UNIT4:

Coding and Testing: Coding; code review; testing; testing in the large vs testing in the small; unit testing; black-box testing; white-box testing; debugging; program analysis tools; integration testing; system testing; some general issues associated with testing.

Software Reliability and Quality Management: Software reliability; statistical testing; software quality; software quality management system; SEI capability maturity model; personal software process.

Coding

- **Definition**: The process of converting design specifications into executable computer programs using programming languages.
- **Objectives**:
  - Translate software design into correct, efficient, and maintainable code.
  - Ensure adherence to coding standards and best practices.

Coding in Software Engineering

23 Mar 2025 | 5 min read

The coding is the process of transforming the design of a system into a computer language format. This coding phase of software development is concerned with software translating design specification into the source code. It is necessary to write source code & internal documentation so that conformance of the code to its specification can be easily verified.

Coding is done by the coder or programmers who are independent people than the designer. The goal is not to reduce the effort and cost of the coding phase, but to cut to the cost of a later stage. The cost of testing and maintenance can be significantly reduced with efficient coding.

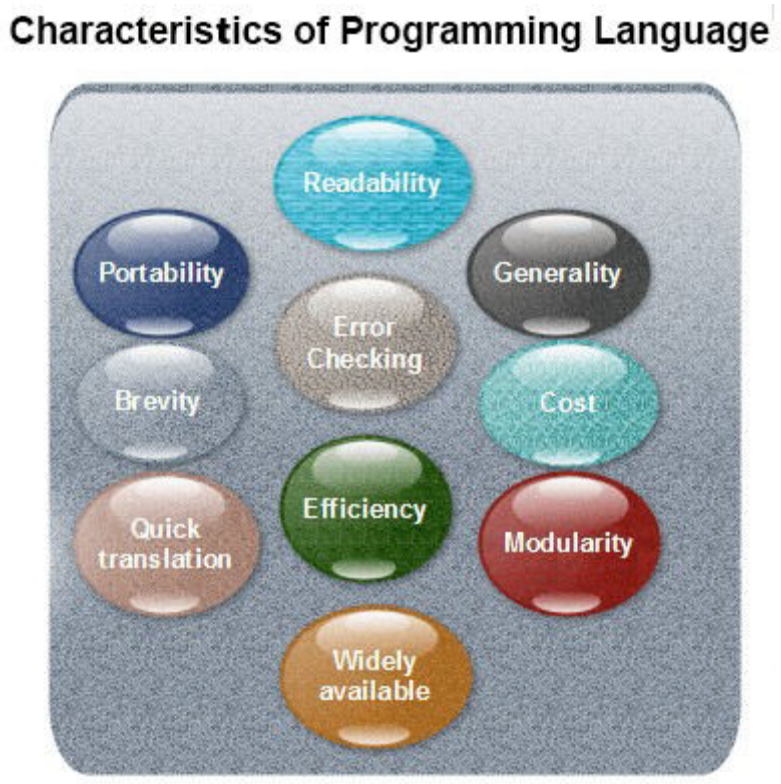
Goals of Coding

1. **To translate the design of system into a computer language format:** The coding is the process of transforming the design of a system into a computer language format, which can be executed by a computer and that perform tasks as specified by the design of operation during the design phase.
2. **To reduce the cost of later phases:** The cost of testing and maintenance can be significantly reduced with efficient coding.
3. **Making the program more readable:** Program should be easy to read and understand. It increases code understanding having readability and understandability as a clear objective of the coding activity can itself help in producing more maintainable software.

For implementing our design into code, we require a high-level functional language. A programming language should have the following characteristics:

Characteristics of Programming Language

Following are the characteristics of Programming Language:



Readability: A good high-level language will allow programs to be written in some methods that resemble a quite-English description of the underlying functions. The coding may be done in an essentially self-documenting way.

Portability: High-level languages, being virtually machine-independent, should be easy to develop portable software.

Generality: Most high-level languages allow the writing of a vast collection of programs, thus relieving the programmer of the need to develop into an expert in many diverse languages.

Brevity: Language should have the ability to implement the algorithm with less amount of code. Programs mean in high-level languages are often significantly shorter than their low-level equivalents.

Error checking: A programmer is likely to make many errors in the development of a computer program. Many high-level languages invoke a lot of bugs checking both at compile-time and run-time.

Cost: The ultimate cost of a programming language is a task of many of its characteristics.

Quick translation: It should permit quick translation.

Efficiency: It should authorize the creation of an efficient object code.

Modularity: It is desirable that programs can be developed in the language as several separately compiled modules, with the appropriate structure for ensuring self-consistency among these modules.

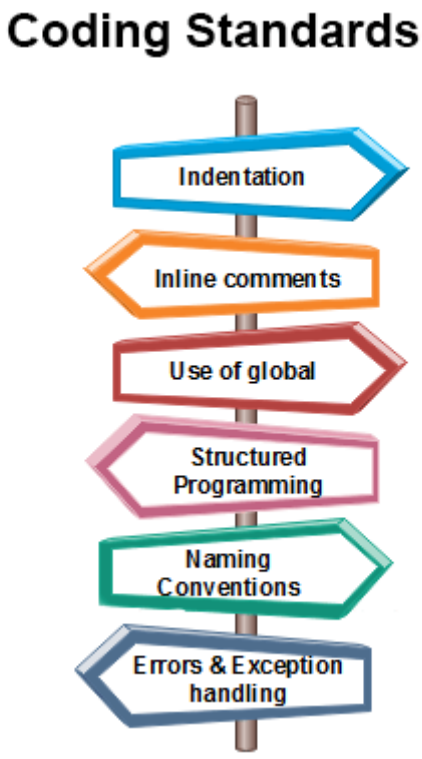
Widely available: Language should be widely available, and it should be feasible to provide translators for all the major machines and all the primary operating systems.

A coding standard lists several rules to be followed during coding, such as the way variables are to be named, the way the code is to be laid out, error return conventions, etc.

Coding Standards

General coding standards refers to how the developer writes code, so here we will discuss some essential standards regardless of the programming language being used.

The following are some representative coding standards:

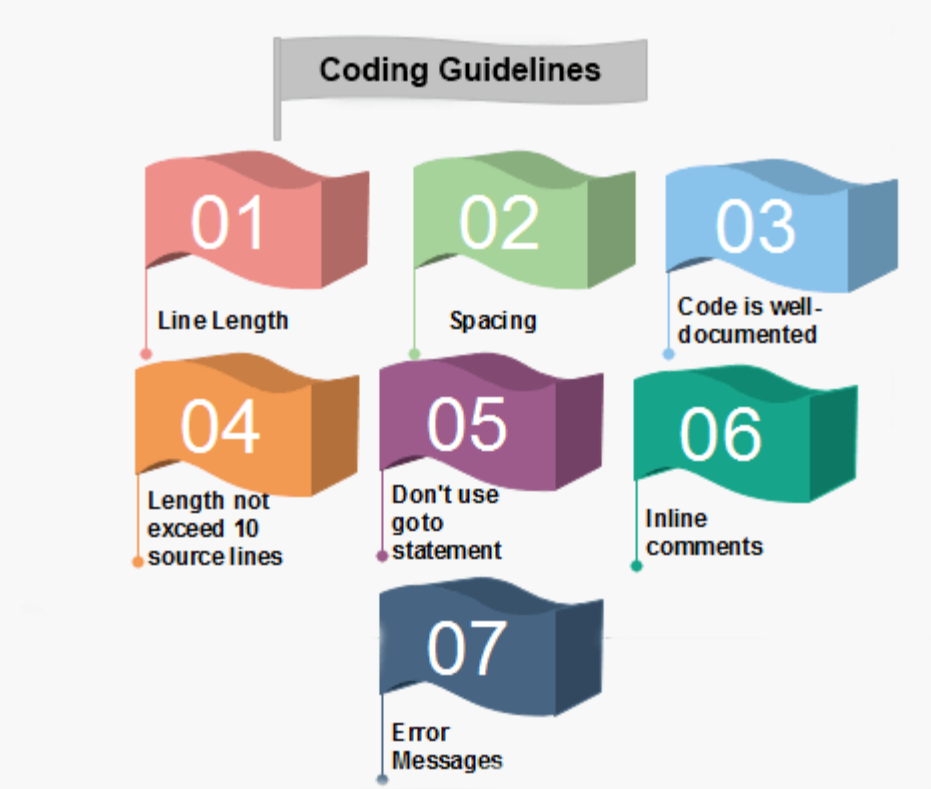


1. **Indentation:** Proper and consistent indentation is essential in producing easy to read and maintainable programs. Indentation should be used to:
    - Emphasize the body of a control structure such as a loop or a select statement.
    - Emphasize the body of a conditional statement
    - Emphasize a new scope block
2. **Inline comments:** Inline comments analyze the functioning of the subroutine, or key aspects of the algorithm shall be frequently used.
3. **Rules for limiting the use of global:** These rules file what types of data can be declared global and what cannot.
4. **Structured Programming:** Structured (or Modular) Programming methods shall be used. "GOTO" statements shall not be used as they lead to "spaghetti" code, which is hard to read and maintain, except as outlined line in the FORTRAN Standards and Guidelines.
5. **Naming conventions for global variables, local variables, and constant identifiers:** A possible naming convention can be that global variable names always begin with a capital letter, local variable names are made of small letters, and constant names are always capital letters.
6. **Error return conventions and exception handling system:** Different functions in a program report the way error conditions are handled should be standard within an organization. For example, different tasks while encountering an error condition should either return a 0 or 1 consistently.

Coding Guidelines

General coding guidelines provide the programmer with a set of the best methods which can be used to make programs more comfortable to read and maintain. Most of the examples use the C language syntax, but the guidelines can be tested to all languages.

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



1. Line Length: It is considered a good practice to keep the length of source code lines at or below 80 characters. Lines longer than this may not be visible properly on some terminals and tools. Some printers will truncate lines longer than 80 columns.

2. Spacing: The appropriate use of spaces within a line of code can improve readability.

Example:

Bad:
```
cost=price+(price*sales_tax)
fprintf(stdout ,"The total cost is %5.2f\n",cost);
```

Better:
```
cost = price + ( price * sales_tax )
fprintf (stdout,"The total cost is %5.2f\n",cost);
```

3. The code should be well-documented: As a rule of thumb, there must be at least one comment line on the average for every three-source line.

4. The length of any function should not exceed 10 source lines: A very lengthy function is generally very difficult to understand as it possibly carries out many various functions. For the same reason, lengthy functions are possible to have a disproportionately larger number of bugs.

5. Do not use goto statements: Use of goto statements makes a program unstructured and very tough to understand.

6. Inline Comments: Inline comments promote readability.

7. Error Messages: Error handling is an essential aspect of computer programming. This does not only include adding the necessary logic to test for and handle errors but also involves making error messages meaningful.

What is Code Review?

The code review is a methodical process where a group of developers work together to analyze and check another developers code to detect errors, give suggestions, and confirm if the developed code is as per the standards. The objective of code review is to enhance the quality, maintainability, stability, security etc of the software which bring positive results to the project. Also, the findings from the code review promote sharing knowledge and learnings among the team members.

Why is Code Review Done?

The code review is done for the reasons listed below ?

- It helps to detect errors, defects, issues etc in the code prior to being deployed to production. Thus a code review helps to fix bugs at the initial phases of software development life cycle (SDLC).
- It motivates developing clean, maintainable, and effective code. The reviewers pass feedback and comments so that the code is as per the standards and best practices.
- It implements consistency in coding among all the developers which enables easy maintenance and understanding of the code base.
- The findings from the code reviews can be shared across teams which propagate domain knowledge and coding guidelines.
- The code reviewers take partial ownership of the code they review thereby increasing collective responsibility towards ensuring quality.
- The code reviewers can work together and collaborate to improve the entire review process which helps in enhancing the overall software quality.
- It can be a part of documentation.
- It is an integral part of ensuring the software quality. By doing code reviews, the team can confirm if the software meets all the functional and non-functional requirements.
- It helps to adopt continuous improvement for the team. By following the suggestions, feedback, and findings from the code review, the team can work up on them, then gradually improve.

Types of Code Review

The types of code reviews are listed below

- **Pull Requests (PR)**

  In Git, the developers raise a PR to incorporate changes to the code. It should be reviewed prior to the changes being merged with the base code.

- **Pair-Programming**

It is a type of review in which two developers work on the same computer. One of them writes the code and the other one reviews it in real time. It is a highly interactive form of code review.

- **Over the Shoulder Review**

It is the type of review in which one developer in the team is requested to review the code of another developer by sitting together and going through the code on the computer.

- **Tool Aided Reviews**

It is a type of review conducted by tools like Github, GitLab, BitBucket, Crucible etc.

- **Email Based Reviews**

It is a type of review in which the code changes sent over email for review. The feedback of the code review is also delivered in email.

- **Checklist Reviews**

It is a type of review in which the reviewers follow the list of checklist items for the review process.

- **Ad Hoc Review**

It is an informal way of review. A developer may be requested to have a quick look at the code and provide feedback not formally.

conform to the coding standards is rejected during code review and the code is reworked by the concerned programmer.

1. 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.

Usually code review is carried out to ensure that the coding standards are followed and also to detect as many errors as possible before testing. Reviews are an efficient way of removing errors from code.

Coding Standards and Guidelines

Good software development organizations usually develop their own coding standards and guidelines.

Representative Coding Standards

- **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.
- **Standard headers for different modules:** The header of different modules should have a standard format and information for ease of understanding and maintenance.
- **Naming conventions for global variables, local variables, and constant identifiers:** A popular naming convention is that variables are named using mixed case lettering.
    - Example: `GlobalData`, `localData`, `CONSTDATA`
- **Conventions regarding error return values and exception handling mechanisms:** The way error conditions are reported by different functions in a program should be standard within an organisation.

Representative Coding Guidelines

- **Do not use a coding style that is too clever or too difficult to understand:** Code should be easy to understand. Many inexperienced engineers actually take pride in writing cryptic and incomprehensible code.
- **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.
- **Do not use an identifier for multiple purposes:** Programmers often use the same identifier to denote several temporary entities. There are several things wrong with this approach and hence it should be avoided.
- **Code should be well-documented:** As a rule of thumb, there should be at least one comment line on the average for every three source lines of code.

- **Length of any function should not exceed 10 source lines:** A lengthy function is usually very difficult to understand as it probably has a large number of variables and carries out many different types of computations.
- **Do not use GOTO statements:** Use of GOTO statements makes a program unstructured. This makes the program very difficult to understand, debug, and maintain.

Code Review

Code Review

1. Importance of Testing vs. Review

- **Testing** is an effective defect removal mechanism. However, it applies only to **executable code**.
- **Review** is a very effective technique to remove defects from **source code**.
- Review has been acknowledged to be **more cost-effective** than testing in removing defects.

2. When Code Review is Done

- Undertaken **after the module successfully compiles** (all syntax errors have been eliminated).
- Focuses on detecting **logical, algorithmic, and programming errors**, not syntax errors.
- Recognised as an **extremely cost-effective strategy** for eliminating coding errors and producing **high-quality code**.
- **Difference:**
  - Reviews **directly detect errors**.
  - Testing only **detects failures**.

3. Activities for Error Elimination

- **Testing** – checks whether the system fails for certain inputs or conditions.
- **Debugging** – locates the specific error causing failure.
- **Correcting** – fixes the error once located.

Debugging is usually the **most laborious and time-consuming** activity.

4. Code Inspection

- Errors are directly detected, saving the effort required to locate them.
- Code inspection examines the code for **common programming errors**.
- **Principal aims:**
  - Detect common programmer mistakes and oversights.
  - Ensure adherence to **coding standards**.

Beneficial Side Effects:

- Programmers receive feedback on:
  - Programming style

    - Choice of algorithm
    - Programming techniques
- Other participants learn from being exposed to another programmer's errors.

5. Types of Reviews

- **Code Inspection**
- **Code Walkthrough**

- **Testing** is an effective defect removal mechanism. However, testing is applicable only to executable code.
- **Review** is a very effective technique to remove defects from source code. In fact, review has been acknowledged to be more cost-effective in removing defects as compared to testing.
- Code review for a module is undertaken **after the module successfully compiles.** That is, all the syntax errors have been eliminated from the module.
- Code review does not target design syntax errors in a program, but is designed to detect **logical, algorithmic, and programming errors.**
- Code review has been recognised as an **extremely cost-effective strategy** for eliminating coding errors and for producing high quality code.
- Reviews **directly detect errors**, whereas testing only helps detect failures.

Eliminating an error from code involves three main activities:

- **Testing** – carried out to detect if the system fails to work satisfactorily for certain types of inputs and under certain circumstances.
- **Debugging** – carried out to locate the error that is causing the failure.
- **Correcting the error** – fixing the code after locating the bug.

Of the three activities, debugging is possibly the most laborious and time-consuming.

In **code inspection**, errors are directly detected, thereby saving significant effort that would have been required to locate the error.

Normally, the following two types of reviews are carried out on the code:

- **Code Inspection**
- **Code Walkthrough**

Code Inspection

- During code inspection, the code is examined for the presence of some common programming errors.
- The principal aim of code inspection is to check for the presence of **common types of errors** that usually creep into code due to programmer mistakes and oversights, and to verify whether coding standards have been adhered to.
- The inspection process has several **beneficial side effects** other than finding errors:

- The programmer usually receives feedback on programming style, choice of algorithm, and programming techniques.
- The other participants gain by being exposed to another programmer's errors.

Code Walkthrough: An Informal Yet Insightful Review Method

Code walkthrough is a less formal approach to code review, where the author leads a meeting to explain their work to peers or supervisors. The primary goals of a code walkthrough are:

- Sharing knowledge about the code and its functionality
- Gathering feedback and alternative perspectives
- Identifying potential issues or areas for improvement
- Fostering collaboration and team cohesion

During a code walkthrough, the author guides the attendees through the code, explaining the logic, design decisions, and implementation details. Participants can ask questions, offer suggestions, and discuss potential improvements.

While code walkthroughs may lack the structured approach of formal inspections, they offer valuable insights and promote a shared understanding of the codebase among team members.

Software Documentation

When software is developed, in addition to the executable files and source code, several kinds of documents are also produced as part of the software engineering process. Examples include: **User's Manual, Software Requirements Specification (SRS), Design Document, Test Document, Installation Manual**, etc.

Importance of Good Documentation

Good documents are helpful in the following ways:

- Enhance the **understandability** of code.
- Help users to **understand and effectively use** the system.
- Help in tackling the **manpower turnover problem**.
- Assist managers in effectively **tracking the progress** of the project.

What is software documentation?

In the software development process, software documentation is the information that describes the product to the people who develop, deploy and use it.

It includes the technical manuals and online material, such as online versions of manuals and help capabilities. The term is sometimes used to refer to source information about the product discussed in design documentation, code comments, white papers and session notes.

Software documentation is a way for engineers and programmers to describe their product and the process they used in creating it in formal writing. Early computer users were sometimes simply given the engineers' or programmers' notes. As software development became more complicated and formalized, technical writers and editors took over the documentation process.

Software documentation shows what the software developers did when creating the software and what IT staff and users must do when deploying and using it. Documentation is often incorporated into the software's user interface and also included as part of help documentation. The information is often divided into task categories, including the following:

- evaluating
- planning
- setting up or installing
- customizing
- administering
- using
- maintaining

Why is software documentation important?

Software documentation provides information about a software program for everyone involved in its creation, deployment and use. Documentation guides and records the development process. It also assists with basic tasks such as installation and troubleshooting.

Effective documentation gets users familiar with the software and makes them aware of its features. It can have a significant role in driving user acceptance. Documentation can also reduce the burden on support teams, because it gives users the power to troubleshoot issues.

Software documentation can be a living document that is updated over the software development lifecycle. Its use and the communication it encourages with users provides developers with information on problems users have with the software and what additional features they need. Developers can respond with software updates, improving customer satisfaction and user experience.

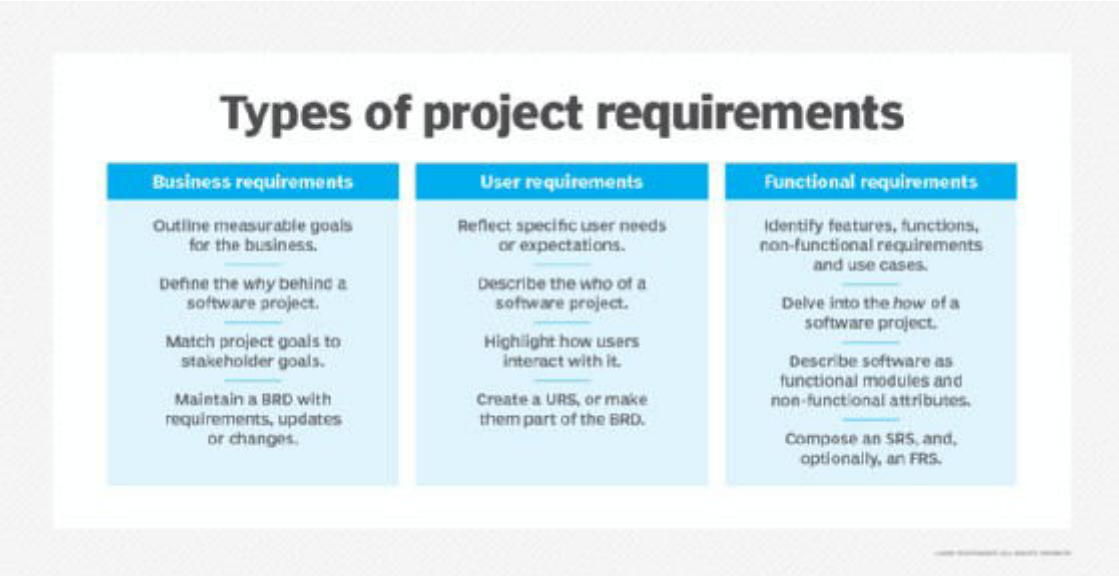
Types of software documentation

The two main types of software documentation are internal and external.

Internal software documentation

Developers and software engineers create internal documentation that is used inside a company. Internal documentation may include the following:

- **Administrative documentation.** This is the high-level administrative guidelines, roadmaps and product requirements for the software development team and project managers working on the software. It also may include status reports and meeting notes.
- **Developer documentation.** This provides instructions to developers for building the software and guides them through the development process. It includes requirements documentation, which describes how the software should perform when tested. It also includes architectural documentation that focuses on how all the components and features work together, and details data flows throughout the product.



Software requirements are detailed in internal software documentation.

External software documentation

Software developers create this documentation to provide IT managers and end users with information on how to deploy and use the software. External documentation includes the following:

- **End-user documentation**. This type gives end users basic instructions on how to use, install and troubleshoot the software. It might provide resources, such as user guides, knowledge bases, tutorials and release notes.

- **Enterprise user documentation.** Enterprise software often has documentation for IT staff who deploy the software across the enterprise. It may also provide documentation for the end users of the software.
- **Just-in-time documentation.** This provides end users with support documentation at the exact time they will need it. This allows developers to create a minimal amount of documentation at the release of a software product and add documentation as new features are added.

1. Testing in the Small

- **Definition:** Testing applied to small, isolated units or components of a software system.
- **Scope:** Limited to **modules, functions, classes, or small subsystems**.
- **Focus:**
  - Correctness of individual units.
  - Detecting local errors (logic errors, boundary conditions, incorrect calculations).
  - Ensuring that code behaves as expected in isolation.
- **Examples:**
  - **Unit Testing** (testing individual functions or methods).
  - **Module Testing** (testing a collection of closely related classes or functions).
- **Tools/Methods:**
  - JUnit, PyTest, NUnit, etc.
  - White-box techniques (control flow, path testing).

2. Testing in the Large

- **Definition:** Testing applied to the **complete integrated system** or **large subsystems**.
- **Scope:** Involves interactions among multiple components, subsystems, or the entire software.
- **Focus:**
  - Integration of components.
  - Performance, scalability, security, and reliability.
  - Detecting errors from component interaction, system environment, and real-world usage.
- **Examples:**
  - **Integration Testing** (testing combined modules).
  - **System Testing** (testing the entire software against requirements).
  - **Acceptance Testing** (validating with end-users).
- **Tools/Methods:**
  - Selenium, JMeter, LoadRunner, Postman, etc.
  - Black-box techniques (functional testing, stress testing).

Key Difference Table

Aspect	Testing in the Small	Testing in the Large
Scope	Small units/modules	Entire system or large subsystems
Focus	Internal correctness (logic/algorithms)	External behavior, integration, performance
Error Detection	Local bugs, boundary conditions	Interface errors, system-level faults

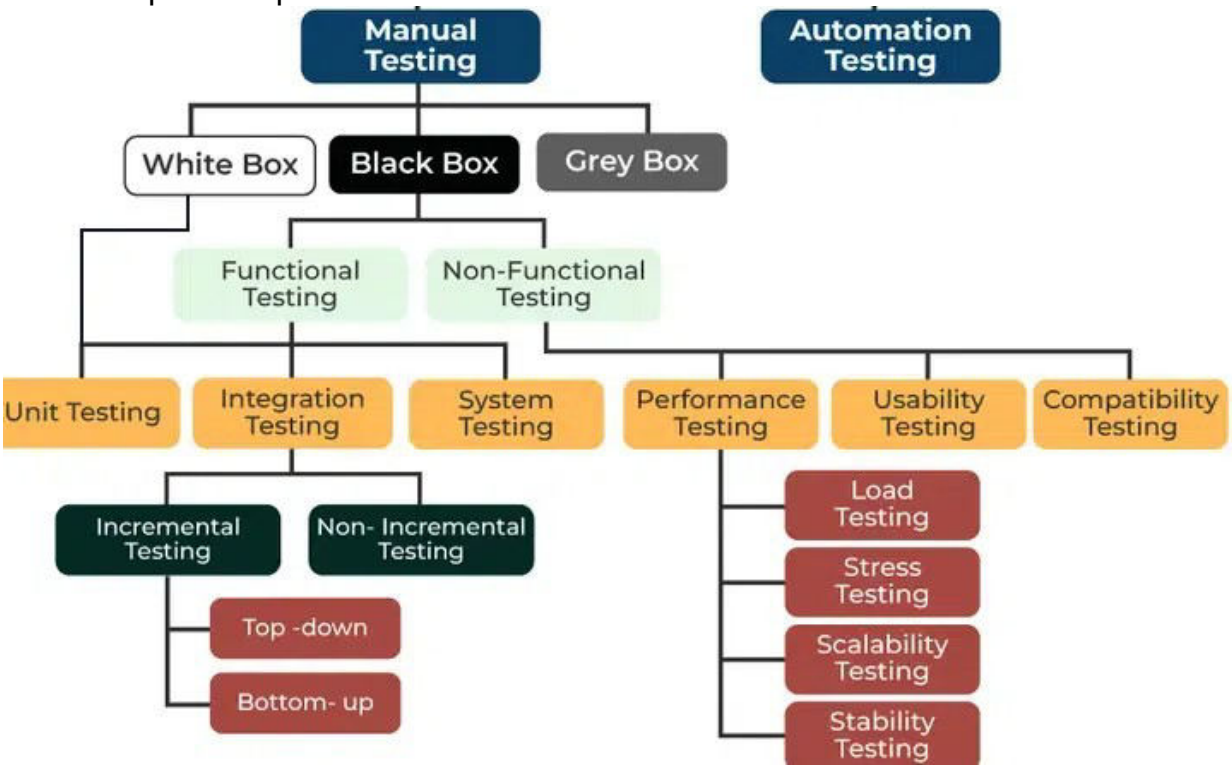
Aspect	Testing in the Small	Testing in the Large
Testing Techniques	Unit tests, white-box testing	Integration, system, acceptance, black-box
Tools	JUnit, PyTest, NUnit	Selenium, JMeter, LoadRunner

System Testing

System Testing is a type of software testing that is performed on a completely integrated system to evaluate the compliance of the system with the corresponding requirements. In system testing, integration testing passed components are taken as input.

- The goal of integration testing is to detect any irregularity between the units that are integrated. System testing detects defects within both the integrated units and the whole system. The result of system testing is the observed behavior of a component or a system when it is tested.
- System Testing is carried out on the whole system in the context of either system requirement specifications or functional requirement specifications or the context of both. System testing tests the design and behavior of the system and also the expectations of the customer.
- It is performed to test the system beyond the bounds mentioned in the software requirements specification (SRS).

There are different levels during the process of testing. In this chapter, a brief description is provided about these levels.



Levels of testing include different methodologies that can be used while conducting software testing. The main levels of software testing are:

- Functional Testing
- Non-functional Testing

Functional Testing

This is a type of black-box testing that is based on the specifications of the software that is to be tested. The application is tested by providing input and then the results are examined that need to conform to the functionality it was intended for. Functional testing of a software is conducted on a complete, integrated system to evaluate the system's compliance with its specified requirements.

There are five steps that are involved while testing an application for functionality.

I The determination of the functionality that the intended application is meant to perform.

II The creation of test data based on the specifications of the application.

III The output based on the test data and the specifications of the application.

IV The writing of test scenarios and the execution of test cases.

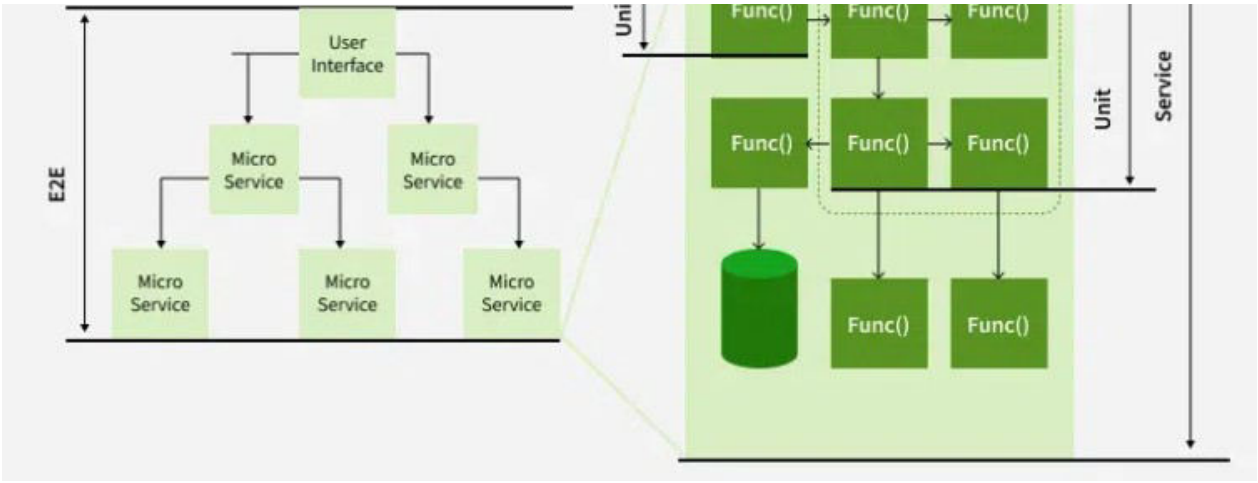
V The comparison of actual and expected results based on the executed test cases.

An effective testing practice will see the above steps applied to the testing policies of every organization and hence it will make sure that the organization maintains the strictest of standards when it comes to software quality.

Unit Testing

This type of testing is performed by developers before the setup is handed over to the testing team to formally execute the test cases. Unit testing is performed by the respective developers on the individual units of source code assigned areas. The developers use test data that is different from the test data of the quality assurance team.

The goal of unit testing is to isolate each part of the program and show that individual parts are correct in terms of requirements and functionality.



Limitations of Unit Testing

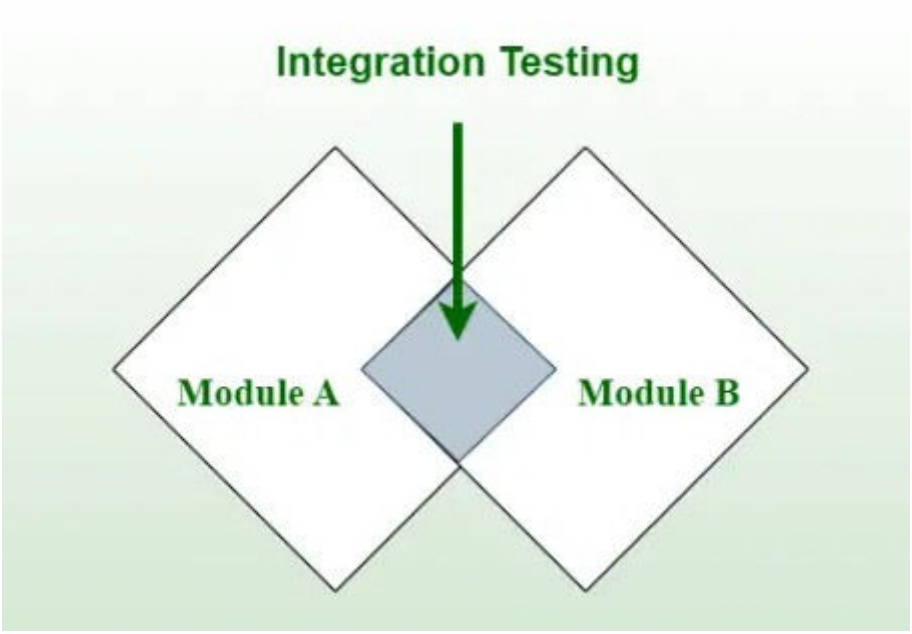
Testing cannot catch each and every bug in an application. It is impossible to evaluate every execution path in every software application. The same is the case with unit testing.

There is a limit to the number of scenarios and test data that a developer can use to verify a source code. After having exhausted all the options, there is no choice but to stop unit testing and merge the code segment with other units.

Integration Testing

Integration testing is defined as the testing of combined parts of an application to determine if they function correctly. Integration testing can be done in two ways: Bottom-up integration testing

and Top-down integration testing.



1
Bottom-up integration

This testing begins with unit testing, followed by tests of progressively higher-level combinations of units called modules or builds.

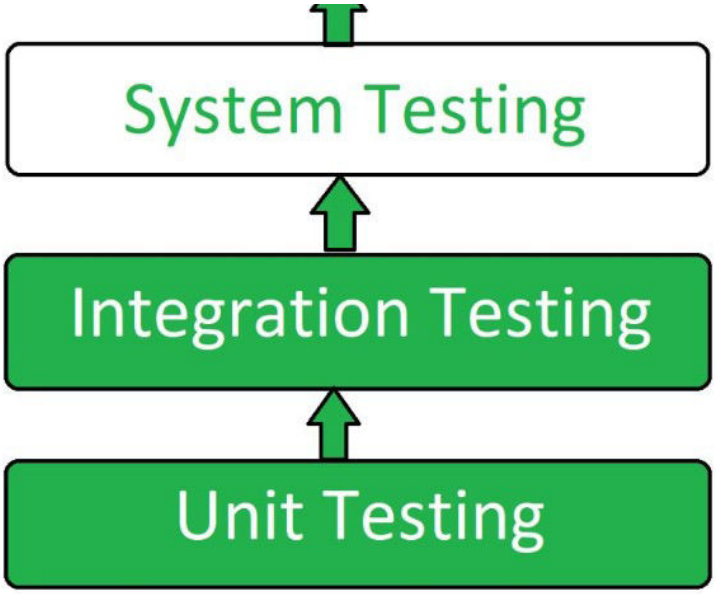
2
Top-down integration

In this testing, the highest-level modules are tested first and progressively, lower-level modules are tested thereafter.

In a comprehensive software development environment, bottom-up testing is usually done first, followed by top-down testing. The process concludes with multiple tests of the complete application, preferably in scenarios designed to mimic actual situations.

System Testing

System testing tests the system as a whole. Once all the components are integrated, the application as a whole is tested rigorously to see that it meets the specified Quality Standards. This type of testing is performed by a specialized testing team.

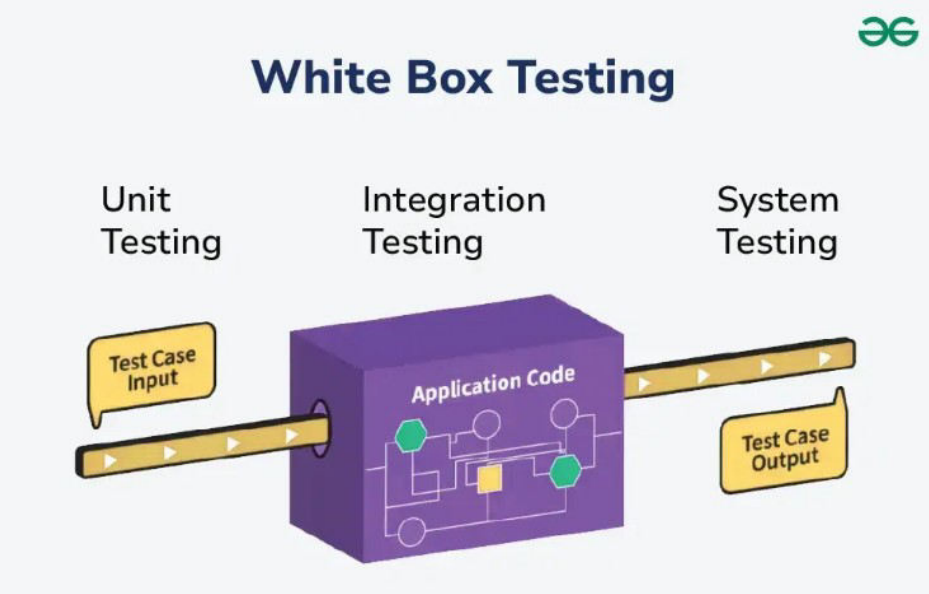


System testing is important because of the following reasons:

- System testing is the first step in the Software Development Life Cycle, where the application is tested as a whole.
- The application is tested thoroughly to verify that it meets the functional and technical specifications.
- The application is tested in an environment that is very close to the production environment where the application will be deployed.
- System testing enables us to test, verify, and validate both the business requirements as well as the application architecture.

White-Box Testing

White-box testing (also called **clear box testing**, **structural testing**, or **glass box testing**) is a **software testing technique** that examines the **internal structure, logic, and code** of a program. Unlike black-box testing (which focuses on functionality without looking at the code), white-box testing ensures that **every path, condition, and statement in the code is tested**.



Key Features

- Focuses on **internal logic, code structure, and flow**.
- Requires knowledge of **programming** and **system design**.
- Ensures **code coverage** (statements, branches, paths, conditions).
- Detects **hidden errors, security vulnerabilities, and inefficient code**.

Common Techniques

1. **Statement Coverage** – Ensures every line of code executes at least once.
2. **Branch Coverage** – Ensures every decision (true/false) is tested.
3. **Path Coverage** – Tests all possible execution paths in the program.
4. **Condition Coverage** – Ensures all logical conditions are evaluated both true and false.

Advantages

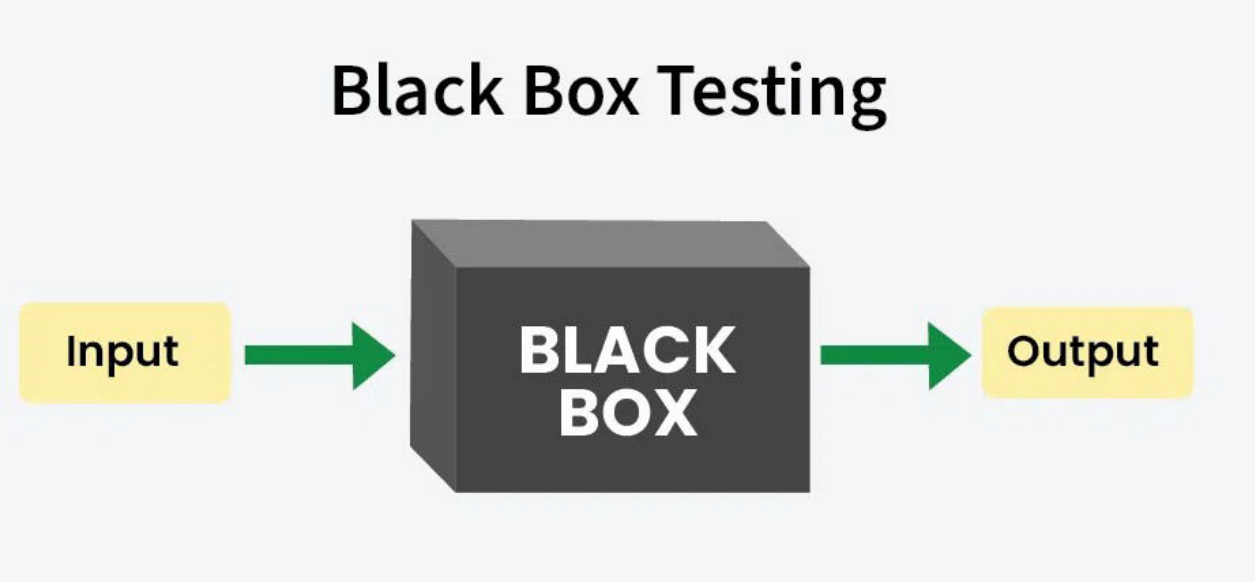
- Helps in **optimizing code** by identifying redundant or dead code.
- Detects **logical errors** early.
- Ensures maximum **test coverage**.

Disadvantages

- Requires **skilled testers** with coding knowledge.
- Can be **time-consuming** for large systems.
- Not effective for **higher-level testing** like user acceptance.

Black-Box Testing

Black-box testing (also called **behavioral testing** or **functional testing**) is a software testing technique that **focuses on the functionality of the application** without looking at its internal code or structure. The tester only knows the **inputs and expected outputs**, not how the system processes them.



Key Features

- Based on **requirements and specifications**.
- Tester does **not need programming knowledge**.
- Validates the **external behavior** of the system.

Common Techniques

1. **Equivalence Partitioning** – Divides input data into valid and invalid classes.
2. **Boundary Value Analysis** – Tests values at the edges (min/max limits).
3. **Decision Table Testing** – Uses rules and conditions to design tests.
4. **State Transition Testing** – Validates system behavior for different states/events.
5. **Use Case Testing** – Based on user scenarios.

Advantages

- Easy to design test cases.
- Detects **missing functionality**.
- Tester's perspective is **close to the end-user**.

Disadvantages

- Cannot detect **hidden errors in code logic**

- Limited coverage (only tests what is visible externally).
- Redundant test cases may occur.

Program Analysis Tools

Definition

- **Program Analysis Tool**: Automated tool that takes source/executable code as input and outputs characteristics like **size, complexity, commenting, standards compliance**, etc.
- Helps in **understanding, improving, and maintaining software systems** across the SDLC.

Importance

1. **Fault & Security Detection** – Finds bugs, vulnerabilities (buffer overflow, injection, etc.).
2. **Memory Leak Detection** – Identifies inefficiencies in memory usage.
3. **Dependency Analysis** – Helps manage relationships among system modules.
4. **Automated Testing Support** – Integrates with CI/CD pipelines for quality assurance.

Classification

1. **Static Program Analysis Tools**
    - Analyze **code without execution**.
    - Checks coding standards, uninitialized variables, unused variables, parameter mismatch.
    - Examples: **Compiler, linters, code inspectors**.
    - Methods: **Code walkthroughs, code inspections**.
2. **Dynamic Program Analysis Tools**
    - Analyze **code during execution**.
    - Insert extra statements to trace execution.
    - Provide reports on **statement, branch, path coverage**.
    - Results often in **charts (histogram, pie chart)**.
    - Used in **white-box testing**.
    - Helps in **eliminating redundant test cases**.

General Issues in Software Testing

1. **Exhaustive Testing is Impossible**
    - It is not feasible to test all possible inputs, paths, and scenarios due to **time and cost constraints**.
2. **Time and Cost Constraints**
    - Testing often has to be completed within tight deadlines and limited budgets, which may reduce coverage.
3. **Incompleteness of Testing**
    - Even after extensive testing, **all defects cannot be guaranteed to be found**.
4. **Evolving Requirements**
    - Changing user requirements during development make it difficult to create stable and reusable test cases.
5. **Dependence on Human Factors**
    - Test case design, execution, and interpretation are prone to **human errors**.
6. **Test Environment Limitations**

- Testing may not exactly replicate the **real-world environment**, leading to undetected issues.

7. **Maintenance of Test Cases**
   - As software evolves, **test cases also need updates**, which adds overhead.
8. **Difficulty in Automating All Tests**
   - Not all scenarios can be automated, especially **usability and exploratory testing**.
9. **Regression Testing Overhead**
   - Frequent code changes require repetitive regression tests, which can be **time-consuming and costly**.
10. **Measuring Test Effectiveness**

- It is hard to measure whether enough testing has been done and if the software is truly ready for release.

Only for BSC CS

Software Reliability and Quality Management: Software reliability; statistical testing; software quality; software quality management system; SEI capability maturity model; personal software process.

Software reliability

Definition 1: The reliability of a software product essentially denotes its trustworthiness or dependability.

Definition 2: The reliability of a software product can also be defined as the probability of the working "correctly" over a given period of time.

- The 10 percent instructions are often called the core1 of a program. The rest 90 per cent of the program statements are called non-core working "correctly" over a given period of time.
- We can say that reliability of a product depends not only on the number of latent errors but also on the exact location of the errors.

A. 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.
- Example: A logic gate may be stuck at 1 or 0, or a resistor might short circuit. To fix a hardware fault, one has to either replace or repair the failed part. In contrast, a software product would continue to fail until the error is tracked down and either the design or the code is changed to fix the bug.
- For this reason, when a hardware part is repaired its reliability would be maintained at the level that existed before the failure occurred; whereas when a software failure is repaired, the reliability may either increase or decrease (reliability may decrease if a bug fix introduces new errors).
- A comparison of the changes in failure rate over the product life time for a typical hardware product as well as a software product is sketched in Figure 11.1. Observe that the plot of change of reliability with time for a hardware component (Figure 11.1(a)) appears like a "bath tub".
- For a software component the failure rate is initially high, but decreases as the faulty components identified are either repaired or replaced.

B. Reliability Metrics of Software Products

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. Let the failures occur at the time instants t1, t2, …, tn. Then, MTTF can be calculated as

3. Mean time to repair (MTTR): Once failure occurs, some time is required to fix the error.
4. Mean time between failures (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.

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.

Ex: The Jelinski-Moranda (J-M) model is one of the earliest software reliability models. The reliability increases by a constant increment each time an error is detected and repaired.

The program failure rate at the ith failure interval is given by
Where
φ = a proportional constant, the contribution any one fault makes to the overall program
N = the number of initial faults in the program
ti = the time between the (i-1)th and the (i)th failures.

Ex: 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.

statistical testing

Statistical testing is a testing process whose objective is to determine the reliability of the product rather than discovering errors.

Steps in Statistical Testing:

1) The first step is to determine the operation profile of the software.
2) The next step is to generate a set of test data corresponding to the determined operation profile.
3) The third step is to apply the test cases to the software and record the time between each failure.
4) After a statistically significant number of failures have been observed, the reliability can be computed.

software quality; software quality management system

Software Quality: Traditionally, the quality of a product is defined in terms of its fitness of purpose. That is, a good quality product does exactly what the users want it to do.

The modern view of a quality associates with a software product several quality factors (or attributes) such as the following:

1) Portability: if it can be easily made to work in different hardware and operating system environments.
2) Usability: if different categories of users (i.e., both expert and novice users) can easily invoke the functions of the product.
3) Reusability: if different modules of the product can easily be reused to develop new products.
4) Correctness: if different requirements as specified in the SRS document.
5) Maintainability: if errors can be easily corrected, new functions can be easily added to the product, and the functionalities of the product can be easily modified, etc.

- Quality Management System: (often referred to as quality system) is the principal methodology used by organizations to ensure that the products they develop have the desired quality.
- Quality system activities:
  The quality system activities encompass the following:
    - ✓ Auditing of projects to check if the processes are being followed.
    - ✓ Collect process and product metrics and analyse them to check if quality goals are being met.
    - ✓ Review of the quality system to make it more effective.
    - ✓ Development of standards, procedures, and guidelines.
    - ✓ Produce reports for the top management summarising the effectiveness of the quality system in the organisation.
- Evolution of Quality Systems
    - o Quality control (QC) focuses not only on detecting the defective products and eliminating them, but also on determining the causes behind the defects, so that the product rejection rate can be reduced.
    - o 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.

- Total quality management (TQM) goes a step further than quality assurance and aims at continuous process improvement.
- 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.

- International standards organisation (ISO) is a consortium of 63 countries established to formulate and foster standardisation. ISO published its 9000 series of standards in 1987.
- ISO 9000 standard specifies the guidelines for maintaining a quality system. The ISO 9000 series of standards are based on the premise that if a proper process is followed for production, then good quality products are bound to follow automatically.
- ISO 9000 certification serves as a reference for contract between independent parties
- ISO 9000 is a series of three standards—ISO 9001, ISO 9002, and ISO9003.

ISO 9001: This standard applies to the organisations engaged in design, development, production, and servicing of goods. This is the standard that is applicable to most software development organisations.

ISO 9002: This standard applies to those organisations which do not design products but are only involved in production. Examples of this category of industries include steel and car manufacturing industries who buy the product and plant designs from external sources and are involved in only manufacturing those products. Therefore, ISO 9002 is not applicable to software development organisations.

ISO 9003: This standard applies to organisations involved only in installation and testing of products.

Why Get ISO 9000 Certification?

There is a mad scramble among software development organisations for obtaining ISO certification due to the benefits it offers. Let us examine some of the benefits that accrue to organisations obtaining ISO certification:

1. Confidence of customers in an organisation increases when the organisation qualifies for ISO 9001 certification.
2. ISO 9000 requires a well-documented software production process to be in place.
3. ISO 9000 makes the development process focused, efficient, and costeffective.
4. ISO 9000 certification points out the weak points of an organisation and recommends remedial action.
5. ISO 9000 sets the basic framework for the development of an optimal process and TQM.

Salient Features of ISO 9001 Requirements

- Document control: All documents concerned with the development of a software product should be properly managed, authorised, and controlled. This requires a configuration management system to be in place.
- Planning: Proper plans should be prepared and then progress against these plans should be monitored.
- Review: Important documents across all phases should be independently checked and reviewed for effectiveness and correctness.

- Testing: The product should be tested against specification.
- Organisational aspects: Several organisational aspects should be addressed e.g., management reporting of the quality team.

Shortcomings of ISO 9000 Certification:

1. ISO 9000 requires a software production process to be adhered to, but does not guarantee the process to be of high quality.
2. ISO 9000 certification process is not fool-proof and no international accreditation agency exists. Therefore it is likely that variations in the norms of awarding certificates can exist among the different accreditation agencies and also among the registrars.
3. Organisations getting ISO 9000 certification often tend to downplay domain expertise and the ingenuity of the developers. These organisations start to believe that since a good process is in place, the development results are truly person-independent.
4. In other words, software development is a creative process and individual skills and experience are important.
5. ISO 9000 does not automatically lead to continuous process improvement. In other words, it does not automatically lead to TQM.

ISO 9000-2000

ISO revised the quality standards in the year 2000 to fine tune the standards. The major changes include a mechanism for continuous process improvement. There is also an increased emphasis on the role of the top management, including establishing measurable objectives for various roles and levels of the organisation. The new standard recognizes that there can be many processes in an organisation.\

SEI capability maturity model

- ✓ SEI CMM is suited for large organisations.
- ✓ SEI Capability Maturity Model (SEI CMM) was proposed by Software Engineering Institute of the Carnegie Mellon University, USA. In simple words, CMM is a reference model for apprising the software process maturity into different levels.
- ✓ This can be used to predict the most likely outcome to be expected from the next project that the organisation undertakes.
- ✓ The different levels of SEI CMM have been designed so that it is easy for an organisation to slowly build its quality system starting from scratch.
- ✓ Capability maturity model integration (CMMI) is the successor of the CMM.
- ✓ SEI CMM classifies software development industries into the following five maturity levels:

Level 1: Initial:

1. A software development organisation at this level is characterised by adhoc activities.
2. Very few or no processes are defined and followed.
3. Since software production processes are not defined, different engineers follow their own process and as a result development efforts become chaotic.
4. Therefore, it is also called chaotic level.
5. The success of projects depends on individual efforts and heroics.
6. When a developer leaves the organisation, the successor would have great difficulty in understanding the process that was followed and the work completed.

Level 2: Repeatable

1. At this level, the basic project management practices such as tracking cost and schedule are established.
2. Configuration management tools are used on items identified for configuration control.
3. Size and cost estimation techniques such as function point analysis, COCOMO, etc.,are used.
4. The necessary process discipline is in place to repeat earlier success on projects with similar applications.
5. Though there is a rough understanding among the developers about the process being followed, the process is not documented.
6. For this reason, the successful development of one product by such an organisation does not automatically imply that the next product development will be successful.

Level 3: Defined

1. At this level, the processes for both management and development activities are defined and documented. There is acommon organisation-wide understanding of activities, roles, and responsibilities.
2. The processes though defined, the process and product qualities are not measured.
3. At this level, the organisation builds up the capabilities of its employees through periodic training programs.
4. Also, review techniques are emphasized and documented to achieve phase containment of errors.
5. ISO 9000 aims at achieving this level.

Level 4: Managed:

1. At this level, the focus is on software metrics. Both process and product metrics are collected. Quantitative quality goals are set for the products and at the time of completion of development it was checked whether the quantitative quality goals for the product are met.
2. Various tools like Pareto charts, fishbone diagrams, etc. are used to measure the product and process quality.

Level 5: Optimizing:

1. At this stage, process and product metrics are collected. Process and product measurement data are analyzed for continuous process improvement.
2. At CMM level 5, an organisation would identify the best software engineering practices and innovations (which may be tools, methods, or processes) and would transfer these organisation-wide.

PERSONAL SOFTWARE PROCESS

- ✓ PSP is based on the work of David Humphrey [Hum97].
- ✓ PSP is a scaled down version of industrial software process.
- ✓ PSP is suitable for individual use.
- ✓ PSP recognizes that the process for individual use is different from that necessary for a team.
- ✓ PSP is a framework that helps engineers to measure and improve the way they work.
- ✓ It helps in developing personal skills and methods by estimating, planning, and tracking performance against plans, and provides a defined process which can be tuned by individuals.

Time measurement: PSP advocates that developers should rack the way they spend time. For example, he may stop the clock when attending a telephone call, taking a coffee break, etc. An engineer should measure the time he spends for various development activities such as designing, writing code, testing, etc.

The PSP is schematically shown in Figure. While carrying out the different phases, an individual must record the log data using time measurement. During post-mortem, they can compare the log data with their project plan to achieve better planning in the future projects, to improve his process, etc.