce new errors

9. PROGRAM ANALYSIS TOOLS :

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

There are two types of tools:

1. Static analysis tools
2. Dynamic analysis tools

9.1 Static analysis tools:

- Static program analysis tools compute various characteristics of a program without executing it.
- These tools analyse the source code to compute certain metrics characterising the source code (such as size, cyclomatic complexity, etc.)
- It also report certain analytical conclusions.
- To what extent the coding standards have been adhered to?
- Whether certain programming errors such as un initialised variables.
- Mismatch between actual and formal parameters and variables that are declared but never used.

Static analysis tools often summarise the results of analysis of every function in a polar chart known as Kiviat Chart. A Kiviat Chart typically shows the analysed values for cyclomatic complexity, number of source lines, percentage of comment lines, Halstead's metrics, etc.

9.2 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
- These tools usually record and analyse the actual behaviour of a program while it is being executed.
- A dynamic program analysis tool is also called as dynamic analyser.
- An important characteristic of a test suite that is computed by a dynamic analysis tool is the extent of coverage achieved by the test suite.

Normally the 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

10. INTEGRATION TESTING :

1. Integration testing is carried out after all (or at least some of) the modules have been unit tested.
2. Successful completion of unit testing, to a large extent, ensures that the unit (or module) as a whole works satisfactorily. E.g : it is checked that no parameter mismatch occurs when one module invokes the functionality of another module.
3. The objective of integration testing is to check whether the different modules of a program interface with each other properly.
4. An important factor that guides the integration plan is the module dependency graph.

Few approaches are:

- **Big-bang approach to integration testing**
  1. Big-bang testing is the most obvious approach to integration testing.
  2. All the modules making up a system are integrated in a single step.
  3. The problem is that once a failure has been detected during integration testing it is very difficult to localise the error as the error may potentially lie in any of the modules.
  4. As a result, big-bang integration testing is almost never used for large programs.

- **Bottom-up approach to integration testing:**
  1. Large software products are often made up of several subsystems.
  2. A subsystem might consist of many modules which communicate among each other through well-defined interfaces.
  3. The test cases must be carefully chosen to exercise the interfaces in all possible manners.
  4. The principal advantage of bottom- up integration testing is that several disjoint subsystems can be tested simultaneously.
  5. A disadvantage of bottom-up testing is the complexity that occurs when the system is made up of a large number of small subsystems that are at the same level

- **Top-down approach to integration testing:**
  1. Top-down integration testing starts with the root module in the structure chart and one or two subordinate modules of the root module.
  2. Simulate the effect of lower-level routines that are called by the routines under test.
  3. Advantage is that it requires writing only stubs, and stubs are simpler to write compared to drivers.
  4. A disadvantage is that in the absence of lower-level routines

- **Mixed approach to integration testing :**
  1. This follows a combination of top-down and bottom-up testing approaches.
  2. Similarly, bottom-up testing can start only after the bottom level modules are ready.
  3. In this approach, testing can start as and when modules become available after unit testing.
  4. In this approach, both stubs and drivers are required to be designed.

11. TESTING OBJECT-ORIENTED PROGRAMS :

- Testing is the process of evaluating a software application to identify defects, ensure it meets requirements, and verify its functionality, performance, and security. It includes manual and automated testing methods.
- **Testing object-oriented:** Testing object -oriented programs is much more difficult and requires much more cost and effort as compared to testing similar procedural programs

11.1 What is a Suitable Unit for Testing Object-Oriented Programs?

Object-oriented Programs

1. For procedural programs, we had seen that procedures are the basic units of Testing.
2. As far as procedural programs are concerned, procedures are the basic units of testing.
3. Since methods in an object-oriented program are analogous to procedures in a procedural program.
4. 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.

During integration testing various unit tested objects are integrated and tested. Finally, system-level testing is carried out

11.2 Do Various Object-orientation Features Make Testing Easy ?

- **Encapsulation:** The encapsulation feature helps in data abstraction, error isolation, and error prevention
- **Inheritance**: The inheritance feature helps in code reuse and was expected to simplify testing
- **Dynamic binding**: Dynamic binding was introduced to make the code compact, elegant, and easily extensible.
- **Object states**: In contrast to the procedures in a procedural program ,objects store data permanently For state-based testing, it is there fore beneficial to have the state model of the objects, so that the conformance of the object to its state model can be tested.

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

1. In traditional procedural programs, procedures are the basic unit of testing. In contrast, objects are the basic unit of testing for object-oriented programs.
2. The various object-orientation features are explicit in the design models, and it is usually difficult to extract from and analysis of the source code.

11.4 Grey-Box Testing of Object-oriented Programs:

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.

State-model-based testing

State coverage: Each method of an object are tested at each state of the object. State transition coverage: It is tested whether all transitions depicted in the state model work satisfactorily State transition path coverage: All transition paths in the state model are Tested Use case-based testing Scenario coverage: Each use case typically consists of a mainline scenario and several alternate scenarios

Class diagram-based testing:

- Testing derived classes
- Association testing
- Aggregation testing
- Sequence diagram-based testing
- Method coverage
- Message path coverage

11.5 Integration Testing of Object-oriented Programs:

These are of two types:

- 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.

12. SMOKE TESTING :

Smoke testing is carried out before initiating system testing to ensure that system testing would be meaningful, or whether many parts of the software would fail .For smoke testing, a few test cases are designed to check whether the basic functionalities are working. E.g : 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.

12.1 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:** refers to the system testing carried out by the test team within the developing organisation.
  2. **Beta Testing:** is the system testing performed by a select group of friendly customers.
  3. **Acceptance Testing:** is the system testing performed by the customer to determine whether to accept the delivery of the system.
- **Performance testing** is carried out to check whether the system meets the nonfunctional requirements identified in the SRS document.
- **Stress testing** is also known as endurance testing. Stress testing evaluates system performance when it is stressed for short periods of time.

- **Volume testing** checks whether the data structures (buffers, arrays, queues, stacks, etc.) have been designed to successfully handle extraordinary situations.
- **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.
- **Compatibility** aims to check the interfaces with the external systems are performing as required.
- **Regression testing** is required when a software is maintained to fix some bugs or enhance functionality, performance.
- **Recovery testing** tests the response of the system to the presence of faults, or loss of power, devices, services, data, etc.
- **Maintenance testing** is the procedures that are required to help maintenance of the system.
- **Documentation testing** is checked whether the required user manual, maintenance manuals, and technical manuals exist and are consistent.
- **Usability testing** concerns checking the user interface. During usability testing, the display screens, messages, report formats, and other aspects relating to the user interface requirements are tested.

13. SOME GENERAL ISSUES ASSOCIATED WITH TESTING :

Two general issues associated with testing.

Test documentation

- A piece of documentation that is produced towards the end of testing is the test summary report**.**
- **It normally specifies the following:**
  - What is the total number of tests that were applied to a subsystem.
  - Out of the total number of tests how many tests were successful.
  - How many were unsuccessful, and the degree to which they were unsuccessful.

Regression testing:

1. Regression testing spans unit, integration, and system testing.
2. Instead, it is a separate dimension to these three forms of testing.
3. However, if only a few statements are changed, then the entire test suite need not be run.
4. only those test cases that test the functions and are likely to be affected by the change need to be run.
5. Whenever a software is changed to either fix a bug, or enhance or remove a feature, regression testing is carried out.

SOFTWARE RELIABILITY & QUALITY MANAGEMENT

1. SOFTWARE RELIABILITY :

The Reliability of a software product essentially denotes its trustworthiness or dependability. The Reliability of a software product can also be defined as a probability of the product working "correctly " over a given period of time.

• Defect-Dependence: A software product with many defects is unreliable, and reliability improves as defects decrease.

• Mathematical Characterization: There is no simple mathematical expression to relate reliability to the number of bugs in a system.

- Impact of Bug Removal :
  - Removing bugs from rarely executed parts of the software has little effect on reliability.
  - Most execution time (90%) is spent on just 10% of the program's instructions (core instructions).
  - Removing errors from frequently executed parts (core) significantly improves reliability.
  - Removing errors from rarely used parts has minimal impact.

- Factors affecting software reliabilty :
  - Location of Bugs: The impact of fixing a bug depends on where it is located in the code.
  - User Execution Profile**:** Reliability also depends on how different users interact with the software.
  - E.g: In library software, members use different features than librarians, leading to different reliability perceptions.

1.1 Hardware versus Software Reliability :

Hardware components fail due to very different reasons as compared to software components. Hardware components fail mostly due to wear and tear, whereas software components fail due to bugs.

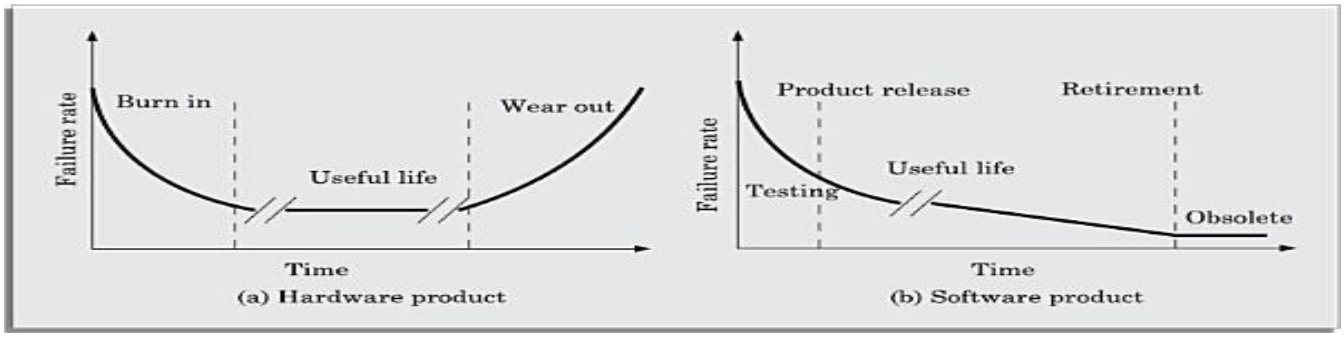


Figure 11.1: Change in failure rate of a product.

1.2 Reliability Metrics of Software Products :

The reliability requirements for different categories of software products may be different six metrics that correlate with reliability as follows:

1. **Rate of occurrence of failure (ROCOF):** ROCOF measures the frequency of occurrence of failures.
2. **Mean time to failure (MTTF):** MTTF is the time between two successive failures, averaged over a large number of failures. To measure MTTF is the failure data for n failures.

3. **Mean time to repair (MTTR):** MTTR measures the average time it takes to track the errors causing the failure and to fix them.
4. **Mean time between failure (MTBF):** The MTTF and MTTR metrics can be combined to get the MTBF metric: MTBF=MTTF+MTTR.
5. **Probability of failure on demand (POFOD):** POFOD measures the likelihood of the system failing when a service request is made.
6. **Availability:** Availability of a system is a measure of how likely would the system be available for use over a given period of time.

- **Shortcomings of reliability metrics of software products :**

These metrics are centered around the probability of occurrence of system failures but take no account of the consequences of failures. A scheme of classification of failures is as follows:

- **Transient:** Transient failures occur only for certain input values while invoking a function of the system.
- **Permanent:** Permanent failures occur for all input values while invoking a function of the system.
- **Recoverable:** When a recoverable failure occurs, the system can recover without having to shutdown and restart the system (with or without operator intervention).
- **Unrecoverable:** In unrecoverable failures, the system may need to be restarted.
- **Cosmetic:** These classes of failures cause only minor irritations, and do not lead to incorrect results.

1.3 Reliability Growth Modelling :

A reliability growth model can be used to predict when (or if at all) a particular level of reliability is likely to be attained. Thus, reliability growth modelling can be used to determine when to stop testing to attain a given reliability level. Lets discuss two very simple reliability growth models.

- **Jelinski and Moranda model :**

The simplest reliability growth model is a **step function model** where it is assumed that the reliability increases by a constant increment each time an error is detected and repaired.

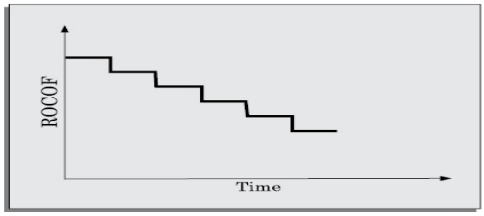


Figure 11.2: Step function model of reliability growth.

- **Littlewood and Verall's model :**

This model allows for negative reliability growth to reflect the fact that when a repair is carried out, it may introduce additional errors. It treats an error's contribution to reliability improvement to be an independent random variable having Gamma distribution.

2. STATISTICAL TESTING :

Statistical testing is used to determine the reliability of a product rather than finding errors.

Operation profile: The operation profile of software is the probability of a user selecting different functionalities within the software**.**

How to define the operation profile for a product?

We need to divide the input data into a number of input classes. Then need to assign a probability value to each input class; to signify the probability for an input value from that class to be selected. The operation profile of a software product can be determined by observing and analysing the usage pattern of the software by a number of users.

2.1 Steps in Statistical Testing :

1.Determine the operation profile of the software.
2.Generate a set of test data corresponding to the operation profile.
3.Apply the test cases to the software and record the time between failures.
4.Observe a statistically significant number of failures to compute reliability.

- **Assumptions for Accurate Results:**
  - A statistically significant number of test cases must be used.
  - A small percentage of inputs that may cause system failure should be included.
  - Common test cases can be easily generated using a test case generator.
  - Unlikely test cases must also be included, but generating them is challenging.

- **Pros of Statistical Testing :**
  - Focuses on the most frequently used parts of the system.
  - Users perceive the system as more reliable.
  - Provides a more accurate reliability estimation compared to other methods.
- **Cons of Statistical Testing**:
  - Difficult to define operation profiles in a repeatable way.
  - Requires a statistically significant number of test cases, which can be challenging.

3. SOFTWARE QUALITY :

Quality is often defined as the "fitness of purpose"—a good product meets the user's needs and requirements.

- **Modern Quality View:**

Software must adapt to changing conditions and be evaluated using quality factors, such as:

- **Portability**: Should work on different hardware and OS environments.
- **Usability**: Should be easy to use for both expert and novice users.
- **Reusability**: Components should be reusable for new products.
- **Correctness**: Must correctly implement the requirements in the SRS document.
- **Maintainability**: Errors should be easy to fix, and functionalities should be modifiable.

- **Limitations for Software:**
  - A functionally correct software product may not be considered high quality if it has an unusable interface.

- If a product does everything users want but is incomprehensible or unmaintainable, it is not truly high quality.

McCall's Quality Factors :

McCall classifies software quality into two levels:

- **Higher-level attributes** (quality factors) – can only be indirectly measured.
- **Second-level attributes** – measurable directly, either objectively or subjectively.

E.g :

- Reliability is a higher-level factor (measured over time).
- Number of defects is a lower-level factor.

ISO 9126 :

- Defines a hierarchical set of quality characteristics.
- Each sub-characteristic is linked to one main quality characteristic.

Differences between McCall's and ISO 9126 :

- McCall's attributes are interrelated, whereas ISO 9126 focuses on distinct categories.
- ISO 9126 applies to software products, while McCall's model also considers process quality.

4. SOFTWARE QUALITY MANAGEMENT SYSTEM :

A Quality Management System (QMS) ensures that developed software products meet desired quality standards.

Important issues associated with a quality system:

Managerial Structure and Individual Responsibilities:

- Quality system is the responsibility of the entire organization.
- A separate quality department exists to handle quality-related activities.
- Top management support is crucial for the success of the quality system.
- Without management support, staff may not take quality processes seriously.

Quality System Activities :

- Auditing projects to check if processes are followed.
- Collecting process and product metrics to ensure quality goals are met.
- Reviewing the quality system to improve its effectiveness.
- Developing standards, procedures, and guidelines for consistency.
- Producing reports for top management to summarize system effectiveness.

Importance of Documentation :

- A well-documented quality system is necessary for consistency.
- Without proper documentation, quality controls become ad hoc, leading to variations in product quality.
- Undocumented systems create uncertainty about the organization's commitment to quality.
- International standards like ISO 9000 provide guidance on organizing a quality system

4.1 Evolution of Quality Systems :

Quality systems of organisations have undergone four stages of evolution

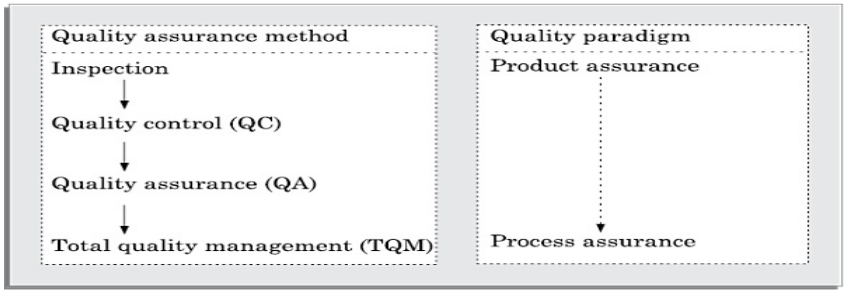


Figure 11.3: Evolution of quality system and corresponding shift in the quality paradigm.

The initial product inspection method gave way to quality control (QC) principles. Thus, quality control aims at correcting the causes of errors and not just rejecting the defective products. The next breakthrough in quality systems, was the development of the quality assurance (QA) principles.

4.2 Product Metrics versus Process Metrics :

Product metrics help measure the characteristics of a product being developed, whereas process metrics help measure how a process is performing.

- Examples of product metrics are LOC and function point to measure size, PM (person- month) to measure the effort required to develop it etc.
- Examples of process metrics are review effectiveness, average number of defects found per hour of inspection, average defect correction time, productivity, average number of failures detected during testing per LOC, number of latent defects per line of code in the developed product.

5 ISO 9000 :

- ISO (International Standards Organization) is a consortium of 63 countries that develop and promote standards.
- The ISO 9000 series of standards was introduced in 1987 to ensure quality in production and business processes.

5.1 What is ISO 9000 Certification?

ISO 9000 certification provides a reference for contracts between independent parties. The standard outlines guidelines for maintaining a quality system, covering both operational and organizational aspects like responsibilities and reporting. ISO 9000 is focused on ensuring repeatable and high-quality product development, not the product itself. **ISO 9000** consists of three standards:

- **ISO 9001**: Applies to organizations involved in design, development, production, and servicing of goods. It is relevant to most software development companies.
- **ISO 9002**: Applies to organizations that only handle production (e.g., steel and car manufacturing). Not applicable to software development.
- **ISO 9003**: Applies to organizations focused solely on installation and testing of products.

5.2 ISO 9000 for Software Industry :

ISO 9000 is a general standard applicable to various industries, from steel manufacturing to service companies. Many of its clauses use generic terms that are hard to interpret for software development organizations. This difficulty arises because software development is quite different from manufacturing other types of products. Two key differences include:

1. Software is intangible, making it difficult to control and manage. Unlike physical products like cars, where progress can be seen at different stages, software development is harder to track until it's fully functional.
2. Software uses data as its raw material, unlike other industries that consume large physical raw materials. For example, a steel company uses iron ore, coal, and other materials, but these concepts don't apply to software.

Due to these differences, ISO 9000's original clauses were challenging to apply to software development. In 1991, ISO released ISO 9000 Part-3 to provide guidance specific to the software industry. However, official guidance is still lacking, and organizations often need to cross-reference ISO 9000-3 for interpretation.

5.3 Why Get ISO 9000 Certification?

Software development organizations are eager to obtain ISO certification because of the numerous benefits it offers. Some key advantages include:

- **Increased customer confidence**: ISO 9001 certification boosts trust in an organization, especially in the international market. Many international clients require ISO 9000 certification for software development contracts, making it essential for software exporters.
- **Identification of weak points**: ISO 9000 highlights an organization's weaknesses and provides recommendations for improvement.
- **Framework for process development**: It establishes a foundation for creating optimal processes and Total Quality Management (TQM).
- **Well-documented processes**: ISO 9000 requires a documented software production process, ensuring repeatability and higher quality software development.
- **Improved efficiency**: The standard makes the development process more focused, efficient, and cost-effective.

5.4 How to Get ISO 9000 Certification?

To obtain ISO 9000 certification, an organization must apply to an ISO 9000 registrar. The registration process involves several stages:

1. **Application**: The organization applies for ISO 9000 certification.
2. **Pre-assessment**: The registrar conducts an initial assessment of the organization.
3. **Document review and audit**: The registrar reviews the organization's documents and suggests improvements.
4. **Compliance audit**: The registrar checks if the suggested improvements have been implemented.
5. **Registration**: After successful completion of previous stages, the ISO 9000 certificate is awarded.
6. **Continued surveillance**: The registrar conducts periodic monitoring of the organization.

5.5 Summary of ISO 9001 Requirements :

A summary of the main requirements of ISO 9001 as they relate of software development are as follows: *Section numbers in brackets correspond to those in the standard itself*:

- **Management Responsibility (4.1)**: Management must have a quality policy, define roles affecting quality, and appoint an independent quality system representative. Regular audits are required to review system effectiveness.
- **Quality System (4.2)**: A documented quality system must be maintained.
- **Contract Reviews (4.3)**: Contracts must be reviewed to ensure understanding and capability to fulfill obligations.
- **Design Control (4.4)**: The design process, including coding, must be controlled with proper configuration management. Design inputs must be verified, and outputs must meet required quality standards. Design changes must be controlled.
- **Document Control (4.5)**: Procedures for document approval, issuance, and changes must be in place, requiring configuration management tools.
- **Purchasing (4.6)**: Purchased materials, including software, must meet required standards.
- **Purchaser Supplied Product (4.7)**: Client-provided materials must be properly managed and checked.
- **Product Identification (4.8)**: Products must be identifiable at all stages, involving configuration management.
- **Process Control (4.9)**: The development process must be well-managed, with quality requirements identified in a plan.
- **Inspection and Testing (4.10)**: Effective testing (unit, integration, system) is required, along with maintaining test records.
- **Inspection and Test Equipment (4.11)**: If used, equipment must be maintained and calibrated.
- **Inspection and Test Status (4.12)**: The status of items must be identified through configuration management and release control.
- **Control of Non-Conforming Product (4.13)**: Faulty or untested software must be kept out of released products.
- **Corrective Action (4.14)**: Errors must be corrected, causes investigated, and processes improved to prevent recurrence.
- **Handling (4.15)**: This refers to the storage, packaging, and delivery of software.
- **Quality Records (4.16)**: Records must confirm quality control steps have been taken.
- **Quality Audits (4.17)**: Audits must ensure the quality system's effectiveness.
- **Training (4.18)**: Training needs must be identified and addressed.

5.6 Salient Features of ISO 9001 Requirements :

The key features of the ISO 9001 requirements are:

- **Document Control**: All development documents must be properly managed, authorized, and controlled, requiring a configuration management system.
- **Planning**: Plans must be prepared, and progress should be monitored.
- **Review**: Documents at each phase must be independently reviewed for effectiveness and accuracy.
- **Testing**: The product must be tested against its specifications.
- **Organizational Aspects**: Various organizational factors, such as management reporting for the quality team, must be addressed.

5.7 ISO 9000-2000 :

ISO revised its quality standards in 2000 to enhance them. Key changes include:

- A focus on continuous process improvement.
- Greater emphasis on the role of top management, with the requirement to establish measurable objectives at various organizational levels.
- Recognition that organizations may have multiple processes.

5.8 Shortcomings of ISO 9000 Certification :

While ISO 9000 is widely used for establishing quality systems, it has several shortcomings:

- **Process quality not guaranteed**: ISO 9000 requires adherence to a software production process but does not ensure the process is of high quality or provide guidelines for defining an appropriate process.
- **Inconsistent certification**: The ISO 9000 certification process is not foolproof, and no single international accreditation agency exists, leading to variations in certification standards among agencies and registrars.
- **Downplaying domain expertise**: Organizations may overemphasize the importance of process over developer expertise. Unlike manufacturing, where process quality directly correlates with product quality, software development requires specialized domain knowledge, and the skills and experience of developers significantly impact results.
- **No guarantee of continuous improvement**: ISO 9000 does not automatically lead to continuous process improvement or Total Quality Management (TQM).

6. SEI Capability Maturity Model (CMM):

Overview of SEI CMM :

- Proposed by the Software Engineering Institute (SEI) at Carnegie Mellon University, USA.
- Inspired by Philip Crosby's concept of quality maturity grid.
- Initially developed to assist the U.S. Department of Defense (DoD) in software acquisition.
- Helps evaluate and improve the software development process in organizations.
- Used for capability evaluation and software process assessment.
- Capability evaluation is for contract awarding authorities to assess vendor performance.
- Software process assessment is used internally by organizations for self-improvement.

Five Levels of SEI CMM :

level 1: **Initial**

- Ad-hoc and chaotic software development processes.
- Success depends on individual efforts.
- No formal project management practices.
- Leads to low-quality products due to time pressure and shortcuts.

Level 2: **Repeatable**

- Basic project management practices are established (cost, schedule tracking).
- Configuration management is introduced.
- Uses size and cost estimation techniques (e.g., Function Point Analysis, COCOMO).
- Success relies on process discipline, but documentation is minimal.
- Organizations can repeat successful processes for similar projects.

Level 3: **Defined**

- Standardized and documented processes for both management and development.
- Organization-wide understanding of activities, roles, and responsibilities.
- Employees receive training programs for skill development.
- Emphasizes peer reviews and process definitions.

Level 4: **Managed**

- Focuses on software metrics (quantitative process and product quality goals).
- Uses statistical tools like Pareto charts and fishbone diagrams.
- Ensures process stability and quality management.
- Project performance is evaluated using process metrics.

Level 5: **Optimizing**

- Emphasizes continuous process improvement.
- Uses feedback from process measurements for improvements.
- Identifies best software engineering practices and integrates innovations.
- Establishes a dedicated team for adopting new tools and methodologies.
- Uses configuration management techniques for process changes.

CMM Shortcomings :

- Organizations struggle with how to improve, even when they understand what needs improvement.
- Heavy documentation and long meetings make the process complex.
- Measuring an organization's true maturity level is difficult since it is activity-based rather than result-oriented.

6.1 Comparison Between ISO 9000 Certification and SEI/CMM :

Comparison of ISO 9000 certification and the SEI CMM model for quality appraisal:

- **ISO 9000** is awarded by an international standards body and can be used in official documents, external communication, and tender quotations. In contrast, **SEI CMM** assessments are for internal use only.
- **SEI CMM** is specifically designed for the software industry, addressing issues unique to this field, while ISO 9000 is a broader standard.
- **SEI CMM** goes beyond quality assurance, guiding organizations toward Total Quality Management (TQM). ISO 9001 aligns with **Level 3** of the SEI CMM model.
- **SEI CMM** provides a structured path for gradual quality improvement through key process areas (KPAs) at different maturity levels. **ISO 9000**, however, either qualifies an organization or does not.

6.2 Is SEI CMM Applicable to Small Organizations?

For small organizations often focused on small-scale applications like e-commerce or internet projects the CMM-based appraisal might be excessive. These organizations typically lack an established product range, revenue base, and experience. Instead, they need to focus on operating efficiently at lower maturity levels, practicing effective project management, reviews, and configuration management.

6.3 Capability Maturity Model Integration (CMMI) :

CMMI is the successor to the CMM. Developed between 1987 and 1997, CMMI Version 1.1 was released in 2002, followed by Version 1.2 in 2006. CMMI was designed to improve the usability of maturity models by integrating various models into a single framework.

After its initial release in 1990, CMMI was adopted across multiple domains, such as systems engineering (SE-CMM), people management (PCMM), and software acquisition (SA-CMM). While these models were useful, organizations faced challenges with overlap and integration issues. CMMI generalizes to apply across domains, with the term "software" absent in its definitions, making it more abstract. Like its predecessor, CMMI outlines five distinct levels of maturity.

7. FEW OTHER IMPORTANT QUALITY STANDARDS:

7.1 Software Process Improvement and Capability Determination (SPICE) :

- SPICE is an ISO standard (IEC 15504) for software process improvement and capability determination.
- It defines different process categories: engineering, management, customer-supplier, and support.
- Each process is assigned one of six capability maturity levels.
- Integrates existing standards to provide a single process reference model for enterprises.

7.2 Personal Software Process (PSP) :

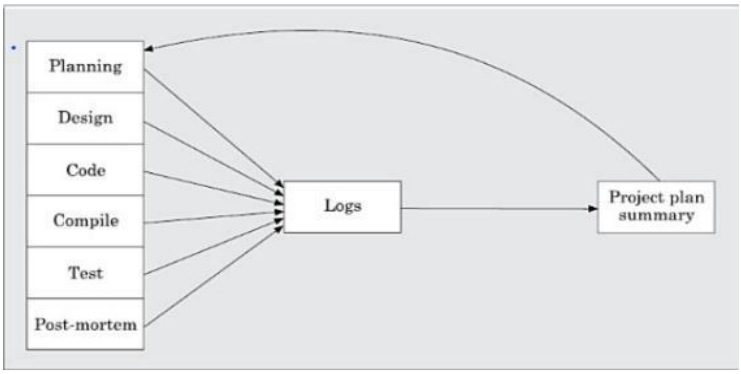
- Developed by David Humphrey, PSP is a scaled-down version of industrial software processes for individual use.
- Helps software engineers measure and improve their efficiency and personal practices.
- PSP focuses on estimating, planning, tracking performance, and refining processes.

Time Measurement:

- Developers should track how they spend time using a stopwatch.
- Helps separate productive time from distractions (e.g., calls, coffee breaks).
- Logs are used to analyze time spent on various tasks like design, coding, and testing.

PSP Planning :

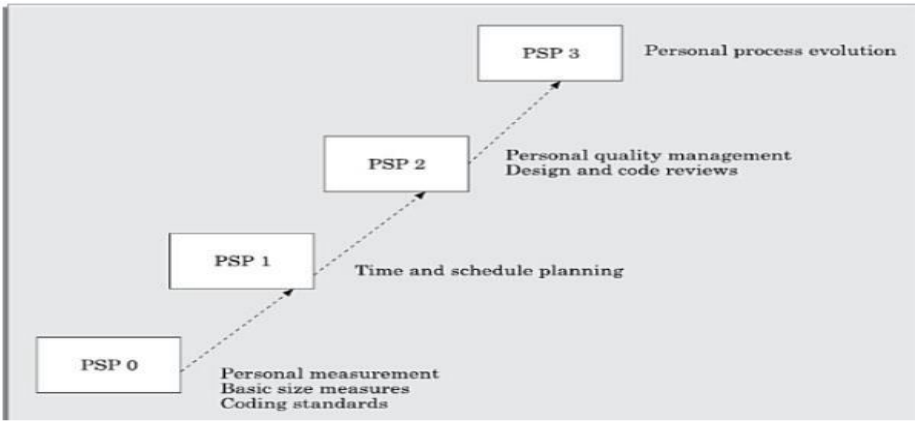
- Engineers must plan projects to avoid excessive time spent on trivial tasks.
- Developers estimate maximum, minimum, and average lines of code (LOC) required.
- Log data is used to refine future project



PSP Framework (Figure 11.4)

- Stages: Planning → Design → Code → Compile → Test → Post-mortem.
- Logs are maintained at each stage and compiled into a project summary

Levels of PSP (Figure 11.5) :



Levels of PSP (Figure 11.5)

- **PSP 0:** Personal measurement, basic size measures, coding standards.
- **PSP 1:** Time and schedule planning.
- **PSP 2:** Personal quality management, design & code reviews.
- **PSP 3:** Personal process evolution.

8. SIX SIGMA:

Origins & Purpose:

- General Electric (GE) adopted Six Sigma in 1995, following Motorola and Allied Signal.
- Six Sigma is designed to improve processes by making them faster, better, and more cost-effective.
- It applies to all business aspects, from production to human resources and technical support.

Concept & Application:

- Six Sigma aims for near perfection by using a disciplined, data-driven approach to eliminate defects.
- It applies to various industries, including manufacturing, transactional processes, and services.

Statistical Meaning:

- A Six Sigma process must have no more than 3.4 defects per million opportunities.
- A Six Sigma defect is any system behavior that does not meet customer specifications.

Methodology & Sub-methods:

- Six Sigma focuses on process improvement and variation reduction.
- It is implemented through two main sub-methodologies:
  - DMAIC (Define, Measure, Analyze, Improve, Control) – for improving existing processes.
  - DMADV (Define, Measure, Analyze, Design, Verify) – for developing new high-quality processes.