Dr. Ambedkar Memorial Institute of Information Technology & Management Science

LECTURE NOTES

ON

SOFTWARE TESTING AND QUALITY ASSURANCE

MCPE2012 MCA (3rd Sem)

Prepared By

Prof. Tapaswini Kar

UNIT-1

Software Testing

Software testing is a systematic process used to evaluate and verify whether a software application functions as intended and meets its specified requirements. It involves identifying defects or bugs early, ensuring that the product is reliable, secure, high-performing, and user-friendly before release.

The process combines verification (checking that the product is built according to specifications) and validation (ensuring the product fulfils user needs). By applying diverse testing techniques and methodologies, software testing safeguards against future performance issues, improves overall quality, and boosts customer satisfaction.

Objectives of Software Testing

If software testing is like preparing a meal for a special occasion, what are our primary goals or objectives for this preparation? Why do we put in all this effort? Here are the direct objectives of software testing:

- **Ensure Reliability:** Just as you'd want your dish to be consistently delicious every time you prepare it, we test to ensure the software is dependable and won't disappoint you when needed.
- **Verify Functionality:** It's like ensuring all the ingredients in a recipe come together ideally. In software, we verify that every feature functions as intended.
- **Identify Defects:** Before serving a dish, you'd want to ensure it tastes right and has no unwanted elements. Similarly, testing helps spot and fix any issues or bugs in the software. It's essential to be aware of common pitfalls, as highlighted in our article on software testing errors to look out for.
- **Assess User Experience:** Just as the presentation and taste of a dish enhances your dining experience, a user-friendly software interface ensures a delightful digital journey.
- **Maintain Quality Standards:** Just as dishes must meet specific culinary standards, the software has quality benchmarks it needs to meet.

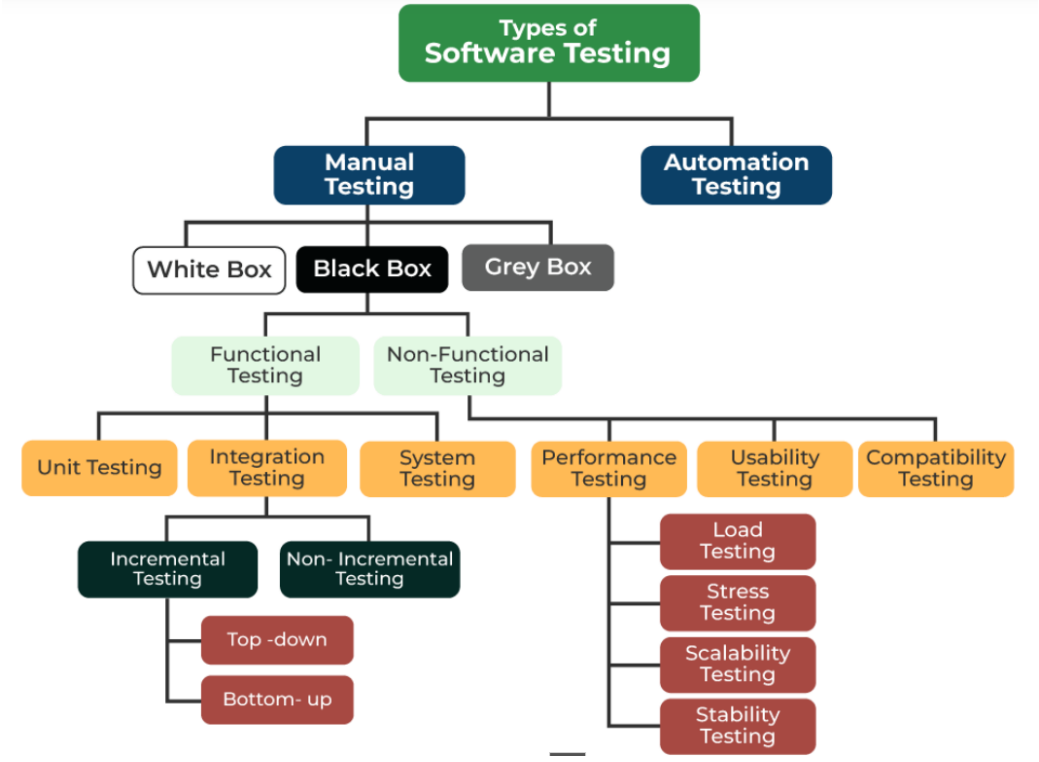
Why Software Testing is Needed

- Detects defects/bugs early.
- Ensures the software meets user requirements.
- Improves reliability, quality, and customer satisfaction.
- Reduces maintenance and failure costs.

Software Testing Best Practices

To ensure a structured and effective testing process, here are some of the best practices in software testing:

- Clearly define the expected output for each test case.
- Developers should avoid testing their own code to eliminate bias.
- Organizations should not test their own software to ensure objectivity.
- Analyze each test result carefully to spot patterns or recurring issues.
- Write test cases that include both valid and invalid inputs.
- Perform negative testing to uncover hidden vulnerabilities.
- Avoid low-value test cases that are costly to execute and offer minimal insights.



Software Testing Process

1. Assess Development Plan and Status

Before creating a Verification, Validation, and Testing (VVT) plan, review the project's development plan for completeness and correctness. This helps testers estimate the resources, tools, and time needed.
Example: Evaluating an e-commerce project's timeline to ensure adequate testing coverage.

2. Develop the Test Plan

Design a comprehensive test plan similar to other project plans, but tailored to the risks associated with the software. The plan outlines scope, strategy, resources, tools, and responsibilities.
Example: Preparing a plan for testing a banking app with a focus on security risks.

3. Test Software Requirements

Verify that requirements are complete, accurate, and non-conflicting. Poor requirements are the leading cause of failures and costly rework.
Example: Ensuring a ride-sharing app specifies clear rules for fare calculation under different conditions.

4. Test Software Design

Evaluate both external and internal design to confirm it aligns with requirements, is efficient, and works with the target hardware or environment.
Example: Reviewing UI wireframes and database schemas for a school management system.

5. Build Phase Testing

During software construction, verify components early to catch defects cheaply. Automated builds may reduce effort, but waterfall-based development requires more intensive checks.
Example: Running static code analysis and unit tests immediately after each build.

6. Execute and Record Results

Perform dynamic testing—run test cases, validate functionality, and record results. Ensure the code meets both requirements and design specifications.
Example: Executing login functionality tests and logging all pass/fail outcomes in JIRA.

7. Acceptance Testing

Allow end-users to evaluate whether the software supports their real-world tasks and expectations—not just documented requirements.
Example: Hospital staff testing a patient management system to confirm it simplifies daily operations.

8. Report Test Results

Communicate findings continuously, both verbally and in writing. Reporting early helps address defects at a lower cost.
Example: Sending a daily defect summary to developers and managers.

9. Software Installation Testing

Verify that the software can be installed and run correctly in the production environment, integrating smoothly with operating systems and related software.
Example: Testing the installation of a POS (Point-of-Sale) system on production hardware

10. Test Software Changes

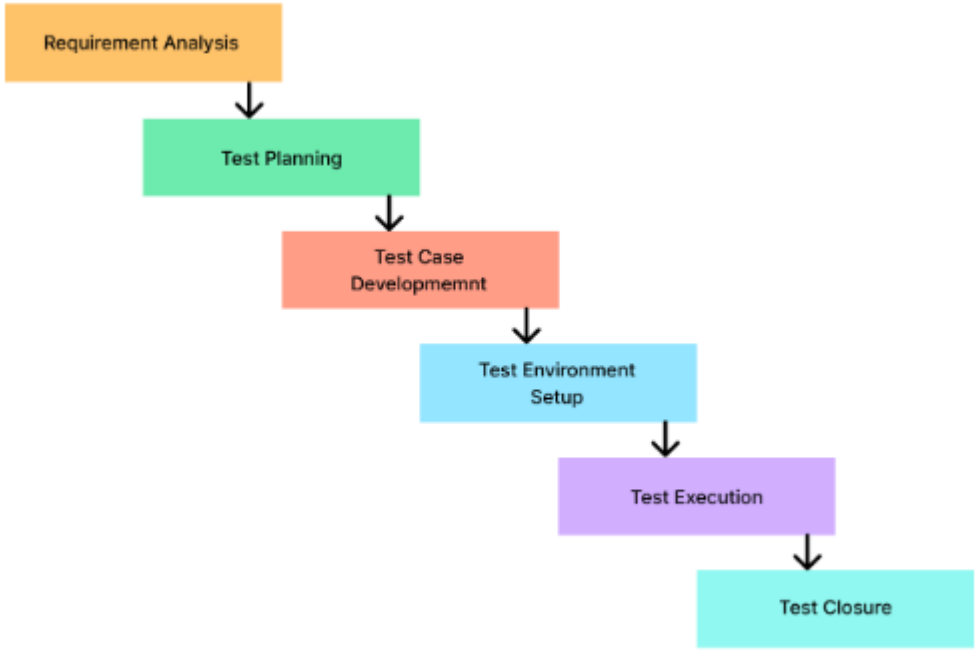
Whenever requirements or code change, update the test plan and evaluate the impact. Re-test to ensure modifications don’t introduce new issues.
Example: Re-running regression tests after adding a new discount feature to an online store.

11. Evaluate Test Effectiveness

At the end of the testing assignment, review how effective the testing was. Involve testers, developers, users, and QA teams to identify improvements.
Example: Holding a retrospective meeting to improve future testing strategies.

Software Testing Life Cycle (STLC)

Software Testing Life Cycle consist of six (generic) phases: 1) Analysis, 2)Planning, 3) Development, 4) Construction, 5) Execution, 6) Test Closure. Each phase in the life cycle is described with the respective activities.



1. Requirement Analysis

Requirement analysis involves identifying, analysing, and documenting the requirements of a software system. The testing team works closely with stakeholders to understand functionality, performance, usability, and business rules.

- To ensure that all requirements are testable and clearly defined.
- To create a **Requirement Traceability Matrix (RTM)** mapping each requirement to test cases.

Example:
Suppose you're testing an **e-commerce website**. In this phase, you clarify:

- Product search must return accurate results within 2 seconds.
- Payment gateway must support credit cards, UPI, and PayPal.
  Ambiguities like "fast loading" are clarified into measurable targets.

2. Test Planning

A phase where the scope, objectives, resources, tools, schedules, and responsibilities for testing are defined. Risks are assessed, and a Master Test Plan is created.

- To minimize risks and ensure all testing activities are organized.
- To determine timelines and responsibilities for each team member.

Example:
For the e-commerce site, the team decides:

- Manual testing for UI.
- Selenium for automation.
- LoadRunner for performance tests.
- Assigns a 3-week testing window before the release.

3. Test Case Development / Design
Creating detailed test cases, scripts, and test data based on requirements. Test cases are reviewed for coverage and accuracy. Test automation can also be initiated here.

Steps:

- **Test Design:** Identify test scenarios.
- **Test Case Creation:** Include input data, steps, and expected output.
- **Test Case Review:** Peer-review test cases for completeness.

Example:

  - Input: Select a product and click "Add to Cart."
  - Expected Output: Cart count increases by 1, and product appears in the cart list.

4. Test Environment Setup
Setting up hardware, software, networks, and databases to simulate a production environment.

- To replicate real-world conditions for accurate results.
- To enable functional, performance, and load testing.

Example:
For the e-commerce platform:

- Use the same database version and server capacity as production.
- Configure payment gateway sandbox for transaction testing.
- Automate environment setup using scripts for consistency

5. Test Execution
Executing test cases on the application to verify functionality, performance, and compliance with requirements. Includes logging and retesting defects.

Activities:

- Run manual and automated test cases.
- Log defects in a tracking system (e.g., JIRA).
- Perform regression testing after fixes.

Example:

- Run the "Add to Cart" test case: If adding a product fails, log a defect.
- After developers fix the issue, retest and check if other related features (like Checkout) still work.

6. Defect Tracking and Reporting
Managing and tracking bugs found during test execution.

Activities:

- Record defects with severity, priority, and reproduction steps.
- Communicate with developers for resolution.
- Generate reports showing test coverage, pass/fail rates, and defect density.

Example:
A bug report for "Payment fails with valid card" includes steps to reproduce, screenshots, severity level (critical), and environment details.

7. Test Closure
The final phase where all testing activities are evaluated, and a closure report is prepared.

Activities:

- Review and analyse test results.

- Assess test coverage and exit criteria.
- Document lessons learned and metrics.
- Archive test artifacts for future use.

Example:

- A Test Closure Report summarizes pass/fail statistics, defect counts, and overall product quality.
- Team documents improvements for the next release cycle.

Error

Error is a situation that happens when the Development team or the developer fails to understand a requirement definition and hence that misunderstanding gets translated into buggy code. This situation is referred to as an Error and is mainly a term coined by the developers.

- Errors are generated due to wrong logic, syntax, or loop that can impact the end-user experience.
- It is calculated by differentiating between the expected results and the actual results.
- It raises due to several reasons like design issues, coding issues, or system specification issues and leads to issues in the application.

Below are the **main categories of errors**:

1. Functional Errors: These occur when the software does not perform its intended functions according to the requirements or specifications.

Examples: A button not working, incorrect data processing, features not implemented correctly.

2. Logic Errors: Flaws in the program's algorithm or reasoning that lead to incorrect or unexpected outputs, even if the syntax is correct.

Examples: Incorrect calculations, wrong conditional statements, infinite loops.

3. Syntax Errors: Violations of the rules of the programming language, like grammatical mistakes in code. These are often caught by compilers or interpreters.

Examples: Missing semicolons, incorrect keyword usage, unbalanced parentheses.

4. Runtime Errors: Errors that occur during the execution of a program, often due to unexpected conditions or inputs that the program cannot handle.

Examples: Division by zero, out-of-memory errors, attempting to access a non-existent file.

5. Performance Errors: Issues where the software fails to meet expected performance criteria, such as speed, responsiveness, or resource usage.

Examples: Slow response times, excessive memory consumption, system crashes under heavy load.

6. Usability Errors: Problems related to the user interface and user experience, making the software difficult to use or understand.

Examples: Confusing navigation, unclear error messages, inconsistent design elements.

7. Security Errors (Vulnerabilities): Weaknesses in the software that can be exploited by malicious actors, leading to data breaches, unauthorized access, or system compromise.

Examples: SQL injection vulnerabilities, cross-site scripting (XSS), weak authentication mechanisms.

8. Compatibility Errors: Problems arising when the software does not function correctly across different environments, such as operating systems, browsers, or hardware configurations.

Examples: Layout issues in different browsers, features not working on specific operating system versions.

9. Configuration Errors: Issues caused by incorrect or incompatible settings in the software's configuration, or environmental setup.

Examples: Incorrect database connection strings, misconfigured server settings.

10. Boundary Condition Errors: Failures that occur when the software handles extreme or edge cases of input data incorrectly.

Examples: Errors when processing maximum or minimum allowed values, handling empty inputs.

Stubs and Drivers

- Stubs and Drivers are dummy codes used during software development and testing.
- They help test a module even when some parts of the system are not yet implemented.
- Commonly used in:
    - Top-Down Integration Testing → Stubs
    - Bottom-Up Integration Testing → Drivers
- Also used in porting, distributed computing, and general software development.

Why They Are Needed

During development, you may need to **test a specific module** before the rest of the system is complete.

- If a **called module** is missing → Use a **Stub**.

- If a **caller module** is missing → Use a **Driver**.
  This ensures testing can proceed **without waiting** for other components to be finished.

What Are Dummy Codes?

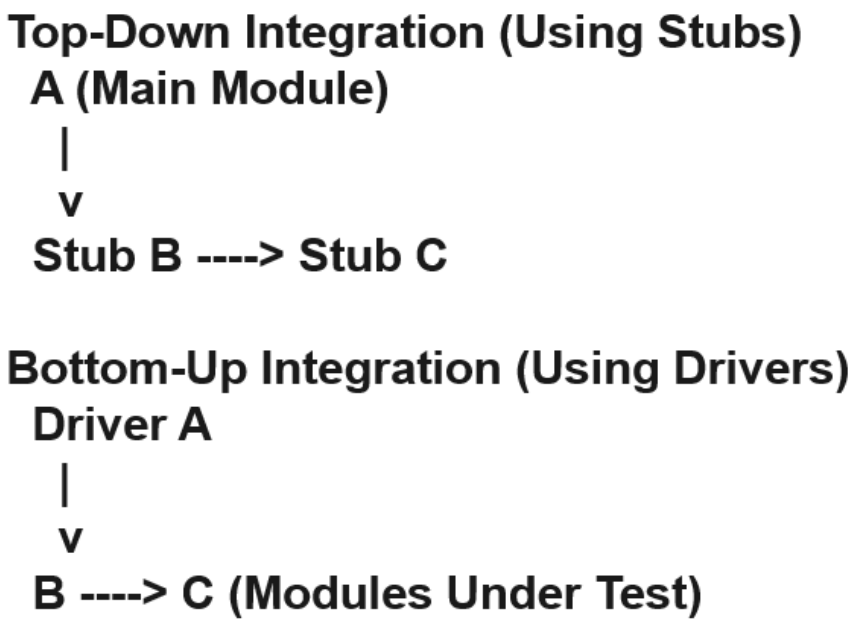
- Dummy codes are placeholder routines that simulate the behavior of the actual, yet-to-be-developed modules.
- They don't perform real processing—they only provide enough functionality for compilation, linking, and testing.

1. Stub (Top-Down Testing)

- A Stub is a dummy module that replaces lower-level components.
- Used when higher-level modules are ready but lower-level modules are not.
- **Example:**
  - Codes: A → B → C
  - A is ready, but B and C are not.
  - Use Stubs for B and C to simulate their expected outputs so A can be tested.

2. Driver (Bottom-Up Testing)

- A Driver is a dummy module that replaces higher-level components.
- Used when lower-level modules are ready but the caller modules are not.
- **Example:**
  - Codes: **A → B → C**
  - **B** and **C** are ready, but **A** is not.
  - Use a **Driver** to simulate **A's input** to test **B** and **C**.



verification and validation

Verification and Validation are quality assurance processes used to ensure that a software system meets specifications, follows standards, and fulfills its intended purpose. Both are essential for delivering reliable and high-quality software.

Verification – "Are we building the product right?"

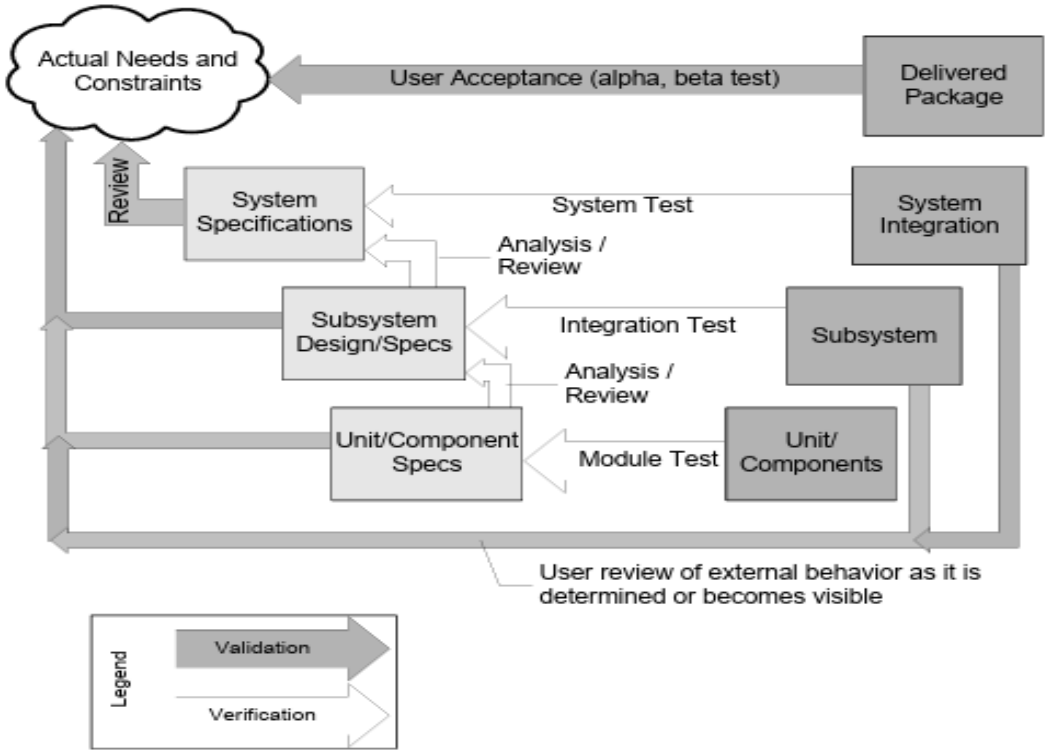
The process of checking whether the software meets specified requirements, standards, and design before execution.

- Nature: Static Testing (no code execution).
- Focus: Examines documents, design, and specifications.
- Goal: Detect defects early in the development phase.
- Techniques: Reviews, Walkthroughs, Inspections, Static analysis
- **Example (Mobile Banking App):**
  Reviewing requirements and design documents to ensure features like fund transfer, balance check, and transaction history are correctly specified and aligned with user requirements.

Validation – "Are we building the right product?"

The process of evaluating the final software product to ensure it meets business and user requirements in real-world conditions.

- Nature: Dynamic Testing (involves code execution).
- Focus: Tests the actual product and its functionality.
- Goal: Ensure the product fulfills its intended use.
- Techniques: Functional testing, System testing, User Acceptance Testing (UAT), Beta testing
- **Example (Mobile Banking App):**
  Testing the app on real devices—checking whether users can log in, transfer money, view transaction history, and providing it to real users for feedback on usability and performance.



Left side of the "V" = Verification → Reviewing and checking documents, designs, and specifications before coding.

Right side of the "V" = Validation → Testing the actual product after coding to ensure it meets user needs.

Bottom of the "V" = Implementation → Where coding happens.

Elements in the Diagram

1. **Actual Needs and Constraints**
   - Represents the real-world requirements from the user or business.
   - Drives the System Specifications phase.
2. **System Specifications (Verification)**
   - Defines what the system should do.
   - Verified through reviews and system tests later.
3. **Subsystem Design/Specs (Verification)**
   - Breaks the system into subsystems/modules.
   - Verified through integration tests later.
4. **Unit/Component Specs (Verification)**
   - Specifies small components or units.

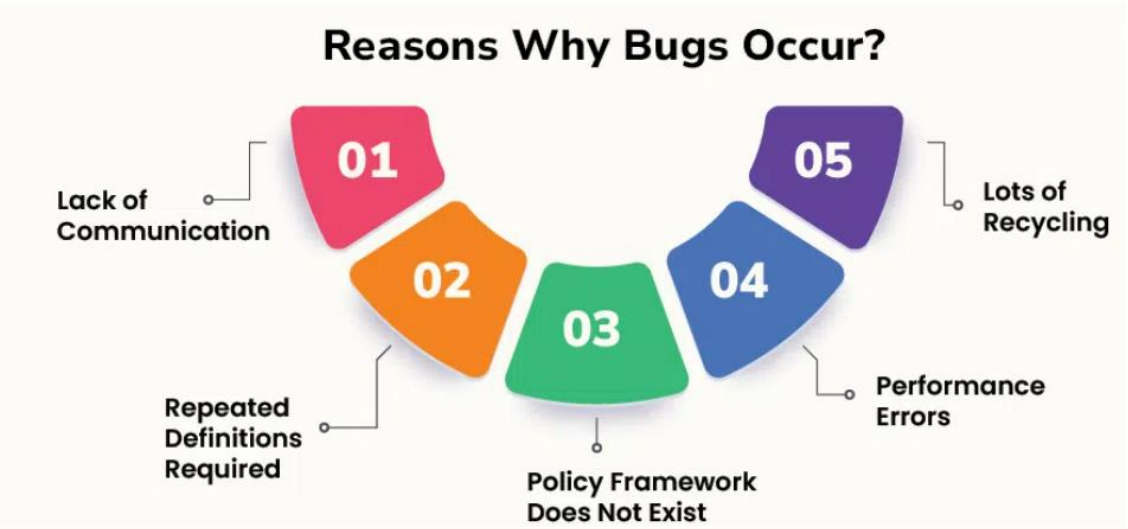
- Verified through module tests later.

5. **Unit/Components (Coding)**
   - Actual development of individual components.
   - Module testing validates each unit.
6. **Subsystem & System Integration (Validation)**
   - Subsystem Integration: Combines components → Integration testing checks their interaction.
   - System Integration: Combines subsystems → System testing checks overall functionality.
7. **User Acceptance (Validation)**
   - End users test the final product (alpha/beta testing) to ensure it meets actual needs.

BUG

A software bug is an error, flaw, failure, or fault in a computer program or system that causes it to produce an inaccurate or unexpected output or to behave in unintended ways. In software testing, a bug is identified when the actual behavior of the software deviates from the expected behavior, as defined by the software's requirements.

A malfunction in the software/system is an error that may cause components or the system to fail to perform its required functions. In other words, an error encountered during the test can cause malfunction. For example, incorrect data description, statements, input data, design, etc.



Reasons for Occurrence of Bugs (Short Version)

1. **Human Error** – Mistakes in logic, syntax, or linking due to human involvement.

2. **Communication Failure –** Incorrect or missing information between SDLC phases or developers.
3. **Misunderstood Requirements/Design –** Poorly communicated requirements or wrong design specs passed to developers.
4. **Unrealistic Timeframes –** Tight schedules or insufficient resources causing rushed work and inadequate testing.
5. **Poor Design Logic –** Inadequate R&D or wrong choice of components, tools, or techniques.
6. **Poor Coding Practices –** Inefficient code, typos, wrong validation, or faulty tools causing hidden errors.
7. **Lack of Skilled Testing –** Inadequate testing lets bugs go unnoticed.

Bug Tracking Tools (Summary)

Katalon TestOps

- Free orchestration platform for bug tracking.
- Works on Cloud/Desktop (Windows/Linux).
- Supports frameworks (JUnit, Pytest, Mocha), CI/CD tools (Jenkins, CircleCI), and Jira/Slack.
- Real-time data tracking, performance reports, Smart Scheduling, dashboards, and KPI tracking.

Kualitee

- Combines bug tracking, test cases, and workflow management in one tool.
- Features: Create/assign/track bugs, tracing between defects and tests, custom permissions, interactive dashboards, REST API integration, and intuitive UI.

QA Coverage

- Comprehensive test management and defect tracking.
- Customizable error-tracking process from detection to closure.
- Tracks risks, enhancements, suggestions, and includes needs management, test case design, execution, and reporting.

Role of a Software Tester

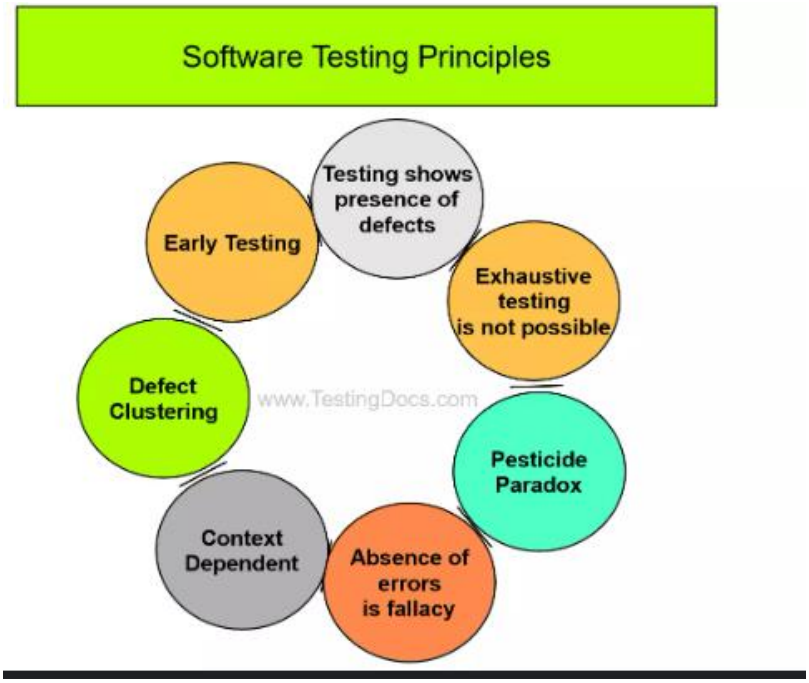
A software tester plays a **key role** in the Software Testing Life Cycle (STLC). Their main job is to **find bugs early**, ensure fixes do not break other features, and confirm the software meets user needs. While **developers prove** that the software works, **testers find weaknesses** by running different test cases and configurations.

Responsibilities of a Software Tester

1. **Understand the Product/Application** – Analyze specifications and requirements to know what needs to be tested.
2. **Create a Test Strategy** – Write test plans, prioritize testing based on risks, and prepare test data.
3. **Set Up Test Environments** – Prepare the environment for both manual and automated testing.
4. **Execute Test Cases** – Run tests (functional, performance, security, etc.) to check software behavior.
5. **Report Bugs Clearly** – Provide detailed defect reports with steps to reproduce and screenshots.
6. **Collaborate with Teams** – Work closely with developers, business analysts, and project managers for smooth communication.
7. **Re-Test and Perform Regression Testing** – Ensure fixed bugs don't affect other parts of the software.
8. **Maintain Test Documentation** – Keep records of test plans, cases, and results for future reference.
9. **Suggest Improvements** – Provide feedback on design or usability to enhance product quality.
10. **Stay Updated** – Learn new testing tools and techniques to improve testing efficiency.

Software testing axioms

Software testing axioms are fundamental principles, or "self-evident truths," that guide the testing process, such as the impossibility of exhaustive testing, that testing finds defects but cannot prove their absence, and that testing is a risk-based activity. These principles, often known as the Seven Principles of Software Testing, help testers understand the nature and goals of testing, ensuring more effective and efficient test strategies.



1. Testing Shows Presence of Defects

- Testing can show the existence of defects in the software, but it cannot prove that there are no defects.
- Even if no bugs are found, it does not mean the software is error-free.

2. Exhaustive Testing is Not Possible

- It is impractical to test all possible input combinations and scenarios due to limited time and resources.
- Testing should focus on critical areas and high-risk functionalities instead of trying to cover everything.

3. Early Testing

- Testing should begin as early as possible in the software development lifecycle (SDLC), preferably during requirements and design phases.
- Early defect detection reduces cost and effort for fixing issues later.

4. Defect Clustering

- This principle emphasizes focusing more testing effort on modules with a history of defects or complex functionality.

5. Pesticide Paradox

- Repeating the same tests over time will eventually fail to find new defects.
- Test cases should be regularly reviewed and updated to catch new bugs.

6. Context-Dependent Testing

- Testing strategies, techniques, and priorities depend on the type of software, business domain, and project requirements.
- For example, safety-critical software like aviation systems requires more rigorous testing than a simple web application.

7. Absence of Errors is Fallacy

- Finding and fixing defects does not guarantee the system is ready for use.
- The software may still fail to meet user expectations or business requirements.

Testing of component-based software system

Testing of component-based software systems involves a multi-faceted approach to ensure the quality and functionality of individual components and their interactions within the larger system. This process typically includes:

1. Component Testing (Module Testing):

Verifying the behavior, usability, and functionality of individual software components in isolation.

To ensure each component meets its specified requirements and functions correctly before integration.

Typically performed by developers or dedicated testers using test harnesses or drivers to simulate the component's environment and provide inputs.

Early detection and resolution of defects, reducing the complexity of debugging later in the development cycle.

2. Integration Testing:

Testing the interactions and interfaces between integrated components.

To identify issues that arise when multiple components work together, ensuring data flow and communication are correct.

Can be performed using various strategies like top-down, bottom-up, or sandwich approaches, gradually adding and testing components until a complete subsystem or system is formed.

3. System Testing:

Evaluating the complete, integrated system to verify it meets all specified requirements from an end-to-end perspective.

To validate the system's overall functionality, performance, security, and other non-functional attributes.

4. Other Relevant Testing Types:

Regression Testing: Re-running previously executed tests to ensure that new changes or integrations have not introduced new defects or negatively impacted existing functionality.

Performance Testing: Assessing the system's responsiveness, stability, and resource usage under various load conditions.

Security Testing: Identifying vulnerabilities and weaknesses in the system's security mechanisms.

Code Inspection:

Code inspection is one of the most formal evaluation techniques in software testing. It is a static testing method aimed at identifying defects early in the development process to prevent flaw amplification later. The primary purpose of code inspection is to detect defects in code, design, or documentation. Additionally, it may highlight process improvements that can enhance overall software quality.

Key Features of Code Inspection:

- **Formal Process:** Inspections follow a structured approach with clearly defined roles such as moderator, author, reviewer, and recorder.
- **Documentation:** Findings are recorded in a detailed inspection report, including metrics and observations for process improvement.
- **Error Correction:** Inspections can help correct errors in the document or code under review.
- **Preparation:** Thorough preparation is essential; reviewers must study the source materials to ensure consistency and completeness.
- **Benefits:** Improves software dependability, maintainability, and accessibility. It reduces the cost of fixing errors compared to later stages of development.

How does Code Inspection work?

Participants in inspection activities follow a predetermined process and have clear duties to play.

- **Planning —** The moderator plans the inspection. The inspection team is given relevant materials, and after that, the team schedules the inspection meeting and works together.
- **Overview meeting —** The author gives the inspection team members a brief summary of the project and its code if they are unfamiliar with it.
- **Preparation —** After that, each inspection team conducts a code inspection using a few inspection checklists.
- **Inspection meeting —** During this meeting, the reader goes through the work output, section by section, and inspectors point out any flaws.
- **Rework —** The author modifies the work output following the inspection meeting’s action plans.

- **Follow-up —** Hold a meeting with the team after the code inspection to discuss the reviewed code.

Advantages of Code Inspection

- Enhances the overall product quality.
- Finds the flaws/bugs in the program code.
- In any event, it shows a process improvement.
- Rapidly and effectively locates and removes faults.
- Aids in learning from past failures.

Disadvantages of Code Inspection

- It takes more time and planning.
- The procedure is slower.

Code Walkthrough:

A walkthrough is an informal, yet structured review technique. In a walkthrough, the author explains and presents their work product—such as code, design documents, or test cases—to peers or a supervisor to gather feedback. Unlike inspections, walkthroughs are less formal but still purposeful and effective in improving quality.

Key Features of Code Walkthrough:

- **Informal Peer Review:** Can be done individually or in a group setting.
- **Focus on Feasibility:** Evaluates the correctness and practicality of the proposed solution.
- **Static Quality Check:** Detects potential issues before implementation, which is less expensive than fixing problems during execution.
- **Interactive Discussion:** Encourages questions, suggestions, and collaborative problem-solving.

Guidelines for performing code Walkthrough

- The code walkthrough team should be a manageable size. It should ideally be made up of three to seven individuals.
- The discussion does need to center on finding flaws rather than how to correct them.
- Managers should refrain from attending code Walkthrough sessions to promote teamwork and avoid the perception among engineers that they are being evaluated.

Advantages of Code Walkthrough

- Walkthrough offers a variety of viewpoints.
- The primary goal of the code walkthrough is to equip team members with information about the substance of the document under review.
- It raises awareness of the document's content while also revealing problems.

Disadvantages of Code Walkthrough

- Completeness is confined to the areas where the team raises doubts.
- The code walkthrough lacks diversity, with the author driving the process and others validating that what has been said matches what has been done.

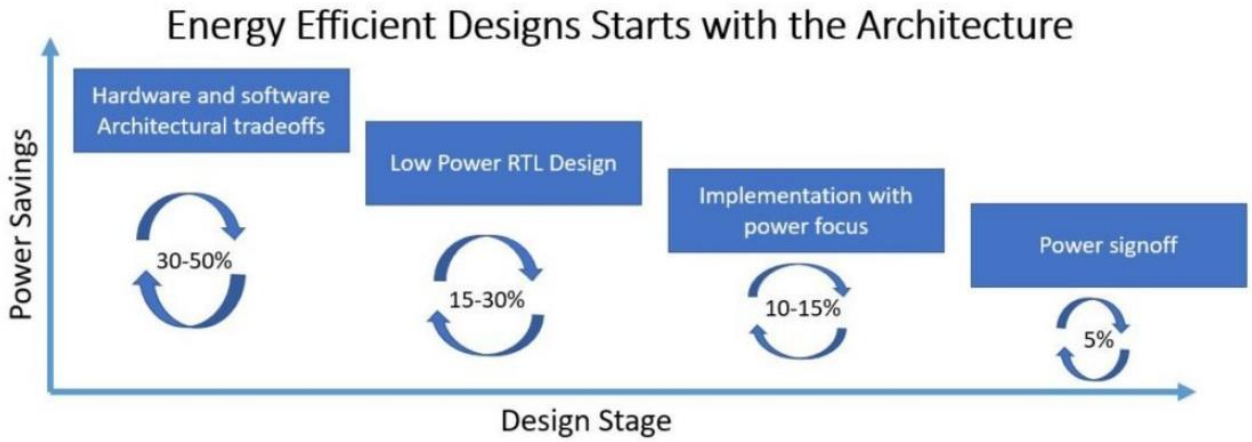
Energy Efficiency in Software Testing

Energy efficiency software testing refers to the process of evaluating and optimizing the energy consumption and performance of software applications or systems. The goal is to ensure that software operates effectively while minimizing power usage, battery drain, and resource waste. As technology continues to evolve and digital systems become integral to our daily lives, the energy footprint of software has emerged as a critical concern. Inefficient software not only affects device performance but also contributes to increased energy costs and environmental impact.

Importance of Energy-Efficient Software Testing

The growing dependency on software across industries—ranging from mobile applications to large-scale enterprise systems—has led to a significant rise in energy consumption. Data centers, cloud services, and mobile devices collectively account for a large portion of global energy use. Consequently, software engineers and testers are now focusing on designing, developing, and testing software that is both functionally effective and energy efficient.

Recognizing this, our conference introduced a dedicated track on energy-efficient software testing to promote sustainability in the software industry. This initiative reflects our ongoing commitment to environmental responsibility and technological innovation.



Key Principles of Energy-Efficient Design

1. **Efficient Resource Utilization**
    - Use only the necessary CPU, memory, and storage.
    - Avoid overloading the system with unused services or background processes. *Example:* A mobile app should stop using GPS when the user closes the map.
2. **Modular Design**
    - Break the system into smaller, independent modules.
    - Each module runs only when needed, saving energy. *Example:* In a music app, the "download manager" runs only when the user downloads a song, not always.
3. **Algorithm Optimization**
    - Choose or design algorithms that complete tasks faster with fewer resources. *Example:* Using an efficient sorting algorithm like QuickSort instead of Bubble Sort reduces CPU cycles.
4. **Hardware Awareness**
    - Design software to use hardware features smartly (like low-power modes). *Example:* A mobile game can automatically lower frame rate when the battery is low.
5. **Energy-Aware Scheduling**
    - Schedule heavy tasks (like updates or backups) during idle times or when the system is charging.
      ◈ *Example:* Cloud apps perform data synchronization when servers are under low load.
6. **Data Efficiency**
    - Minimize unnecessary data transfers or network usage since they consume power. *Example:* A chat app should compress images before sending them.
7. **Use of Cloud and Virtualization**
    - Cloud services allow flexible resource allocation — using only what's needed and turning off idle servers. *Example:* Amazon and Google use auto-scaling to power down unused servers, saving energy.

UNIT_2

Software Testing

Software testing is an essential phase of the Software Development Lifecycle (SDLC) that focuses on evaluating, verifying, and validating a software product or application.

Its primary goal is to ensure the software functions as expected, meets technical and design specifications, and fulfills user requirements effectively and efficiently.

The process involves identifying and resolving errors or bugs that could impact performance, security, or usability in the future. By confirming that the software operates correctly, securely, and reliably under various conditions, testing helps deliver a high-quality product that meets both business objectives and user expectations.

Software Testing Methods

Software Testing Methods are the different approaches or techniques used to evaluate and verify a software product or application. They define *how* testing is carried out to ensure the software is free from defects, meets requirements, and performs efficiently.

These methods can be broadly categorized into functional and non-functional testing, and further broken down by approach (manual vs. automated) or knowledge of the internal system (black-box, white-box, gray-box).

1. Functional Testing: Verifies that the software's features and operations work according to the specified requirements.

Ensures each function of the software behaves as expected with valid and invalid inputs.

- **Techniques/Types:**
    - Unit Testing
    - Integration Testing
    - System Testing
    - Regression Testing
    - Smoke and Sanity Testing
- **Example:** Checking whether a login form correctly authenticates users with valid credentials and rejects invalid ones.

2. Non-Functional Testing: Evaluates the non-functional aspects of the software—such as performance, security, usability, and reliability—rather than specific behaviors or functions.

Ensures the application performs well under various conditions and meets quality standards.

- **Techniques/Types:**
  - Performance Testing (e.g., Load, Stress, Volume)
  - Security Testing
  - Usability Testing
  - Compatibility Testing
  - Reliability and Maintainability Testing
- **Example:** Measuring how quickly a web application loads when 1,000 users access it simultaneously.

3. Based on Execution

- **Manual Testing:** Test cases are executed manually without using tools or scripts. Best for small or ad-hoc testing.
- **Automation Testing:** Uses tools or scripts to run tests automatically. Suitable for repetitiveorlarge-scaletesting.
  Tools: Selenium, JUnit, TestNG.

4. Based on Knowledge of Code

- **Black-BoxTesting:**
  Tests the software without knowing its internal code or logic. Focuses on inputs and outputs.
  Example: Checking if the login feature works with valid and invalid credentials.
- **White-BoxTesting:**
  Tests the internal logic, structure, and code paths of the software.
  Example: Verifying loops, conditions, and branches in the code.
- **Gray-BoxTesting:**
  Combines both black-box and white-box techniques, using partial knowledge of the internal structure.

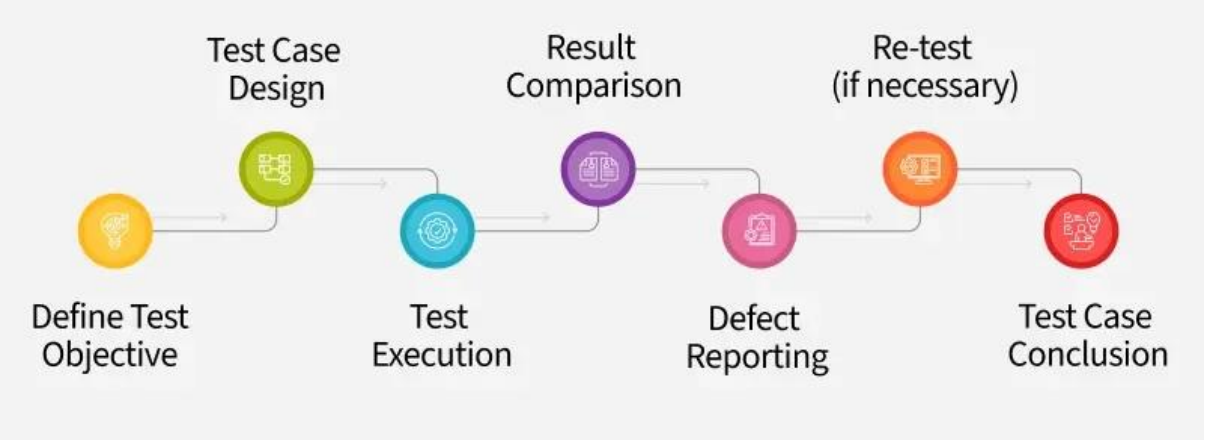
Software Testing Fundamentals

Definition of Software Testing

1. Software testing is a method of assessing the functionality of a software program.
2. There are many different types of software testing but the two main categories are dynamic testing and static testing.

Test case design

Writing effective test cases is a critical skill in software testing, ensuring that the software meets the desired functionality and quality standards. A well-written test case provides clear instructions to verify specific features of the software and helps in identifying potential defects easily.



(Flow of Test Case)

Test Case Template

A Test Case Template is a simple, organized format used in software testing to create test cases. It helps ensure that all tests are written clearly and consistently.

Let's look at a basic test case template for the login functionality.

- The Test case template contains the header section which has a set of parameters that provide information about the test case such as the tester's name, test case description, Prerequisite, etc.
- The body section contains the actual test case content, such as test ID, test steps, test input, expected result, etc.

Below is the table that shows the basic template of a test case:

Fields	Description
Test Case ID	Each test case should have a unique ID.
Test Case Description	Each test case should have a proper description to let testers know what the test case is about.
Pre-Conditions	Conditions that are required to be satisfied before executing the test case.
Test Steps	Mention all test steps in detail and to be executed from the end-user's perspective.
Test Data	Test data could be used as input for the test cases.
Expected Result	The result is expected after executing the test cases.
Post Condition	Conditions need to be fulfilled when the test cases are successfully executed.
Actual Result	The result that which system shows once the test case is executed.
Status	Set the status as Pass or Fail on the expected result against the actual result.
Project Name	Name of the project to which the test case belongs.
Module Name	Name of the module to which the test case belongs.
Reference Document	Mention the path of the reference document.

Fields	Description
Created By	Name of the tester who created the test cases.
Date of Creation	Date of creation of test cases.
Reviewed By	Name of the tester who reviews the test case.
Date of Review	When the test cases were reviewed.
Executed By	Name of the tester who executed the test case.
Date of Execution	Date when the test cases were executed.
Comments	Include comments which help the team to understand the test cases

Parameters of a Test Case

Here are the important parameters of the testcase which is helping to the development process of software:

- Module Name: Describes the functionality being tested.
- Test Case ID: A unique identifier for each test case.
- Tester Name: The name of the person conducting the test.
- Test Scenario: A brief description of what the test will cover.
- Test Case Description: Specifies the condition or feature being tested.
- Test Steps: A detailed list of actions to execute the test.
- Prerequisite: Any conditions that must be met before starting the test.
- Test Priority: Indicates the order of importance for testing.
- Test Data: Inputs required for the test.
- Test Expected Result: The expected outcome of the test.
- Actual Result: The actual outcome after test execution.

- Environment Information: Details of the test environment, like OS and software version.
- Status: Whether the test passed, failed, or is not applicable.
- Comments: Any additional notes about the test.

Types Of Test Cases

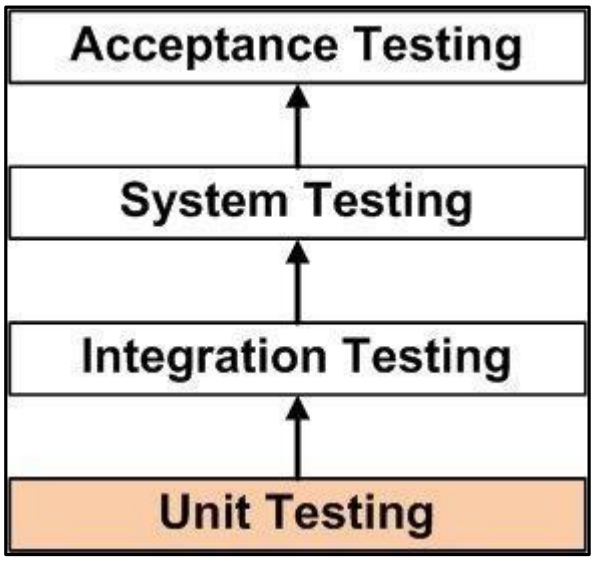
- Formal Test Cases: Formal test cases are test cases that follow the basic test case format. It contains the test case parameters such as conditions, ID, Module name, etc. Formal Test cases have set input data and expected results, they are performed as per the given order of steps.
- Informal Test Cases: Informal test cases are test cases that don't follow the basic test case format. In these, as the tests are performed the test cases are written in real-time then pre-writing them, and the input and expected results are not predefined as well.

Testing Strategies

Unit Testing

Unit Testing is a level of the software testing process where individual units/components of a software/system are tested.

The purpose is to validate that each unit of the software performs as designed.



i. A unit is the smallest testable part of software.

ii. It usually has one or a few inputs and usually a single output.

iii. In procedural programming a unit may be an individual program, function, procedure, etc.

iv. In object-oriented programming, the smallest unit is a method, which may belong to a base/super class, abstract class or derived/child class.

Advantages

i. Unit testing increases confidence in changing/maintaining code.

ii. If good unit tests are written and if they are run every time any code is changed, the likelihood of any defects due to the change being promptly caught is very high.

iii. If unit testing is not in place, the most one can do is hope for the best and wait till the test results at higher levels of testing are out.

iv. If codes are already made less interdependent to make unit testing possible, the unintended impact of changes to any code is less.

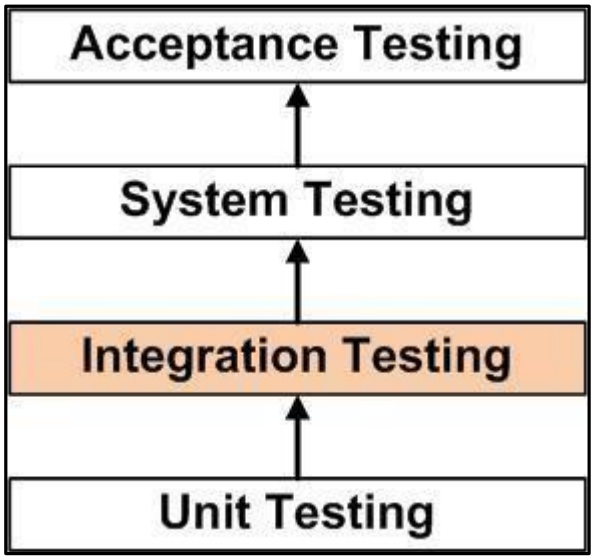
v. Codes are more reusable. In order to make unit testing possible, codes need to be modular. This means that codes are easier to reuse.

vi. The cost of fixing a defect detected during unit testing is lesser in comparison to that of defects detected at higher levels.

vii. Compare the cost (time, effort, destruction, humiliation) of a defect detected during acceptance testing or say when the software is live.

viii. Debugging is easy. When a test fails, only the latest changes need to be debugged. With testing at higher levels, changes made over the span of several days/weeks/months need to be debugged.

Integration Testing



i. **Integration Testing** is a level of the software testing process where individual units are combined and tested as a group. The purpose of this level of testing is to expose faults in the interaction between integrated units.

ii. Test drivers and test stubs are used to assist in Integration Testing.

Alpha and Beta Testing

Alpha Testing

i. Alpha testing is one of the most common software testing strategy used in software development. It's specially used by product development organizations.

ii. This test takes place at the developer's site. Developers observe the users and note problems.

iii. Alpha testing is testing of an application when development is about to complete. Minor design changes can still be made as a result of alpha testing.

iv. Alpha testing is typically performed by a group that is independent of the design team, but still within the company, e.g. in-house software test engineers, or software QA engineers.

v. Alpha testing is final testing before the software is released to the general public. It has two phases:

vi. In the **first phase** of alpha testing, the software is tested by in-house developers. They use either debugger software, or hardware-assisted debuggers. The goal is to catch bugs quickly.

vii. In the **second phase** of alpha testing, the software is handed over to the software QA staff, for additional testing in an environment that is similar to the intended use.

viii. Alpha testing is simulated or actual operational testing by potential users/customers or an independent test team at the developers' site.

ix. Alpha testing is often employed for off-the-shelf software as a form of internal acceptance testing, before the software goes to beta testing.

Beta testing

i. It is also known as field testing.

ii. It takes place at customer's site.

iii. It sends the system to users who install it and use it under real-world working conditions.

iv. A beta test is the second phase of software testing in which a sampling of the intended audience tries the product out.

v. Beta testing can be considered "pre-release testing.

vi. The goal of beta testing is to place your application in the hands of real users outside of your own engineering team to discover any flaws or issues from the

user's perspective that you would not want to have in your final, released version of the application.

vii. Closed beta versions are released to a select group of individuals for a user test and are invitation only, while

viii. Open betas are from a larger group to the general public and anyone interested.

ix. The testers report any bugs that they find, and sometimes suggest additional features they think should be available in the final version

System Testing

Concept of System Testing

i. In system testing the behavior of whole system/product is tested as defined by the scope of the development project or product.

ii. It may include tests based on risks and/or requirement specifications, business process, use cases, or other high level descriptions of system behavior, interactions with the operating systems, and system resources.

iii. System testing is most often the final test to verify that the system to be delivered meets the specification and its purpose.

iv. System testing is carried out by specialist's testers or independent testers.

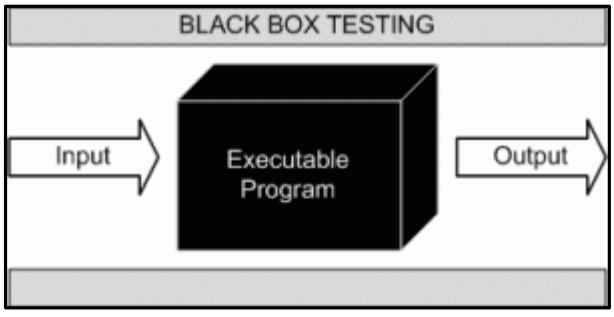
v. System testing should investigate both functional and non-functional requirements of the testing.

Concept of White-box and Black-Box Testing

Black Box Testing

i. Black Box Testing, also known as Behavioral Testing, is a software testing method in which the internal structure/design/implementation of the item being tested is not known to the tester.

ii. These tests can be functional or non-functional, though usually functional.



Black Box Testing

iii. This method is named so because the software program, in the eyes of the tester, is like a black box; inside which one cannot see.

iv. Black Box Testing is contrasted with White Box Testing. View Differences between Black Box Testing and White Box Testing.

v. This method of attempts to find errors in the following categories:

- Incorrect or missing functions
- Interface errors
- Errors in data structures or external database access
- Behavior or performance errors
- Initialization and termination errors

Black Box Testing Techniques

Equivalence partitioning

a) Equivalence Partitioning is a software test design technique that involves dividing input values into valid and invalid partitions and selecting representative values from each partition as test data.

Boundary Value Analysis

b) Boundary Value Analysis is a software test design technique that involves determination of boundaries for input values and selecting values that are at the boundaries and just inside/outside of the boundaries as test data.

Cause Effect Graphing

c) Cause Effect Graphing is a software test design technique that involves identifying the cases (input conditions) and effects (output conditions), producing a Cause-Effect Graph, and generating test cases accordingly.

BLACK BOX TESTING ADVANTAGES

i. Tests are done from a user's point of view and will help in exposing discrepancies in the specifications

ii. Tester need not know programming languages or how the software has been implemented

iii. Tests can be conducted by a body independent from the developers, allowing for an objective perspective and the avoidance of developer-bias

iv. Test cases can be designed as soon as the specifications are complete

BLACK BOX TESTING DISADVANTAGES

i. Only a small number of possible inputs can be tested and many program paths will be left untested

ii. Without clear specifications, which is the situation in many projects, test cases will be difficult to design

iii. Tests can be redundant if the software designer/ developer has already run a test case.

iv. Ever wondered why a soothsayer closes the eyes when foretelling events? So is almost the case in Black Box Testing.

Types of Black Box Testing

Black box testing covers various approaches, each designed to validate different aspects of software functionality. These testing types can be used individually or combined to create comprehensive test coverage for applications.

Here's a look at the different types of black box testing:

1. **Functional testing**

Functional testing evaluates the features and functions of the software, making sure they perform as per requirements.

For an e-commerce website, this can involve checking if products can be easily searched, added to the cart, and purchased successfully. It can focus on the most crucial aspects of the software, integration between key components or the system as a whole.

2. **Non-functional testing**

One of the important types of black box testing, it goes beyond functional testing to evaluate aspects such as performance, usability, and security.

Rather than asking "Does it work?" non-functional testing

asks "How well does it work?" These tests measure quality attributes that impact user satisfaction and system reliability.

3. **Regression testing**

This kind of testing ensures that new code changes don't negatively impact your existing functionality. After modifications or updates to the software, regression test suites verify that previously working features still function correctly.

Many teams automate these tests to run whenever code changes are made, catching unexpected side effects before they reach production.

4. **User acceptance testing (UAT)**

UAT involves actual end users checking that the software meets their needs and expectations. It typically occurs in the final stages before release when stakeholders and customer representatives test the application in scenarios that mimic real-world usage.

Their feedback helps determine if the software is ready for deployment or needs further refinement.

1. Smoke testing

A preliminary check to verify that the most critical functions work properly. Before conducting more extensive testing, teams run these smoke tests to ensure the application doesn't "catch fire" when started.

These quick tests check if users can log in, navigate basic screens, and perform essential operations without encountering major failures.

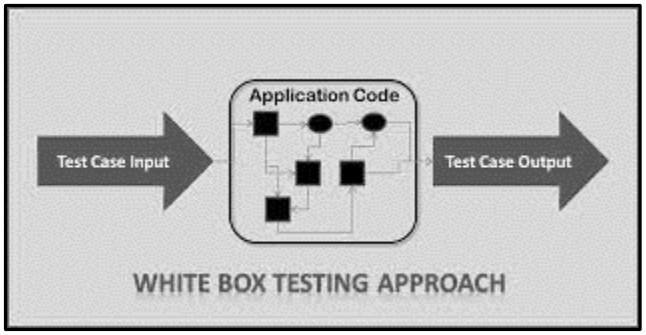
2. Integration testing

As its name suggests, it checks whether the different components or systems are working together as expected. When multiple modules are combined, integration testing confirms they communicate and exchange data correctly.

For example, testing might verify that a payment processing service properly integrates with a shopping cart system, maintaining data consistency throughout the transaction process.

White Box Testing

1. **White Box Testing** (also known as Clear Box Testing, Open Box Testing, Glass Box Testing, Transparent Box Testing, Code-Based Testing or Structural Testing) is a software testing method in which the internal structure/design/implementation of the item being tested is known to the tester.
2. The tester chooses inputs to exercise paths through the code and determines the appropriate outputs.
3. Programming know-how and the implementation knowledge is essential.
4. White box testing is testing beyond the user interface and into the nitty- gritty of a system.



Advantages

i. Testing can be commenced at an earlier stage. One need not wait for the GUI to be available.

ii. Testing is more thorough, with the possibility of covering most paths.

White Box Testing Disadvantages

i. Since tests can be very complex, highly skilled resources are required, with thorough knowledge of programming and implementation.

ii. Test script maintenance can be a burden if the implementation changes too frequently.

iii. Since this method of testing it closely tied with the application being testing, tools to cater to every kind of implementation/platform may not be readily available.

iv. White Box Testing is like the work of a mechanic who examines the engine to see why the car is not moving.

Criteria	**Black Box Testing**	**White Box Testing**
Definition	Black Box Testing is a software testing method in which the internal structure/ design/ implementation of the item being tested is NOT known to the tester	White Box Testing is a software testing method in which the internal structure/ design/ implementation of the item being tested is known to the tester.
Levels Applicable To	Mainly applicable to higher levels of testing: Acceptance Testing System Testing	Mainly applicable to lower levels of testing: Unit Testing Integration Testing
Responsibility	Generally, independent Software Testers	Generally, Software Developers
Programming Knowledge	Not Required	Required
Implementation Knowledge	Not Required	Required

Basis for Test Cases	Requirement Specifications	Detail Design

Types of White Box Testing

White box testing can be done for different purposes at different places. There are three main types of White Box testing which is follows:-

1. **Path Testing**: White box testing will be checks all possible execution paths in the program to sure about the each one of the function behaves as expected. It helps verify that all logical conditions in the code are functioning correctly and efficiently with as properly manner, avoiding unnecessary steps with better code reusability.
2. **Loop testing**: It will be check that loops (for or while loops) in the program operate correctly and efficiently. It checks that the loop handles variables correctly and doesn't cause errors like infinite loops or logic flaws.
3. **Unit Testing**: Unit Testing checks if each part or function of the application works correctly. It will check the application meets design requirements during development.
4. **Mutation Testing**: It is a type of Software Testing that is performed to design new software tests and also evaluate the quality of already existing software tests. Mutation testing is related to modification a program in small ways.
5. **Integration Testing**: Integration Testing Examines how different parts of the application work together. After unit testing to make sure components work well both alone and together.
6. **Penetration testing**: Penetration testing, or pen testing, is like a practice cyber attack conducted on your computer systems to find and fix any weak spots before real attackers can exploit them. It focuses on web application security, where testers try to breach parts like APIs and servers to uncover vulnerabilities such as code injection risks from unfiltered inputs.

What is Static Testing?

Static Testing also known as Verification testing or Non-execution testing is a type of Software Testing method that is performed to check the defects in software without actually executing the code of the software application.

1. Static testing is performed in the early stage of development to avoid errors as it is easier to find sources of failures and it can be fixed easily.

2. Static testing is performed in the white box testing phase of the software development where the programmer checks every line of the code before handing it over to the Test Engineer.
3. The errors that can't not be found using Dynamic Testing, can be easily found by Static Testing.
4. It involves assessing the program code and documentation.
5. It involves manual and automatic assessment of the software documents.

Documents that are assessed in Static Testing are:

1. Test Cases
2. Test Scripts.
3. Requirement Specification.
4. Test Plans.
5. Design Document.
6. Source Code.

Static Testing Techniques

Below are some of the static testing techniques:

1. Informal Reviews: In informal review, all the documents are presented to every team member, they just review the document and give informal comments on the documents. No specific process is followed in this technique to find the errors in the document. It leads to detecting defects in the early stages.
2. Walkthroughs: Skilled people or the author of the product explains the product to the team and the scribe makes note of the review of comments.
3. Technical Reviews: Technical specifications of the software product are reviewed by the team of your peers to check whether the specifications are correct for the project. They try to find discrepancies in the specifications and standards. Technical specifications documents like Test Plan, Test Strategy, and requirements specification documents are considered in technical reviews.
4. Code Reviews: Code reviews also known as Static code reviews are a systematic review of the source code of the project without executing the code. It checks the syntax of the code, coding standards, code optimization, etc.
5. Inspection: Inspection is a formal review process that follows a strict procedure to find defects. Reviewers have a checklist to review the work products. They record the defects and inform the participants to rectify the errors.

Benefits of Static Testing

Below are some of the benefits of static testing:

1. Early detection of defects: Static testing helps in the early detection of defects by reviewing the documents and artifacts before execution, issues can be detected and resolved at an early stage, thus saving time and effort later in the development process.
2. Cost-effective: Static testing is more cost-effective than dynamic testing techniques. Defects found during static testing are much cheaper to find and fix for the organization than in dynamic testing. It reduces the development, testing, and overall organization cost.
3. Easy to find defects: Static testing easily finds defects that dynamic testing does not detect easily.
4. Increase development productivity: Static testing increases development productivity due to quality and understandable documentation, and improved design.
5. Identifies coding errors: Static testing helps to identify coding errors and syntax issues resulting in cleaner and more maintainable code.

Limitations of Static Testing

Below are some of the limitations of static testing:

1. Detect some issues: Static testing may not uncover all issues that could arise during runtime. Some defects may appear only during dynamic testing when the software runs.
2. Depends on the reviewer's skills: The effectiveness of static testing depends on the reviewer's skills, experience, and knowledge.
3. Time-consuming: Static testing can be time-consuming when working on large and complex projects.

What is Dynamic Testing?

Dynamic Testing is a type of Software Testing that is performed to analyze the dynamic behavior of the code. It includes the testing of the software for the input values and output values that are analyzed.

1. The purpose of dynamic testing is to confirm that the software product works in conformance with the business requirements.
2. It involves executing the software and validating the output with the expected outcome.
3. It can be with black box testing or white box testing.
4. It is slightly complex as it requires the tester to have a deep knowledge of the system.
5. It provides more realistic results than static testing.

Dynamic Testing Techniques

Dynamic testing is broadly classified into two types:

1. White box Testing: White box testing also known as clear box testing looks at the internal workings of the code. The developers will perform the white box testing where they will test every line of the program's code. In this type of testing the test cases are derived from the source code and the inputs and outputs are known in advance.
2. Black box Testing: Black box testing looks only at the functionality of the Application Under Test (AUT). In this testing, the testers are unaware of the system's underlying code. They check whether the system is generating the expected output according to the requirements. The Black box testing is further classified as, Functional Testing and Non-functional Testing.

Benefits of Dynamic Testing

Below are some of the benefits of dynamic testing:

1. Reveals runtime errors: Dynamic testing helps to reveal runtime errors, performance bottlenecks, memory leaks, and other issues that become visible only during the execution.
2. Verifies integration of modules: Dynamic testing helps to verify the integration of modules, databases, and APIs, ensuring that the system is working seamlessly.
3. Accurate reliability assessment: Dynamic testing helps to provide accurate quality and reliability assessment of the software thus verifying that the software meets the specified requirements and functions as intended. This helps to make sure that the software functions correctly in different usage scenarios.

Limitations of Dynamic Testing

Below are some of the limitations of dynamic testing:

1. Time-consuming: Dynamic testing can be time-consuming in the case of complex systems and large test suites.
2. Requires effort: It requires significant effort in complex systems to debug and pinpoint the exact cause.
3. Challenging: In case of testing exceptional or rare conditions it can be challenging to conduct.
4. May not cover all scenarios: Dynamic testing may not cover all possible scenarios due to a large number of potential inputs and execution paths.

Static Testing vs Dynamic Testing

Below are the differences between static testing and dynamic testing:

Parameters	Static Testing	Dynamic Testing
Definition	Static testing is performed to check the defects in the software without actually executing the code.	Dynamic testing is performed to analyze the dynamic behavior of the code.
Objective	The objective is to prevent defects.	The objective is to find and fix defects.
Stage of execution	It is performed at the early stage of software development.	It is performed at the later stage of the software development.
Code Execution	In static testing, the whole code is not executed.	In dynamic testing, the whole code is executed.
Before/ After Code Deployment	Static testing is performed before code deployment.	Dynamic testing is performed after code deployment.
Cost	Static testing is less costly.	Dynamic testing is highly costly.
Documents Required	Static Testing involves a checklist for the testing process.	Dynamic Testing involves test cases for the testing process.
Time Required	It generally takes a shorter time.	It usually takes a longer time as it involves running several test cases.
Bugs	It can discover a variety of bugs.	It exposes the bugs that are explorable through execution hence discovering only a limited type of bugs.
Statement Coverage	Static testing may complete 100% statement coverage incomparably in less time.	Dynamic testing only achieves less than 50% coverage.

Parameters	Static Testing	Dynamic Testing
Techniques	It includes Informal reviews, walkthroughs, technical reviews, code reviews, and inspections.	It involves functional and non-functional testing.
Example	It is a verification process.	It is a validation process.

Configuration testing is carried out to check whether the software performs correctly under different hardware and software configurations. It ensures that the application behaves as expected when executed in varying environments such as different operating systems, browsers, networks, memory sizes, and processors. For example, an application may be tested on Windows, Linux, and Mac platforms to validate consistent performance.

Compatibility testing focuses on verifying that the software is compatible with other systems, environments, or applications. It examines whether the application functions properly with different versions of browsers, operating systems, or hardware and whether it can integrate smoothly with other applications. Compatibility can be forward, where the software works with future versions of an environment, or backward, where it supports older versions.

Graphical User Interface testing, often abbreviated as GUI testing, deals with checking the visual elements of the software to ensure consistency, usability, and correctness. It validates that buttons, menus, icons, layouts, colors, and fonts are properly displayed and function as expected. Alongside GUI testing, **documentation testing** is performed to confirm that user manuals, online help files, and guides are accurate, consistent, and aligned with the software's actual functionality so that users can rely on them without confusion.

Security testing is concerned with the protection of data and resources. It ensures that the application is safeguarded against unauthorized access, data breaches, and vulnerabilities. Through techniques such as penetration testing, vulnerability scanning, authentication and authorization checks, and encryption validation, security testing guarantees confidentiality, integrity, and availability of information within the system.

Test planning refers to the process of defining the scope, objectives, approach, resources, and overall strategy for conducting testing activities. It establishes what will be tested, how testing will be performed, the entry and exit criteria, as well as the risks and deliverables associated with the testing process.

Test budgeting is the estimation of costs required for the complete testing process. It accounts for expenses such as manpower, testing tools, training, infrastructure, test

environments, automation costs, and defect management. An effective budget helps in resource allocation and prevents overspending.

Test scheduling involves preparing a timeline for testing activities so that the process is completed efficiently within project deadlines. It specifies when each test phase will begin and end, sets milestones, allocates tasks to team members, and considers dependencies such as code readiness or environment availability. A well-prepared schedule ensures that testing aligns smoothly with the overall software development life cycle.

Test Planning

A test plan is a comprehensive document that outlines all testing-related activities for a project. It serves as a blueprint for testing, detailing what will be tested, how it will be tested, by whom, and the distribution of test types among testers. Test planning ensures that testing activities are organized, efficient, and aligned with project goals.

Purpose:

- Acts as a roadmap for the QA team.
- Coordinates testing activities among testers and other stakeholders, such as Business Analysts and Project Managers.
- Serves as a reference for monitoring progress, risks, and resource allocation.
- Helps maintain a current and adaptable plan as the project evolves.

Test Planning Process:

- Preparation: The test manager prepares the test plan before the testers are involved.
- Documentation: The plan includes objectives, scope, strategy, resources, schedule, risks, and success criteria.
- Execution & Coordination: It guides the QA team in conducting tests and ensures effective communication with stakeholders.
- Adaptation: The test plan is continuously updated according to project progress.

Phases of Test Planning

1. Analyze the Product

- Understand the product by performing requirement analysis, walkthroughs, and interviews with clients, designers, and developers.
- Key focus areas:
    - Primary objectives and purpose of the product.
    - Intended users and target audience.

- Hardware, software, and platform specifications.

2. Design the Test Strategy

- The Test Strategy is a high-level document defining how testing will be conducted.
- Key elements include:
  - Scope of Testing: Features/modules to be tested and those excluded.
  - Types of Testing: Functional, non-functional, regression, performance, security, etc.
  - Risk Analysis: Identify potential risks and mitigation plans.
  - Test Logistics: Assign responsibilities, define resource allocation, and tools to be used.

3. Define Test Objectives

- Establish clear goals for the testing process.
- Objectives include:
  - Features and functionalities to be validated (e.g., GUI, performance, workflows).
  - Expected outcomes and success criteria for each feature.
  - Compliance with standards, business requirements, and user expectations.

4. Define Test Criteria

- Suspension Criteria: Benchmarks that determine when testing should be paused due to issues such as critical defects or unstable builds.
- Exit Criteria: Benchmarks defining successful completion of testing, ensuring all test objectives and quality standards are met.
- Additional: Define pass/fail criteria for each test scenario for clarity in reporting.

5. Resource Planning

- Identify all resources required to execute the testing plan efficiently.
- Includes:
  - Human resources: QA engineers, test managers, developers for defect fixes.
  - Hardware & software: Devices, servers, operating systems, browsers.
  - Tools and infrastructure: Test management, defect tracking, automation tools.

6. Plan Test Environment

- Setup realistic testing environments that simulate actual user conditions.

- Ensure environments include:
    - Different OS versions, browsers, and network settings.
    - Required software configurations, databases, and third-party integrations.
    - Access to sandbox or staging servers for safe testing.
- Additional: Maintain environment documentation to ensure repeatability of tests.

7. Schedule and Estimation

- Break down the project into smaller, manageable tasks with estimated effort.
- Define timelines, milestones, and dependencies for each testing activity.
- Additional considerations:
    - Buffer time for defect fixing and retesting.
    - Parallel execution of tests where feasible to optimize schedule.
    - Integration with project management tools for tracking progress.

8. Determine Test Deliverables

- Identify all outputs required for successful testing:
    - Test cases, test scripts, test data, and automation scripts.
    - Test summary reports, defect logs, and metrics dashboards.
    - Test environment setup documents and configuration records.
- Ensure deliverables are updated, shared, and approved by stakeholders.

Budgeting and Scheduling

Budgeting and Scheduling are two essential components of project management. They work hand in hand to ensure that a project is completed within its financial limits and on time. While budgeting focuses on managing financial resources, scheduling focuses on managing time and task sequences.

1. Budgeting

Budgeting is the process of creating a financial plan for how money will be spent during a project. It involves estimating costs, allocating funds, and ensuring resources are used efficiently.
To ensure a project stays within its financial limits and to help manage monetary resources effectively throughout the project lifecycle.

Steps in Budgeting:

1. Determine income and estimate all expected expenses.

2. Identify cost elements such as labor, materials, and equipment.
3. Estimate the cost of each activity, task, or goal.
4. Allocate funds to each activity based on priority and estimation.
5. Regularly monitor and adjust the budget as the project progresses.

Example:
If a software development project has a total budget of ₹5,00,000, the funds might be allocated to salaries, tools, testing, and training based on their estimated cost.

Types of Budgeting Methods

1. Zero-Based Budgeting (ZBB):

Starts from zero each period — every expense must be justified from scratch. Focuses on cost efficiency.

2. Incremental Budgeting:

Uses last year's budget as a base and adds or subtracts a fixed amount. Simple but may continue inefficiencies.

3. Activity-Based Budgeting (ABB):

Allocates funds based on the cost of specific activities that support business goals. Focuses on performance and cost drivers**.**

4. Value Proposition Budgeting (VPB):

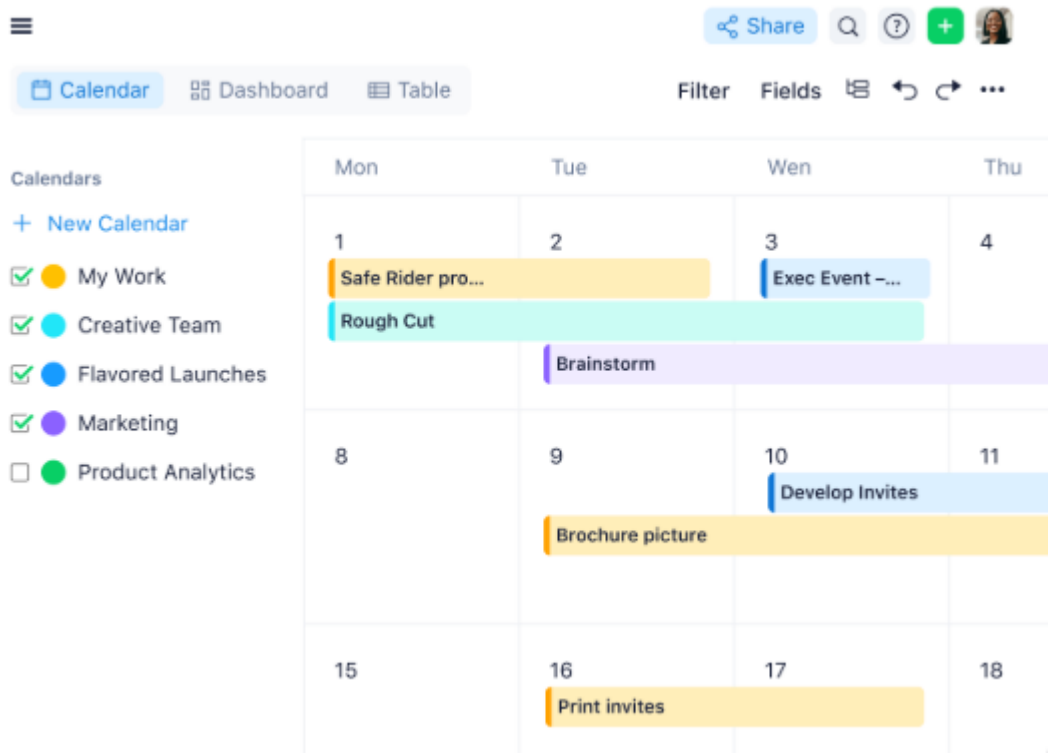
Allocates resources based on the value delivered to customers or stakeholders. Ensures every expense adds measurable value.

2. Scheduling

Scheduling is the process of creating a timeline that defines when activities will start and finish. It ensures that project tasks are completed in the correct sequence and within the plannedduration.
To determine when each activity should be performed and to ensure the project is completed on time.

Steps in Scheduling:

1. Identify and list all project tasks and activities.
2. Estimate the time required to complete each task.
3. Determine task dependencies (which tasks depend on others).
4. Arrange tasks in a logical and sequential order.
5. Represent the schedule visually using tools such as a Gantt Chart, PERT Chart, or Network Diagram.



Example:
If “Design UI” must be completed before “Develop Frontend,” the schedule ensures that development starts only after the design phase is finished.

3. Relationship Between Budgeting and Scheduling

Budgeting and Scheduling are interdependent and must be developed together for effective project planning and control.

- The schedule outlines when and how work will be performed.
- The budget assigns monetary values to that scheduled work.
- Any change in the schedule can affect the budget, and vice versa.

Example:
If a task is scheduled to take 10 hours and the labor cost is ₹500/hour, then ₹5,000 must be allocated in the budget for that task. If the task takes longer, both the schedule and budget will need adjustments.

UNIT-3

Software testing Metrics

Software testing metrics are quantifiable indicators used to evaluate the progress, quality, productivity, efficiency, and overall health of the software testing process. They provide data-driven insights that help teams make informed decisions for process improvement, quality assurance, and future planning.

The main purpose of software testing metrics is to enhance the efficiency and effectiveness of the testing process by providing accurate, measurable data. These metrics help assess how well testing objectives are being met and guide teams in optimizing their testing strategies.

In general, a metric expresses the degree to which a system, component, or process possesses a particular attribute in numerical terms. For example, the weekly mileage of an automobile compared to its ideal mileage specified by the manufacturer is a practical illustration of a metric.

Key software testing metrics include:

- **Test Coverage:** Measures how much of the code or functionality has been tested.
- **Defect Density:** Represents the number of defects found per unit of code.
- **Test Execution Rate:** Compares the number of tests executed versus those planned.
- **Defect Leakage:** Indicates the number of defects discovered after the software has been released.

Software test metrics are also valuable in defining and maintaining quality standards. By regularly monitoring these metrics, teams can track their productivity, evaluate testing progress, identify bottlenecks, and continuously improve their testing approach.

Types of Software Testing Metrics

Software testing metrics are generally categorized into three main types — Process Metrics, Product Metrics, and Project Metrics. Each category serves a distinct purpose in evaluating and improving different aspects of the software testing life cycle. These metrics help Quality Assurance (QA) teams assess performance, maintain standards, and ensure continuous process enhancement.

Before defining and applying these metrics, it is important to:

- Clearly set testing standards and procedures.
- Identify the required data for accurate measurement.
- Select an appropriate testing framework that aligns with project goals.
- Define the purpose and target audience for each metric.
- Customize the measurements according to project needs and phases.

1. Process Metrics

Process Metrics focus on evaluating the efficiency and effectiveness of testing activities. They help in identifying areas for process optimization and improvement within the Software Development Life Cycle (SDLC).

- Test Case Effectiveness:
  Measures how efficiently test cases identify defects.
  Formula: (Defects Detected / Test Cases Run) × 100
  Use Case: Evaluates the quality of test cases and helps refine them for better defect detection.
- Cycle Time:
  Tracks the total time taken to complete a testing cycle.
  Insight: Assists in determining testing speed and identifying bottlenecks or inefficiencies in the process.
  Example: If a testing cycle consistently exceeds the expected duration, it may signal environment or resource-related delays.
- Defect Fixing Time:
  Calculates the time required to fix a defect from detection to closure.
  Use Case: Helps pinpoint slowdowns in the defect resolution process and optimize team workflows.

2. Product Metrics

Product Metrics are used to assess the quality, performance, and reliability of the software product being tested. These metrics provide insights into the stability and robustness of the system.

Number of Defects:

Reflects the overall quality of the product and the efficiency of the testing effort.
Insight: Helps teams locate problematic modules and apply corrective actions.

Defect Severity:

Categorizes defects according to their impact on the system.
Common Categories: Critical, Major, and Minor.
Use Case: Enables prioritization of bug fixes so that high-severity issues are resolved first.

Passed/Failed Test Case Metrics:

Measures the proportion of passed test cases against the total executed cases.
Formula: (Passed Test Cases / Total Test Cases) × 100
Use Case: Assists in analyzing the system's stability and identifying failure-prone **areas.**

3. Project Metrics

Project Metrics evaluate the overall progress, resource utilization, and cost-effectiveness of the testing project. They help ensure that testing activities are aligned with the project's timeline and budget.

Test Coverage:

Indicates the percentage of functionalities tested.
Formula: (Tested Functionalities / Total Functionalities) × 100
Use Case: Confirms that all key functionalities are covered during testing.

Cost of Testing:

Represents the total expenditure incurred in the testing phase, including resource and infrastructure costs.
Use Case: Helps in budget management and identifying cost-saving opportunities.

Budget/Schedule Variance:

Monitors deviations from planned costs and schedules.
Use Case: Ensures adherence to project deadlines and financial plans, minimizing risks of overruns.

Formula for Test Metrics:

To get the percentage execution status of the test cases, the following formula can be used:

Percentage test cases executed = (No of test cases executed / Total no of test cases written) x 100

Similarly, it is possible to calculate for other parameters also such as test cases that were not executed, test cases that were passed, test cases that were failed, test cases that were blocked, and so on. Below are some of the formulas:

1. Test Case Effectiveness:

Test Case Effectiveness = (Number of defects detected / Number of test cases run) x 100

2. Passed Test Cases Percentage: Test Cases that Passed Coverage is a metric that indicates the percentage of test cases that pass.

Passed Test Cases Percentage = (Total number of tests ran / Total number of tests executed) x 100

3. Failed Test Cases Percentage: This metric measures the proportion of all failed test cases.

Failed Test Cases Percentage = (Total number of failed test cases / Total number of tests executed) x 100

4. Blocked Test Cases Percentage: During the software testing process, this parameter determines the percentage of test cases that are blocked.

Blocked Test Cases Percentage = (Total number of blocked tests / Total number of tests executed) x 100

5. Fixed Defects Percentage: Using this measure, the team may determine the percentage of defects that have been fixed.

Fixed Defects Percentage = (Total number of flaws fixed / Number of defects reported) x 100

6. Rework Effort Ratio: This measure helps to determine the rework effort ratio.

Rework Effort Ratio = (Actual rework efforts spent in that phase/ Total actual efforts spent in that phase) x 100

7. Accepted Defects Percentage: This measures the percentage of defects that are accepted out of the total accepted defects.

Accepted Defects Percentage = (Defects Accepted as Valid by Dev Team / Total Defects Reported) x 100

8. Defects Deferred Percentage: This measures the percentage of the defects that are deferred for future release.

Defects Deferred Percentage = (Defects deferred for future releases / Total Defects Reported) x 100

Example of Software Test Metrics Calculation:

Let's take an example to calculate test metrics:

S No.	Testing Metric	Data retrieved during test case development
1	No. of requirements	5

S No.	Testing Metric	Data retrieved during test case development
2	The average number of test cases written per requirement	40
3	Total no. of Test cases written for all requirements	200
4	Total no. of Test cases executed	164
5	No. of Test cases passed	100
6	No. of Test cases failed	60
7	No. of Test cases blocked	4
8	No. of Test cases unexecuted	36
9	Total no. of defects identified	20
10	Defects accepted as valid by the dev team	15
11	Defects deferred for future releases	5
12	Defects fixed	12

1. Percentage test cases executed = (No of test cases executed / Total no of test cases written)x100

=(164/200)x100
= 82

2. Test Case Effectiveness = (Number of defects detected / Number of test cases run) x 100

=(20/164)x100
= 12.2

3. Failed Test Cases Percentage = (Total number of failed test cases / Total number of tests executed) x 100

= (60 / 164) * 100
= 36.59

4. Blocked Test Cases Percentage = (Total number of blocked tests / Total number of tests executed) x 100

= (4 / 164) * 100
= 2.44

5. Fixed Defects Percentage = (Total number of flaws fixed / Number of defects reported) x 100

= (12 / 20) * 100
= 60

6. Accepted Defects Percentage = (Defects Accepted as Valid by Dev Team / Total Defects Reported) x 100

= (15 / 20) * 100
= 75

7. Defects Deferred Percentage = (Defects deferred for future releases / Total Defects Reported) x 100

= (5 / 20) * 100
= 25

Defect Management

What is a Defect?

A defect is any flaw, error, or deviation from the expected behavior of a software application. It occurs when the actual result of a program does not match the intended or specified outcome.

Defects may arise due to various reasons such as:

- Errors in code or logic
- Incomplete or incorrect implementation
- Miscommunication in requirements
- Integration issues between software modules

Defects are also referred to as bugs, issues, or faults, depending on the terminology used within the organization or testing team.

Defect Management

Defect Management is the systematic process of identifying, reporting, tracking, and resolving defects (or bugs) found during software development and testing. It plays a vital role in ensuring software quality, customer satisfaction, and cost efficiency by detecting and fixing issues early in the development lifecycle.

The process involves several steps, such as analyzing test results, prioritizing defects, assigning them to developers for resolution, and verifying the fixes to ensure the issues are properly addressed. Defect management is a continuous and iterative process carried out throughout the software development lifecycle to maintain a high-quality product that meets user expectations.

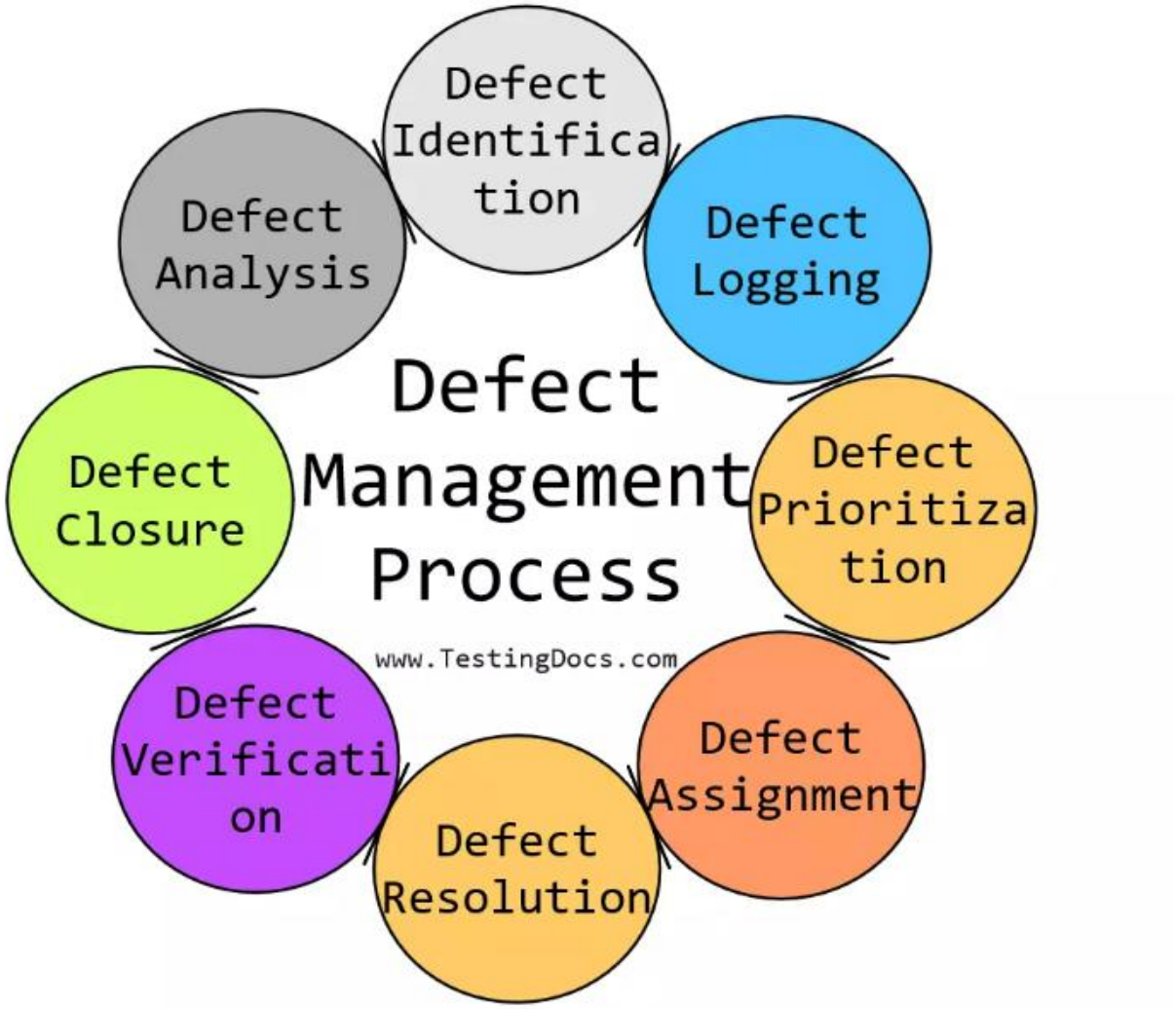
Role of Automation in Defect Management

Test automation tools play a crucial role in effective defect management by automating test execution and defect reporting processes. Modern automation platforms such as Testsigma integrate seamlessly with popular bug-tracking tools like Jira, Trello, and Bugzilla.

These integrations allow automatic generation and logging of bug reports directly from failed test cases, thereby:

- Saving time
- Reducing manual errors in bug reporting
- Enhancing collaboration between testing and development teams

Defect Management Process and Metrics related to defects



Defect Management Process

The Defect Management Process is a structured approach used to identify, record, analyze, prioritize, fix, verify, and close defects that arise during the software development and testingphases.
Its primary goal is to ensure that every defect is addressed efficiently to maintain high-quality software and reduce future risks or rework.

Below are the key stages involved in the defect management process:

1. Defect Identification

Defects are detected during different levels of testing such as unit testing, integration testing, system testing, and user acceptance testing.

- Testers compare the expected outcome with the actual result to find deviations.
- Defects can also be identified during code reviews, walkthroughs, or customer feedback sessions.
- Early detection helps reduce the cost and impact of defects later in the development cycle.

2. Defect Logging

Once a defect is found, it must be documented in a defect tracking tool (such as Jira, Bugzilla, or Trello).

- Each defect entry includes:
    - Unique Defect ID
    - Description of the issue
    - Module or component name
    - Steps to reproduce the defect
    - Expected vs. actual results
    - Severity and priority levels
    - Screenshots or logs (if any)
- Proper documentation ensures that the defect can be reproduced and fixed efficiently.

3. Defect Triage

The triage process involves analyzing and prioritizing defects based on their impact on the system and urgency for resolution.

- The QA team, project manager, and development team jointly decide which defects should be addressed first.
- Factors such as severity, customer impact, frequency of occurrence, and resource availability are considered.
- This step ensures optimal use of resources and helps avoid unnecessary delays.

4. Defect Assignment

Once triaged, defects are assigned to appropriate developers or teams based on their expertise and workload.

- Clear ownership helps ensure accountability and timely resolution.
- The defect tracking tool updates the status to “Assigned”, and notifications are sent to the responsible personnel.

5. Defect Resolution

In this stage, the assigned developer analyzes the root cause of the defect and applies the required fix.

- This may involve code correction, configuration changes, or updating system documentation.
- After implementing the fix, the developer changes the status to “Fixed” or “Resolved” and passes it back to the testing team for verification.

6. Defect Verification

After a defect has been resolved, the testing team re-tests the application to ensure that:

- The defect has been fixed successfully.
- No new defects have been introduced because of the fix (regression testing).
- If the issue still persists, the defect is reopened and reassigned for further investigation.

7.Defect Closure

- Once the tester confirms that the defect has been resolved and the functionality works as expected, the defect status is updated to “Closed.”
- Closure includes verifying that all documentation has been updated and that the fix has not impacted other modules.
- Closed defects are stored for future reference and analysis to avoid similar issues.

9. Root Cause Analysis (RCA)

After closure, a root cause analysis is often conducted to identify why the defect occurred in the first place.

- This helps prevent similar defects in the future.
- RCA may include reviewing requirements, testing strategies, or coding standards to identify process weaknesses.

Goals of Defect Management Process (DMP)

- Prevent the defect
- Detection at an early stage
- Reduce the impact or effects of defect on software
- Resolving or fixing defects
- Improving process and performance of software

Metric Name	Formula	Purpose / Insight
1. Defect Density	(Total Defects Found / Size of the Software) × 1000	Measures the number of defects per unit of code (e.g., per 1000 lines of code). Helps evaluate code quality.
2. Defect Removal Efficiency (DRE)	(Defects Removed During Testing / Total Defects Found) × 100	Indicates how effectively the testing process is identifying and removing defects before release.
3. Defect Leakage	(Defects Found After Release / Total Defects Found) × 100	Measures the percentage of defects that escaped into production. Lower values indicate better testing.
4. Defect Severity Index (DSI)	Σ (Severity Level × No. of Defects at that Level) / Total Defects	Reflects the overall impact of defects based on severity levels (Critical, Major, Minor).
5. Mean Time to Detect (MTTD)	Total time taken to detect all defects / Total number of defects detected	Measures how quickly defects are discovered after introduction.
6. Mean Time to Repair (MTTR)	Total time taken to fix all defects / Total number of defects fixed	Evaluates the average time required to resolve defects.
7. Defect Age	Time between defect detection and its closure	Helps track the time defects remain open and identify delays in resolution.
8. Reopened Defect	(No. of Reopened Defects / Total	Indicates how many fixed defects were reopened,

Metric Name	Formula	Purpose / Insight
Percentage	No. of Defects Fixed) × 100	showing fix quality or testing thoroughness.

Defect metrics

Defect metrics in software testing quantify software quality by measuring various aspects of defects, such as Defect Density (defects per unit of code), Defect Leakage (defects found post-release), Defect Severity (impact on the system), and Defect Removal Efficiency (ability to fix defects). Tracking these metrics helps teams evaluate testing effectiveness, identify problem areas, prioritize bug fixes, and improve the software development life cycle.

Example:

Let's assume during testing:

- Total defects found = 100
- Software size = 20,000 LOC (Lines of Code)
- Defects removed during testing = 90
- Defects found after release = 10

Defect Density = (100 / 20,000) × 1000 = **5** defects/KLOC
Defect Removal Efficiency = (90 / 100) × 100 = 90%
Defect Leakage = (10 / 100) × 100 = **10%**

Benefits of Using Defect Metrics

- Provides quantitative insights into software quality
- Helps track the effectiveness of testing efforts
- Identifies problematic modules with recurring defects
- Improves project estimation and planning
- Enhances accountability and process improvement

Configuration and Compatibility Testing

Configuration Testing is a type of software testing that verifies the performance, stability, and functionality of a system under different combinations of software and hardware environments.

The goal is to identify the best possible configuration under which the system can work efficiently without any errors or performance issues, while meeting all functional requirements.

Different configurations may include:

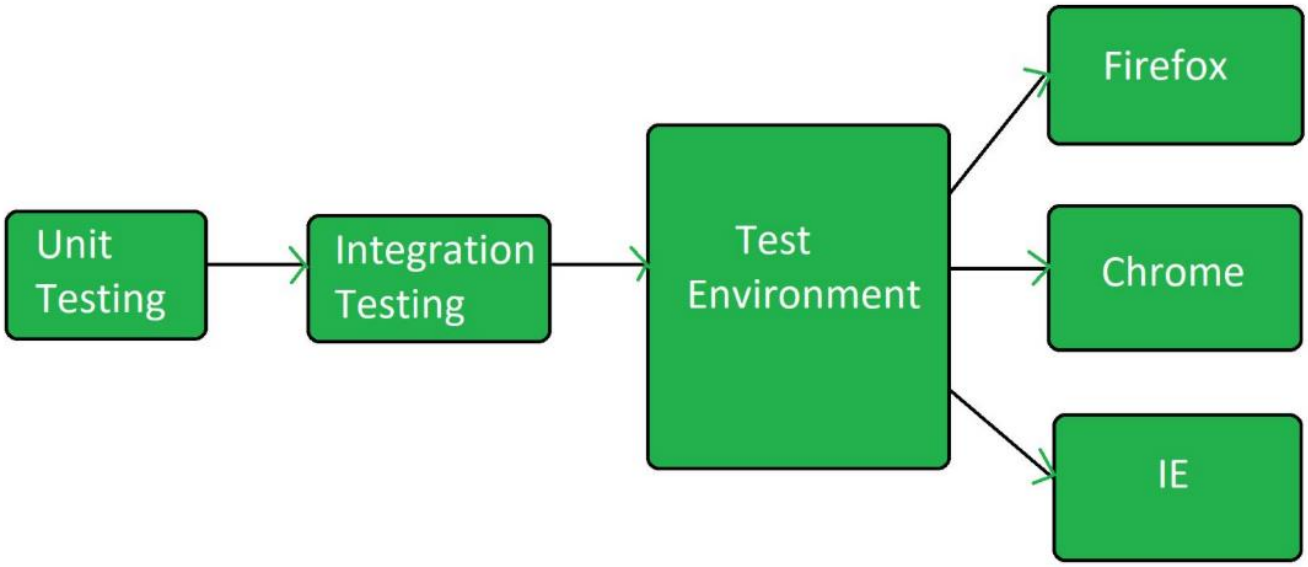
- **Operating Systems:** Windows XP, Windows 7 (32/64 bit), Windows 8, Windows 10, Linux, macOS, etc.
- **Databases:** Oracle, MySQL, DB2, MSSQL Server, Sybase, etc.
- **Browsers:** Internet Explorer 8/9, Firefox 16.0, Chrome, Microsoft Edge, Safari, etc.
- **Hardware Components:** Different CPUs, memory sizes, hard drives, and peripheral devices (printers, scanners, webcams, etc.).

This testing ensures that the application behaves consistently across all these combinations and identifies any configuration-specific defects early.

Objectives of Configuration Testing

- To determine the best configuration setup for the application.
- To identify defects caused by specific settings or environmental changes (e.g., language, regional, or time zone differences).
- To evaluate system performance by adding or modifying resources such as CPU, memory, or load balancers.
- To ensure the system meets performance and functional requirements in different supported environments.
- To confirm the system's compatibility with other software and hardware mentioned in the Software Requirement Specification (SRS).

Example of Configuration Testing



Let’s consider a three-tier desktop application developed using ASP.NET, which includes:

- Client Layer: Windows XP, Windows 7, Windows 8, Windows 10
- Business Logic Server Layer: Windows Server 2008 R2, Windows Server 2012 R2
- Database Layer: SQL Server 2008, SQL Server 2008 R2, SQL Server 2012

To ensure proper functionality, the tester must test different combinations of:

- **Client OS**
- Server Platform
- Database Version

For instance:

- (Windows 7 + Windows Server 2012 R2 + SQL Server 2012)
- (Windows 10 + Windows Server 2008 R2 + SQL Server 2008 R2)

Testing all such combinations helps confirm that the application performs correctly and does not fail under any supported configuration.

Hardware Configuration Testing

Configuration Testing is not limited to software—it also applies to hardware. In Hardware Configuration Testing, the application is tested with various physical devices like:

- Printers
- Scanners
- Webcams
- Storage devices

This ensures that the system interacts smoothly with all hardware components it supports.

Purpose of Configuration Testing

The main purposes are:

1. To evaluate system performance and stability under different system settings.
2. To discover the optimal configuration for achieving maximum performance and reliability.
3. To detect and resolve configuration-related defects before the software is released.
4. To ensure that the system remains compatible and functional with all the software and hardware combinations defined in the project requirements.

By conducting Configuration Testing, teams can minimize configuration-related issues in production and ensure a seamless user experience across diverse environments.

Compatibility Testing

Compatibility Testing is performed to determine whether a software application can run smoothly on various platforms without any issues. It helps identify environment-specific defects that might affect usability, appearance, or performance.

This testing is conducted only after the application becomes stable, as it focuses on verifying how well the application interacts with other system components rather than finding functional bugs.

For example, a web application might work perfectly in Google Chrome, but display layout issues or slow loading times in Mozilla Firefox or Safari. Compatibility testing helps detect and fix such inconsistencies.

Objectives of Compatibility Testing

- To ensure the application performs consistently across all supported environments.
- To verify the software’s usability and accessibility for all users, regardless of their platform.
- To detect issues related to UI alignment, data handling, performance, or display errors in different systems.
- To confirm that updates or upgrades in operating systems, browsers, or drivers don’t break the existing functionality.
- To enhance customer satisfaction by ensuring a uniform experience across multiple devices and configurations.

Types of Compatibility Testing

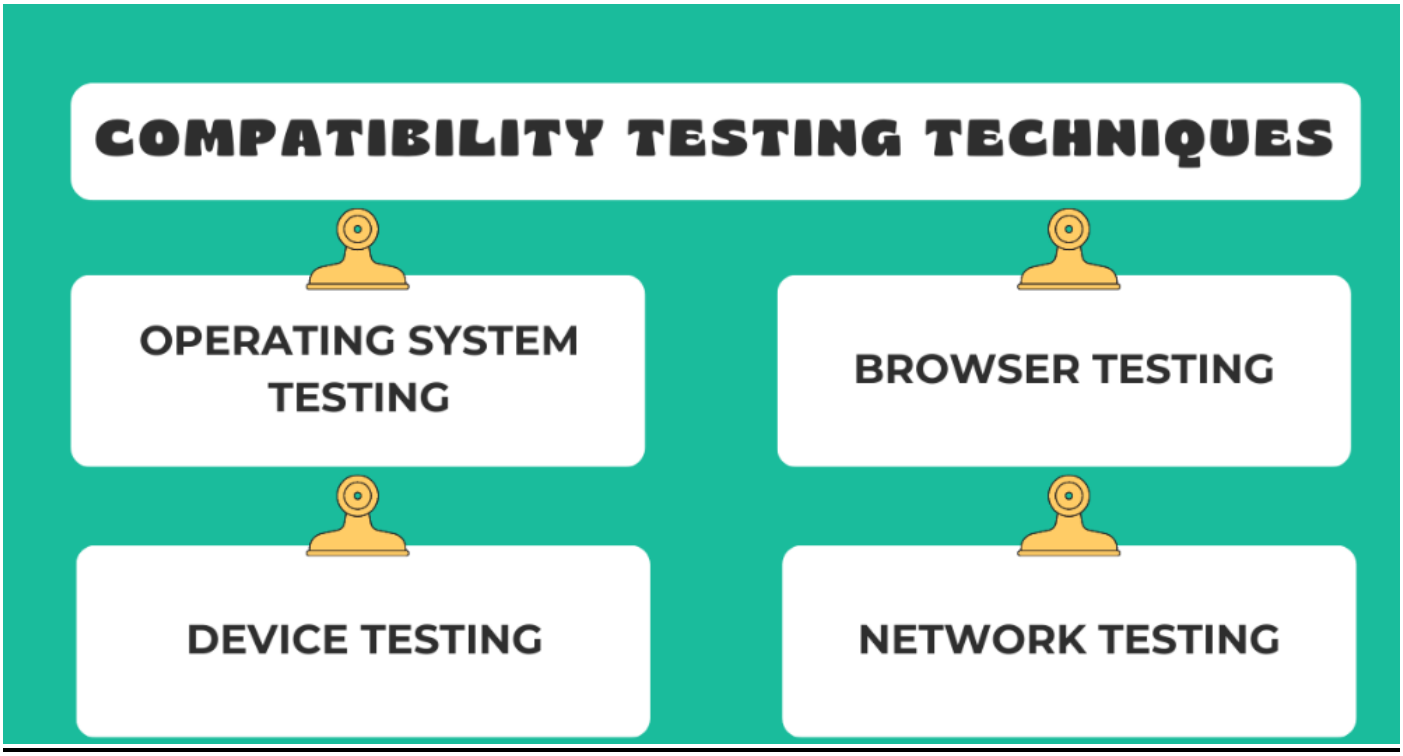
Hardware Compatibility Testing:
Ensures the software works correctly with different hardware devices like printers, scanners, sound cards, and network adaptors.
Software Compatibility Testing:
Checks how the application interacts with other software, such as databases, browsers, or third-party tools.
Operating System Compatibility Testing:
Verifies the software's behavior on different OS versions like Windows, Linux, macOS, Android, or iOS.
Browser Compatibility Testing:
Ensures web applications work correctly across various browsers and their versions—like Chrome, Firefox, Safari, and Edge.
Network Compatibility Testing:
Tests the software's performance under different network conditions (Wi-Fi, 4G, 5G, or low bandwidth).
Mobile Device Compatibility Testing:
Confirms that mobile applications run smoothly on different screen sizes, resolutions, and OS versions.

Example

Suppose a web-based e-commerce application is developed. The testing team performs compatibility testing to ensure it:

- Loads correctly on Windows, macOS, and Linux.
- Displays properly on browsers like Chrome, Firefox, Safari, and Edge.
- Functions well on devices like desktops, tablets, and smartphones.
- Performs consistently under different network speeds.

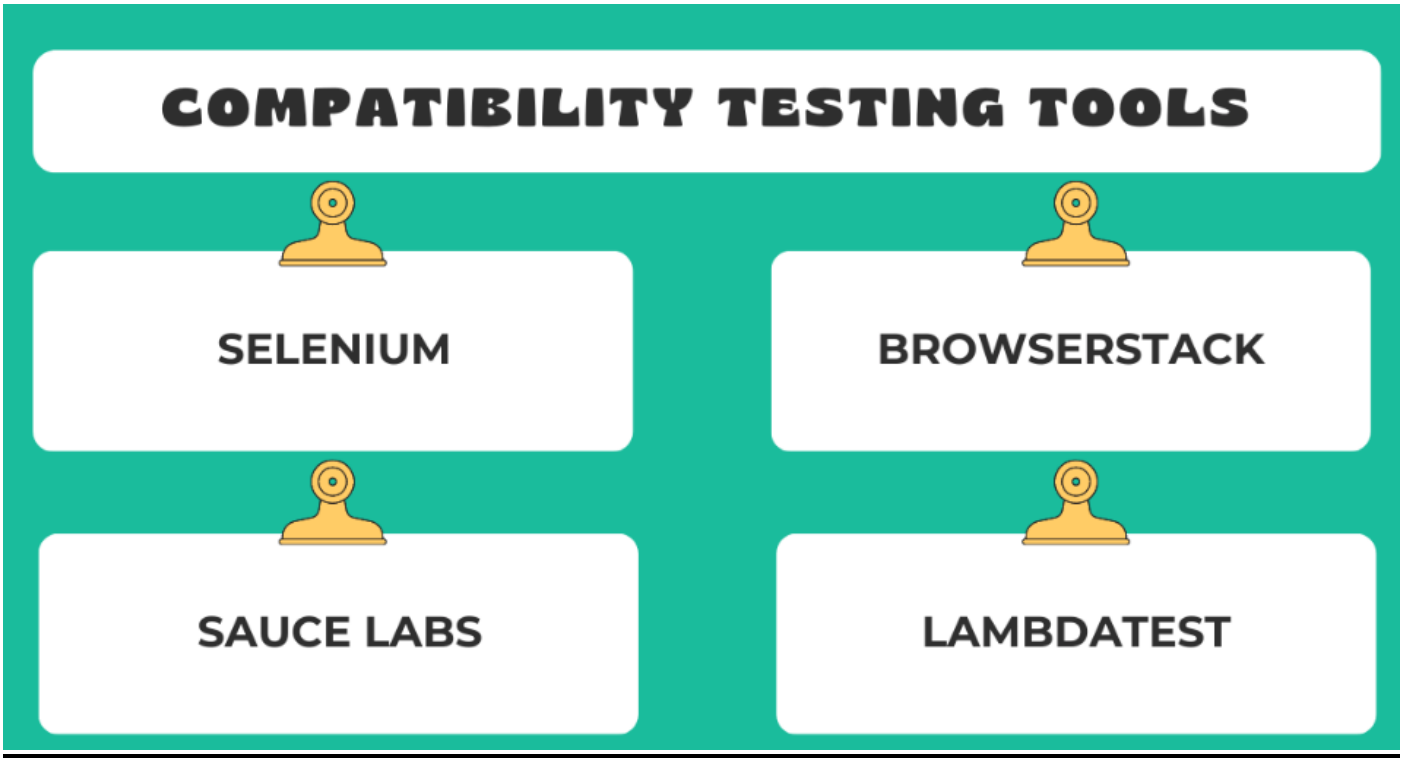
Compatibility Testing Techniques:



1. **Operating System Testing:**
    - Ensures the application works correctly across different **operating systems** (Windows, macOS, Linux, Android, iOS, etc.) and their various **versions**.
    - Helps identify OS-specific issues such as installation errors, UI distortions, or feature malfunctions.
    - Example: Testing a software application on **Windows 10, Windows 11, macOS Ventura, and Ubuntu 22.04** to ensure consistent performance.
2. **Browser Testing:**
    - Verifies that **web applications** render and function properly across multiple browsers such as **Google Chrome, Mozilla Firefox, Safari, Microsoft Edge, and Opera**, including older and newer versions.
    - Ensures **layout consistency**, **JavaScript functionality**, **CSS rendering**, and **performance** remain stable.
    - Example: A webpage's elements may appear correctly in Chrome but misaligned in Safari — browser testing detects and fixes these issues.
3. **Device Testing:**
    - Confirms that the application performs well on a variety of **devices** like **desktops, laptops, tablets, and smartphones**.
    - Ensures proper **UI responsiveness**, **display scaling**, and **touch interaction** across different screen sizes and resolutions.
    - Example: Testing a mobile banking app on Android, iPhone, and iPad devices to verify uniform usability.

4. **Network Testing:**
    - Evaluates how the application behaves under various **network conditions** such as **Wi-Fi, 4G, 5G, and low bandwidth connections**.
    - Helps identify issues related to **load time, data synchronization, or connection loss**.
    - Example: Ensuring that an e-commerce site loads correctly even with slow network speeds.

Compatibility Testing Frameworks and Tools



1. **Selenium:** A widely-used automation tool for browser compatibility testing.
2. **BrowserStack:** Offers cloud-based testing on real devices and browsers.
3. **Sauce Labs:** Provides a cloud-based platform for automated and manual testing across various environments.
4. **LambdaTest:** A cross-browser testing tool with a wide range of browser and OS combinations.

Software Testing Tools

Software testing tools are applications that assist in testing software to ensure quality, performance, and reliability. They help in both manual and automated testing, covering unit, integration, system, and non-functional testing activities.

Types of Software Testing Tools

1. **Static Testing Tools:** Analyze code or documentation without executing the software.
    - Examples: Flow Analyzers, Path Tests, Coverage Analyzers, Interface Analyzers.
2. **Dynamic Testing Tools:** Test software by executing it with real or simulated data.

    Examples: Test Drivers, Test Beds, Emulators, Mutation Analyzers.
3. **By Testing Purpose:**
    - Test Management Tools: Plan, manage, and report testing activities (e.g., JIRA, TestLodge).
    - Automated Testing Tools: Automate testing tasks to save time and increase accuracy (e.g., Selenium, Appium, Katalon).
    - Performance Testing Tools: Evaluate stability, speed, scalability, and performance (e.g., JMeter, WebLOAD, NeoLoad).
    - Cross-Browser Testing Tools: Test web apps across different browser-OS combinations (e.g., LambdaTest, Testsigma).
    - Integration Testing Tools: Test interfaces between modules (e.g., Citrus, FitNesse).
    - Unit Testing Tools: Validate individual modules (e.g., JUnit, PHPUnit).
    - Mobile Testing Tools: Ensure mobile app compatibility across devices (e.g., Robotium, Test IO).
    - GUI Testing Tools: Test the graphical user interface (e.g., Squish, EggPlant, AutoIT).
    - Bug Tracking Tools: Track, manage, and report defects (e.g., Trello, GitHub, JIRA).
    - Security Testing Tools: Detect vulnerabilities and safeguard applications (e.g., NetSparker, Vega, ImmuniWeb).

Top Software Testing Tools & Highlights

1. BrowserStack Test Management: Centralized test repository, integrates with CI/CD tools, supports automation frameworks.
2. LambdaTest: Cloud-based test execution, AI-powered, cross-browser and cross-device testing.
3. TestGrid: Scriptless automation, CI/CD integration, cross-browser and cross-device support.
4. QA Wolf: Combines automation and QA services, parallel execution, reduces flaky tests.
5. aqua cloud: AI-driven test management, automated test case generation, enhanced reporting.
6. TestLodge: Manual test management, integration with bug-tracking tools, structured test planning.
7. Selenium: Open-source web automation, multi-browser support, quick test execution.
8. Ranorex: GUI test automation for web, mobile, and desktop applications.
9. TestProject: Free, end-to-end automation for web, mobile, and API testing, supports codeless testing.
10. Katalon Platform: Comprehensive quality management, automated UI testing, easy integration and deployment.

Benefits of Automation Testing

Automation testing transforms software development by automating repetitive and complex testing tasks, offering significant advantages over manual testing. Key benefits include:

1. Cost Savings

- Automated tests reduce the need for extensive manual testing, saving labor costs.
- Test scripts can be **reused across multiple test cycles**, lowering the cost of developing new test cases.
- Parallel execution across devices and environments reduces testing time, minimizing overall project expenses.\

2. Faster Feedback and Development Cycles

- Automation provides **instant feedback** on application performance, enabling developers to address issues quickly.
- Integration with **CI/CD pipelines** ensures that tests run automatically after code changes, accelerating release cycles.
- Early detection of defects through a **shift-left approach** reduces rework and improves development efficiency.

3. Better Resource Allocation

- Human testers can focus on **exploratory testing and complex scenarios**, while automation handles repetitive and high-volume tasks.
- Teams can **strategically balance manual and automated testing**, increasing overall productivity.

4. Higher Accuracy and Reliability

- Automation eliminates **human errors** and ensures tests are performed **consistently and precisely**.
- Validates data with high accuracy, producing **reliable results** and dependable test outcomes.

5. Increased Test Coverage

- Automation enables testing across **multiple devices, browsers, operating systems, and configurations**, covering more scenarios than manual testing.
- Regression testing ensures that **new code changes do not break existing functionality**, enhancing software quality.

6. Early Bug Detection

- Automated testing identifies defects early in the development cycle, reducing **costly post-release fixes**.
- Helps teams maintain a **high-quality product** by preventing major issues from reaching production

7. Scalability and Performance Testing

- Automation can execute **large volumes of test cases simultaneously**, supporting large-scale applications.
- Load and performance tests simulate real-world user interactions, helping detect **bottlenecks and performance issues**.

8. Maximized ROI

- Saves time, effort, and costs in the long term by **reducing manual intervention**.
- Reusable test scripts ensure **efficient testing for future versions**, improving overall return on investment

9. Speed and Efficiency

- **Faster execution:** Automated tests run quicker than manual ones, reducing testing cycles.
- **Parallel execution:** Tests can run simultaneously on multiple platforms or devices.
- **Continuous testing:** Supports ongoing integration and deployment pipelines without delays

10. Improved Collaboration and Transparency

- Automated reporting provides **clear, detailed, and shareable results** for all stakeholders.
- Enhances communication between developers, testers, and managers.
- Facilitates tracking of testing progress, defects, and quality metrics.

Random Testing

Random testing, also known as monkey testing or ad hoc testing, is a black box testing technique where test cases are selected arbitrarily, and the outcomes are compared to verify whether the results are correct. In this approach, the system is tested by generating random and independent inputs to validate its behavior against expected outputs.

History:

- Random testing was first introduced by Melvin Breuer in 1971.
- Later, in 1975, it was evaluated by Vishwani Agrawal and Pratima to assess its effectiveness in software testing.

Key Points:

- Test cases are chosen randomly, without following any specific pattern or logic.
- It is primarily a black box assessment, focusing on system behavior rather than internal code structure.
- Random testing can help identify unexpected or rare errors that may not be captured through structured testing.
- There exist formulas and guidelines in literature for determining the number of tests, as well as success and failure rates for random testing.

Working of Software Random Testing

The working of the software random testing is listed below −

Step 1 − Determine the range of input data sets.

Step 2 − Select the input data arbitrarily from the range of input data sets.

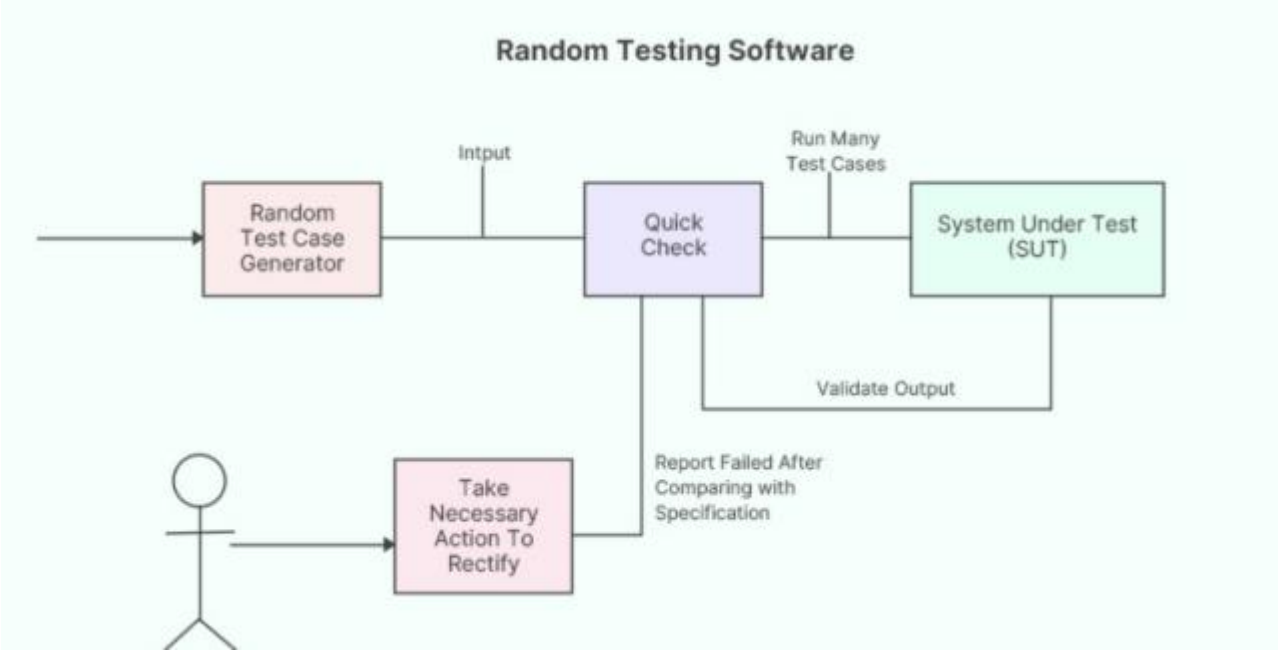
Step 3 − Verify the software with the help of the random data selected, and then create an unmethodical test set.

Step 4 − Analyze the test results, and compare them with the end user requirements.

Step 5 − On the events of failure, the corresponding defects are logged, and shared with the development team.

Step 6 − The defects once fixed, are again retested.

.



Types of Random Testing

1. Random input sequence generation: It is also known as Random Number Generator (RNG) in which a random sequential number or symbols is being generated which cannot be assumed during the random selection.

2. Random sequence of data inputs: In this, all the data are selected randomly for the inputs which can be used during the testing.

3. Random data selection from an existing database: The record where all the data are available from that record only the data can be selected for testing afterward no additional data cannot be added which are not available in the record.

Advantages of Random Testing

1. Simple to implement – Does not require detailed knowledge of the system or code.
2. Unbiased testing – Random inputs may reveal unexpected defects that structured testing might miss.
3. Effective for robustness testing – Can uncover rare or unusual bugs by testing unpredictable scenarios.
4. Supports stress testing – Can simulate high load or extreme usage conditions.
5. No prior test design required – Test cases are generated dynamically, reducing planning effort.

Disadvantages of Random Testing

1. No guarantee of complete coverage – Random tests may miss some important paths or conditions.
2. Inefficient for large systems – May require a huge number of tests to find significant defects.
3. Hard to reproduce defects – Random inputs can make it difficult to reproduce and debug errors.
4. Limited in scope – May not test specific functional requirements or critical scenarios.
5. Not suitable for complex logic testing – Fails to effectively test specific business rules or conditional flows.

Bug bashes and beta testing

Bug bashes and beta testing are both methods for finding software defects, but they differ in who performs the testing and when they occur. A bug bash is an internal, time-boxed event where an organization's employees "bash" or aggressively look for bugs through exploratory testing. Beta testing involves real, external users testing the software in a real-world environment before the final release to gather feedback and find any remaining issues.

Bug Bashes

A Bug Bash is a collaborative testing event where a team of developers, testers, product managers, and sometimes even stakeholders or users, intensively test a software application in a short period to identify as many bugs as possible.

- Focuses on finding high-impact and critical bugs quickly.
- Usually performed before major releases.
- Participants use the application in an exploratory way, trying unusual or edge-case scenarios.
- Helps improve team engagement and encourages diverse perspectives in testing.
- Bugs found are logged, prioritized, and fixed before release.

Advantages:

- Quick identification of critical defects.
- Encourages collaboration across teams.
- Helps discover issues that automated or routine tests might miss.

Disadvantages:

- Can be chaotic without proper coordination.
- Not a substitute for structured testing.
- Results depend on participants' experience and effort.

Beta Testing

Beta Testing is the final phase of testing conducted by real users outside the development team before the official release. Its goal is to validate the software in real-world conditions.

- Helps assess usability, functionality, and performance in actual user environments.
- Usually performed on a limited group of end users.
- Feedback is collected for bug fixes, improvements, and feature validation.
- Can be open beta (anyone can participate) or closed beta (invitation-only).

Advantages:

- Provides real-world feedback on software quality.
- Helps identify issues that internal testing may miss.
- Improves user satisfaction by incorporating early feedback.

Disadvantages:

- May expose the software to uncontrolled environments, risking negative impressions.
- Feedback may be inconsistent or incomplete.
- Fixing critical issues during beta can delay the release.

Test Planning

A test plan is a comprehensive document that outlines all testing-related activities for a project. It serves as a blueprint for testing, detailing what will be tested, how it will be tested, by whom, and the distribution of test types among testers. Test planning ensures that testing activities are organized, efficient, and aligned with project goals.

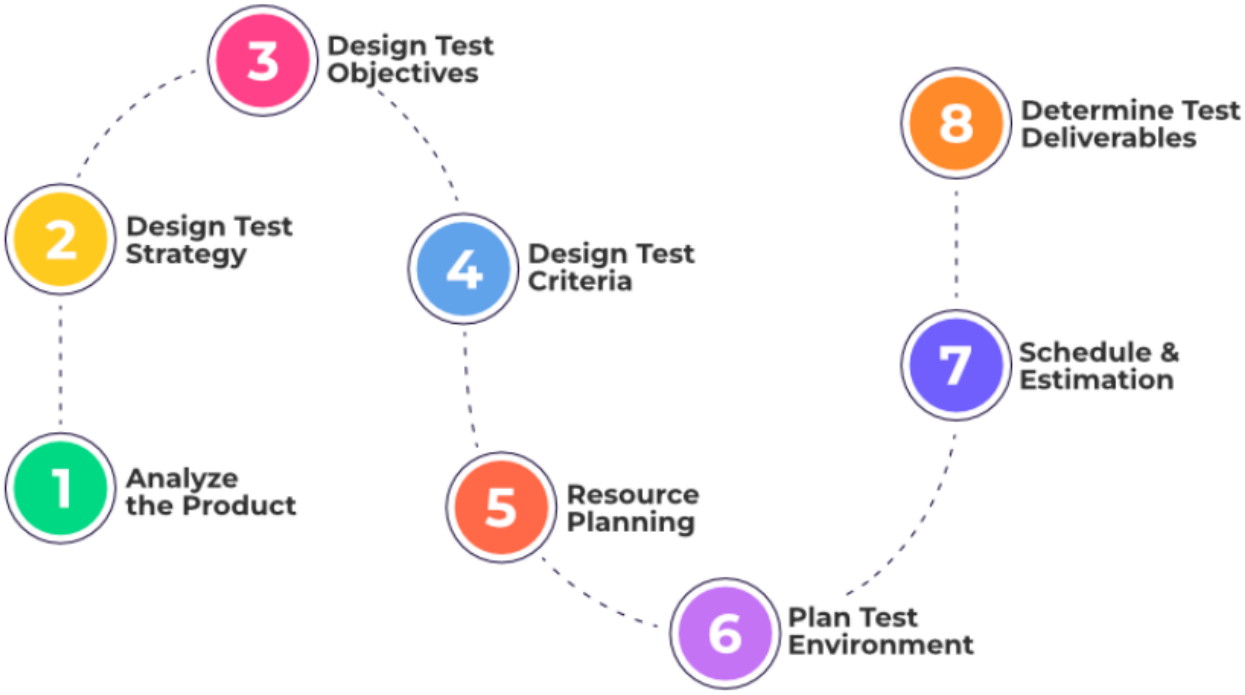
Purpose:

- Acts as a roadmap for the QA team.
- Coordinates testing activities among testers and other stakeholders, such as Business Analysts and Project Managers.
- Serves as a reference for monitoring progress, risks, and resource allocation.
- Helps maintain a current and adaptable plan as the project evolves.

Test Planning Process:

- Preparation: The test manager prepares the test plan before the testers are involved.
- Documentation: The plan includes objectives, scope, strategy, resources, schedule, risks, and success criteria.
- Execution & Coordination: It guides the QA team in conducting tests and ensures effective communication with stakeholders.
- Adaptation: The test plan is continuously updated according to project progress.

How to create a Test Plan?



Phases of Test Planning

1. Analyze the Product

- Understand the product by performing requirement analysis, walkthroughs, and interviews with clients, designers, and developers.
- Key focus areas:
    - Primary objectives and purpose of the product.
    - Intended users and target audience.
    - Hardware, software, and platform specifications.

2. Design the Test Strategy

- The Test Strategy is a high-level document defining how testing will be conducted.
- Key elements include:
    - Scope of Testing: Features/modules to be tested and those excluded.
    - Types of Testing: Functional, non-functional, regression, performance, security, etc.
    - Risk Analysis: Identify potential risks and mitigation plans.
    - Test Logistics: Assign responsibilities, define resource allocation, and tools to be used.

3. Define Test Objectives

- Establish clear goals for the testing process.
- Objectives include:
    - Features and functionalities to be validated (e.g., GUI, performance, workflows).
    - Expected outcomes and success criteria for each feature.
    - Compliance with standards, business requirements, and user expectations.

4. Define Test Criteria

- Suspension Criteria: Benchmarks that determine when testing should be paused due to issues such as critical defects or unstable builds.
- Exit Criteria: Benchmarks defining successful completion of testing, ensuring all test objectives and quality standards are met.
- Additional: Define pass/fail criteria for each test scenario for clarity in reporting.

5. Resource Planning

- Identify all resources required to execute the testing plan efficiently.
- Includes:
    - Human resources: QA engineers, test managers, developers for defect fixes.
    - Hardware & software: Devices, servers, operating systems, browsers.
    - Tools and infrastructure: Test management, defect tracking, automation tools.

6. Plan Test Environment

- Setup realistic testing environments that simulate actual user conditions.
- Ensure environments include:
    - Different OS versions, browsers, and network settings.
    - Required software configurations, databases, and third-party integrations.
    - Access to sandbox or staging servers for safe testing.
- Additional: Maintain environment documentation to ensure repeatability of tests.

7. Schedule and Estimation

- Break down the project into smaller, manageable tasks with estimated effort.
- Define timelines, milestones, and dependencies for each testing activity.
- Additional considerations:
    - Buffer time for defect fixing and retesting.
    - Parallel execution of tests where feasible to optimize schedule.
    - Integration with project management tools for tracking progress.

8. Determine Test Deliverables

- Identify all outputs required for successful testing:
    - Test cases, test scripts, test data, and automation scripts.
    - Test summary reports, defect logs, and metrics dashboards.
    - Test environment setup documents and configuration records.
- Ensure deliverables are updated, shared, and approved by stakeholders.

Why a Test Plan is Important

1. **Provides a Clear Roadmap**
    - Defines objectives, scope, strategy, and resources for testing.
    - Guides the QA team on what to test, how to test, and who will test.
2. **Ensures Proper Resource Allocation**
    - Helps identify human resources, hardware, software, and tools needed.
    - Ensures the right skills and resources are available when needed.
3. **Improves Communication**
    - Acts as a reference document for stakeholders like project managers, developers, and clients.
    - Clarifies responsibilities, schedules, and expectations for everyone involved.
4. **Reduces Risks and Prevents Delays**
    - Identifies potential risks, dependencies, and bottlenecks early.
    - Helps plan mitigation strategies to avoid delays or failures.
5. **Defines Success and Exit Criteria**
    - Clearly states pass/fail benchmarks, suspension criteria, and exit conditions.
    - Ensures everyone knows when testing is complete and successful.
6. **Facilitates Test Tracking and Control**
    - Provides a framework to monitor testing progress, track defects, and report metrics.
    - Helps evaluate quality and efficiency of the testing process.
7. **Supports Future Projects and Regression Testing**
    - Test plans can be reused or adapted for similar projects.
    - Helps in maintaining consistent testing standards over time.
8. **Increases Overall Software Quality**
    - Ensures systematic coverage of all features and requirements.
    - Reduces the likelihood of missed defects or incomplete testing.

Test Case

A Test Case is a detailed set of actions designed to verify whether a particular function or feature of a software application works as expected according to client requirements. It serves as a step-by-step guide for testers to validate the behavior of the application under specific conditions.

A test case outlines the steps to execute, the input data to use, and the expected outcomes to determine if the software performs correctly.

Software testers create test cases during the testing phase to ensure that each functionality meets the defined specifications.

Components:

- **Test Case ID:** A unique name or number to identify the test case.
- **Title/Description:** What the test case is meant to check.
- **Preconditions:** What needs to be set up before running the test?
- **Test Steps:** The actions to perform during the test.
- **Test Data:** The specific information needed for the test.
- **Expected Result:** What should happen if the software is working right?
- **Actual Result:** What happens when the test is run.
- **Status:** Whether the test passed or failed.
- **Postconditions:** What should be true after the test.

Why Do We Write Test Cases?

Test cases are essential in software testing as they ensure that the application functions as intended. The main reasons for writing test cases are:

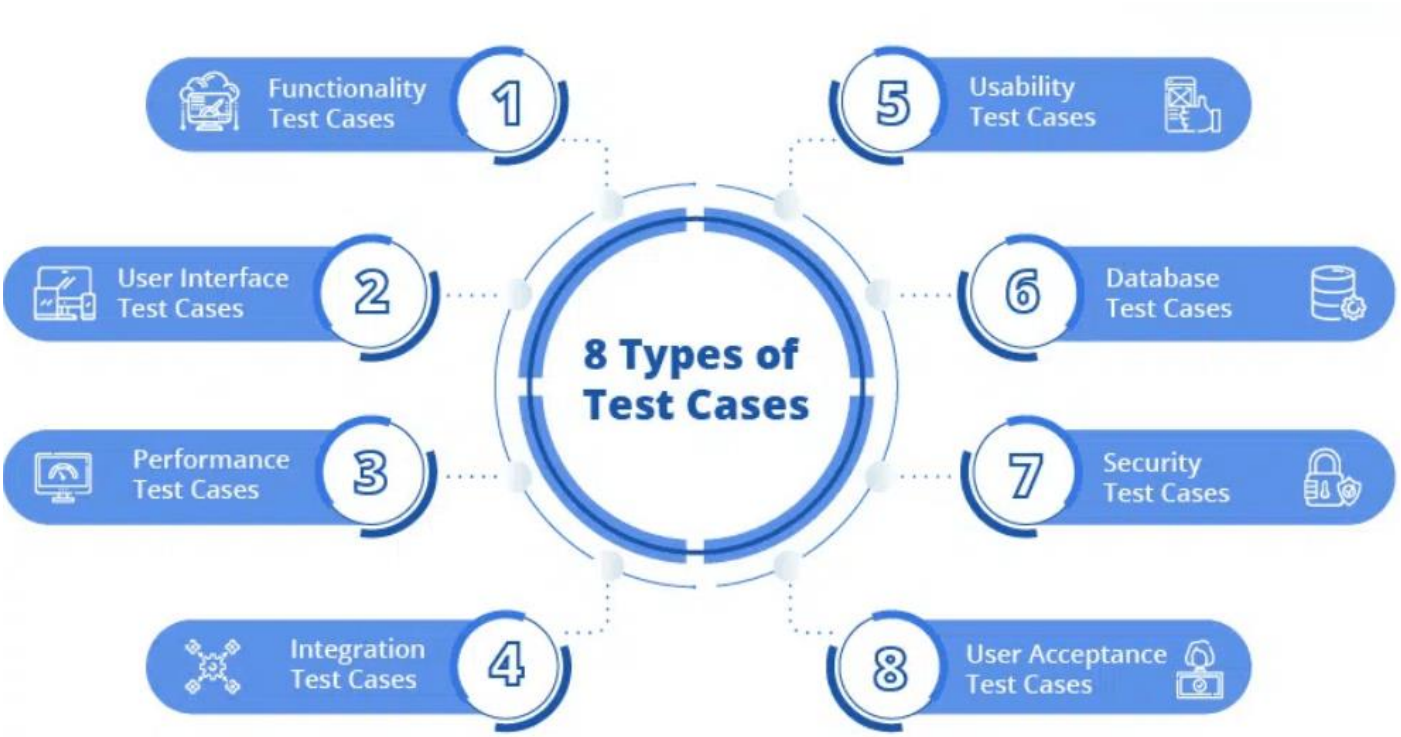
- To verify that each module or feature meets the specified requirements.
- To provide a clear, structured guide for test execution, including inputs and expected results.
- To support **regression testing** and future maintenance by serving as documentation.
- To ensure the software performs correctly under various conditions.
- To help identify gaps or defects early by analyzing all possible scenarios.

Test Case Format

Field	Description
Test Case ID	TC_001
Title	Check if the login works with the correct details
Description	Test if a user can log in with the right username and password.
Preconditions	The user must have a registered and active account.
Test Steps	1. Open the login page. 2. Enter the correct username. 3. Enter the correct password. 4. Click the login button.
Test Data	Username:testuser Password: password123
Expected Result	The user should be redirected to the main dashboard page.
Actual Result	[To be filled in after testing]
Status	[Pass/Fail]
Postconditions	The user is logged in and sees the dashboard.

When Do We Write a Test Case?

Test cases are typically written during the test planning phase, which is part of the software testing life cycle. When the client shares the business requirements, the developer will start software development, and it might take a few months to create this product.



---

1. Functionality Test Cases

These test cases verify whether the software's core functions work as expected. They ensure that each feature or module performs its intended operation according to the requirements.
Example:

- Checking if the login feature accepts valid credentials and rejects invalid ones.
- Verifying that the "Add to Cart" button adds items correctly in an e-commerce application.

Purpose: To validate the correctness and completeness of system functions.

2. User Interface (UI) Test Cases

UI test cases focus on the design, layout, and visual elements of the application. They ensure that all buttons, menus, icons, and labels are correctly placed, readable, and functional.

Example:

- Verifying that fonts, colors, and alignment are consistent across pages.
- Ensuring navigation menus and links redirect to the correct pages.

Purpose: To provide a seamless and visually appealing user experience.

3. Performance Test Cases

These test cases measure the system's responsiveness, stability, and scalability under various load conditions.

Example:

- Testing how fast a webpage loads under heavy traffic.
- Measuring system performance when multiple users log in simultaneously.

Purpose: To ensure the application performs efficiently even under stress or high usage.

4. Integration Test Cases

Integration test cases focus on verifying data flow and interaction between different modules or systems.
They ensure that components work together correctly once integrated.

Example:

- Checking data transfer between the front-end and database.
- Testing the interaction between payment and order modules.

Purpose: To validate seamless communication between connected parts of the software.

5. Usability Test Cases

Usability test cases evaluate how user-friendly and intuitive the application is. They test ease of navigation, accessibility, and clarity of instructions.

Example:

- Checking if a new user can easily complete a purchase without training.
- Ensuring buttons and menus have clear, descriptive labels.

Purpose: To improve user satisfaction and overall experience.

6. Database Test Cases

These test cases verify the correctness, integrity, and security of data stored in the database.

Example:

- Ensuring that data is correctly saved after a form submission.
- Checking for proper handling of database constraints, triggers, and relationships.

Purpose: To ensure data consistency, accuracy, and reliability in all operations.

7. Security Test Cases

Security test cases assess the system's ability to protect data and resist unauthorized access.

Example:

- Verifying that users cannot access restricted pages without login.
- Testing password encryption and secure data transmission.

Purpose: To safeguard sensitive information and maintain data privacy.

8. User Acceptance Test Cases (UAT)

These are end-to-end test cases designed to validate whether the software meets business requirements and is ready for deployment.

Example:

- Checking whether all key business workflows function as expected.
- Ensuring that the software fulfills client or user needs before release.

Purpose: To confirm that the final product is ready for real-world use.

Bug Life Cycle in Software Testing

The Bug Life Cycle (also known as the Defect Life Cycle) is the process that describes the journey of a software defect from its initial identification to its final closure. It provides a structured workflow for tracking and managing bugs effectively throughout the Software Development Life Cycle (SDLC).

In simple terms, it defines the various states a defect passes through — such as *New*, *Assigned*, *Open*, *Fixed*, *Retested*, and *Closed* — ensuring clear communication, accountability, and efficient defect resolution among testers and developers.

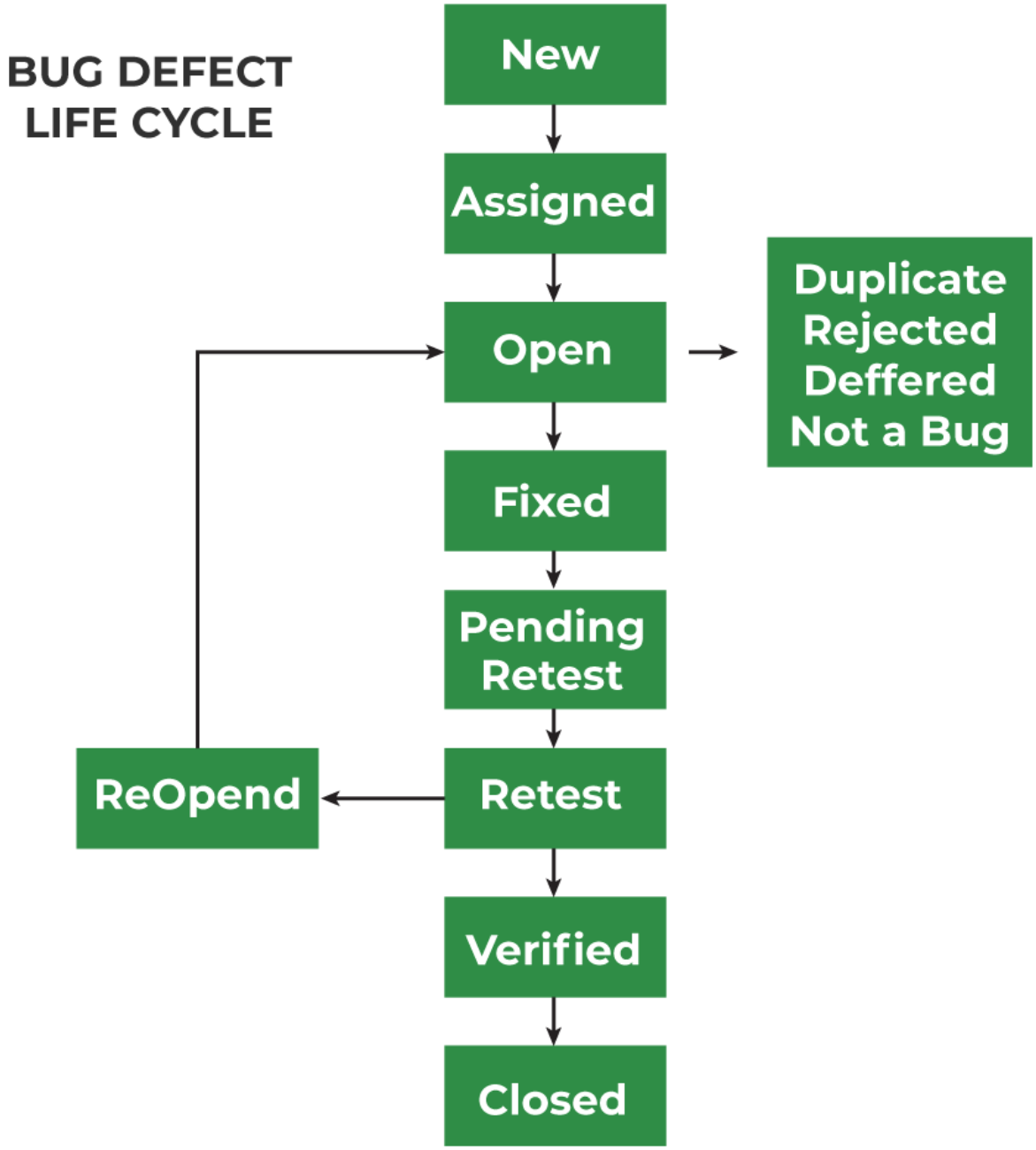
- The bug life cycle can vary from organization to organization and project to project, depending on testing methodologies, tools, and workflows used.
- The number of states a defect passes through differs based on the defect management tool (like Jira, Bugzilla, or Mantis).

- A well-defined bug life cycle helps ensure better quality control, reduces delays in fixing issues, and improves overall software reliability.

Objectives of Bug Life Cycle

- To provide a systematic process for identifying, reporting, tracking, and closing defects.
- To enhance communication and coordination between QA teams, developers, and project managers.
- To ensure no defect is missed or ignored during testing and development.
- To make the defect-fixing process more efficient by tracking the current status and progress of each bug.
- To maintain transparency in the testing process and ensure accountability.
- To support continuous quality improvement by documenting and analyzing recurring defect patterns.

Bug Lifecycle Process



The various stages of a bug lifecycle are:

1. New – A potential defect has been reported and is awaiting validation.
2. Assigned – The defect is assigned to the development team but has not been addressed yet.
3. Active – The developer is investigating and working on the defect. At this stage, it may be postponed for a future release or considered invalid or not a defect.
4. Test – The defect has been fixed and is ready for testing.
5. Verified – QA has retested the fix and confirmed the issue is resolved.

6. Closed – The defect is considered resolved and closed after successful verification or if marked as a duplicate or invalid.
7. Reopened – If the issue persists after a fix, QA reopens the defect for further investigation.

Example of Bug Life Cycle

The bug life cycle enables development teams to systematically track and resolve defects, ensuring that the software meets functional and quality standards before release. Below is an example of the E-Commerce Website Bug Life Cycle, demonstrating how each phase works in a step-by-step process for better understanding and management.

Step 1: New

While testing the search functionality on the e-commerce website, you, as the tester, notice that the search results page is blank when trying to search for a product. You document this issue by creating a defect report that includes details of the problem and the steps to reproduce it. Once the defect is reported, it enters the New state, indicating that the issue has been logged and is awaiting review.

Step 2: Assigned

After the defect is logged, the development team reviews it. Upon assessing the issue, they assign the defect to a developer for further investigation. At this point, the defect status changes to Assigned, meaning it is now the developer's responsibility to address the problem.

Step 3: Open

The developer begins investigating the issue and identifies that the root cause is a bug in the backend code, specifically related to how search queries are processed. The status of the defect is updated to Open, indicating that the developer is actively working to fix the problem

Step 4: Fixed

Once the developer resolves the issue by correcting the faulty backend code, the defect is marked as Fixed. The developer has made the necessary changes, and the fix is now ready for testing.

Step 5: Pending Retest

After fixing the bug, the developer submits the updated code for retesting. The status changes to Pending Retest, as the code now needs to be tested to ensure that the defect has been successfully addressed and that the functionality works as expected.

Step 6: Retest

As the tester, receive the updated code and retest the search functionality. After running the test, you find that the search results page now displays the correct products, confirming that the issue has been fixed. The defect's status is updated to Retest, indicating that the tester is validating the fix.

Step 7: Verified

Since the issue is no longer present after retesting and the functionality works as expected, you confirm that the bug is indeed resolved. The defect is then marked as Verified, showing that the tester has validated the fix and the bug is no longer an issue.

Step 8: Closed

After confirming that the defect is fully resolved and no further issues are present, you close the defect. The status is updated to Closed, indicating that the bug is resolved, and the life cycle is complete.

UNIT-4

Software Quality Assurance (SQA)

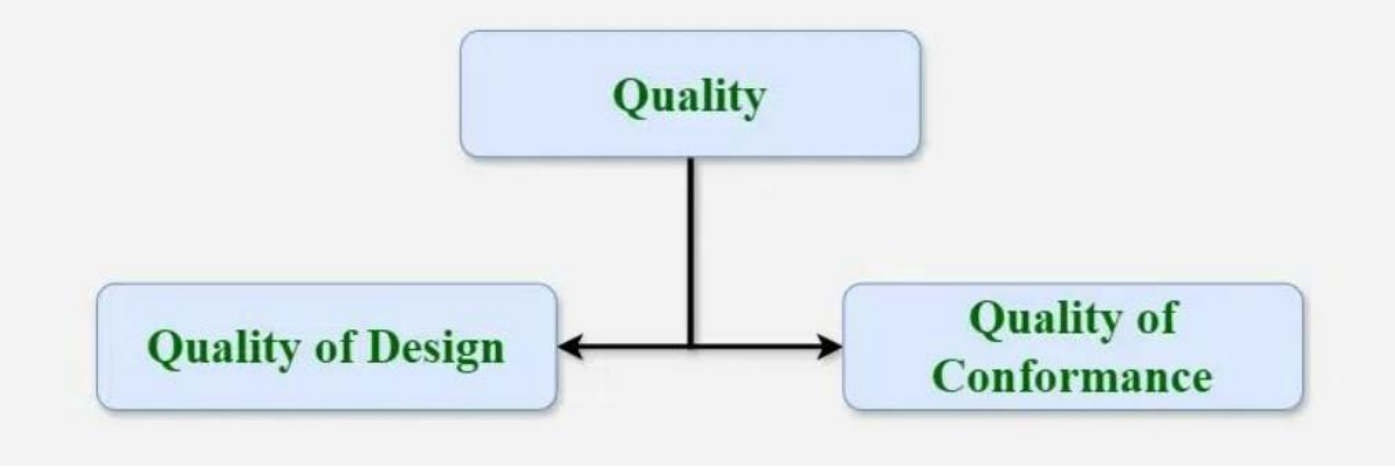
Software Quality Assurance (SQA) is a critical component of the software development process that ensures the quality and reliability of software products. It encompasses a set of methodologies, practices, and activities designed to monitor and improve the software engineering processes and methods used to ensure quality.

SQA focuses on preventing defects rather than detecting them after they occur. It ensures that the processes, procedures, and standards followed during development are suitable for the project and are implemented correctly.

It runs parallel to the software development process and acts as an umbrella activity applied throughout the entire software lifecycle. The primary goal of SQA is to improve the development process so that potential problems can be prevented before they become major issues.

The key components of SQE include:

- **Software Quality Assurance (SQA):** Ensures that all processes, standards, and procedures are properly defined and followed.
- **Quality Control (QC):** Focuses on identifying and fixing defects in the product through testing and inspection activities.



1. Quality

- Quality is the degree to which a product, service, or process meets customer expectations and specified requirements. It represents how well the final output performs its intended function without defects.
- Software Quality in Engineering refers to the capability of a software product to meet specified requirements and satisfy user needs. It includes key attributes such as reliability, usability, efficiency, and maintainability.
- It is achieved through Software Quality Engineering (SQE) — a systematic approach that integrates quality checks throughout the entire software development lifecycle. The main objective is to prevent defects rather than merely detecting them after development.

2. Quality of Design

- Quality of Design refers to how well the requirements and specifications of a product are defined during the design phase.
- It represents the intended level of quality that the designers and engineers plan to achieve.
- The focus is on meeting customer needs and expectations through a good design.
- A better design quality ensures that the product is capable of performing its intended functions effectively.

Example:
Designing a mobile app with an easy-to-use interface, high security, and fast performance reflects high design quality.

3. Quality of Conformance

- Quality of Conformance refers to how well the actual product or process conforms to the intended design and specifications during development and production.
- It focuses on following the defined standards, processes, and guidelines correctly.
- Poor conformance quality results in defects, even if the design quality is high.

Example:
If developers correctly implement all the features as per the app's design without bugs or performance issues, the product has high conformance quality.

Key Aspects of Quality

1. **Customer Satisfaction:**
   It should fulfill the user's needs and expectations.
2. **Reliability:**
   The product should perform consistently over time.
3. **Efficiency:**
   It should use resources (time, memory, etc.) effectively.
4. **Maintainability:**
   It should be easy to update, modify, or correct.

Quality in Software Testing (ST) and Quality Assurance (QA)

1. Quality in Software Testing (ST):

- Focuses on **identifying defects** in the software after it has been developed.
- Ensures that the **final product meets the specified requirements** and works as expected.
- The goal is to **verify and validate** the software to make sure it is free from errors, bugs, or performance issues.
- Quality in testing is **product-oriented**, as it deals with the quality of the final deliverable.

Example:
Testing a login page to ensure it correctly authenticates valid users and rejects invalid ones.

2. Quality in Quality Assurance (QA):

- Focuses on **improving and maintaining processes** to prevent defects during software development.
- Ensures that the methods, standards, and processes used in software development are followed correctly.
- The goal is to build quality into the process itself, rather than checking it after development.
- Quality in QA is process-oriented, as it deals with how the software is developed.

Example:
Implementing a code review process or defining coding standards to ensure error-free development.

Testing and Quality Assurance at Workplace

- **Testing**: The process of evaluating a software application to ensure it meets specified requirements, is free from defects, and performs reliably under various conditions. It involves executing the software and checking for bugs, errors, or gaps between expected and actual behavior.
- **Quality Assurance (QA)**: A broader, proactive approach focused on preventing defects through process improvement, standards adherence, and continuous monitoring. QA ensures the entire software development lifecycle (SDLC) follows best practices to deliver high-quality products.
- **Key Difference**:
    - Testing: Reactive (finds and fixes issues after development).
    - QA: Proactive (prevents issues by improving processes).

Importance of QA and Testing in the Workplace

- Builds **customer trust and satisfaction**.
- Reduces **costs and rework** by preventing and identifying issues early.
- Promotes **team accountability and discipline**.
- Ensures **compliance** with industry standards (like ISO or CMMI).
- Enhances the organization's **reputation and competitiveness**.

Importance at the Workplace

- **Business Benefits**:
    - Reduces downtime and production costs: A single major bug can cost millions (e.g., in banking apps).
    - Enhances reputation: High-quality software leads to positive reviews and repeat business.
    - Compliance: Meets standards like ISO 9001, GDPR, or industry-specific regulations (e.g., HIPAA for healthcare apps).
- **For Teams/Developers**:
    - Early defect detection: 80% of defects are introduced in requirements/design; testing catches them before deployment.
    - Collaboration: Bridges developers, testers, and stakeholders.
    - Career Relevance for MCA Students: QA roles are entry-level opportunities; skills in automation testing boost employability (e.g., in companies like Infosys, TCS).

How They Work Together

1. **QA defines the standards**: and processes to ensure quality.
2. **Testing executes these processes**: by applying various tests to evaluate the product's quality against these standards.
3. **The results from testing**: inform QA on whether the established processes are effective and if further improvements are needed.

By combining QA and testing, workplaces can proactively prevent defects, deliver reliable products that meet customer needs, build user trust, and ultimately enhance their brand reputation and profitability.

Test Management and Organizational Structure

Test Management

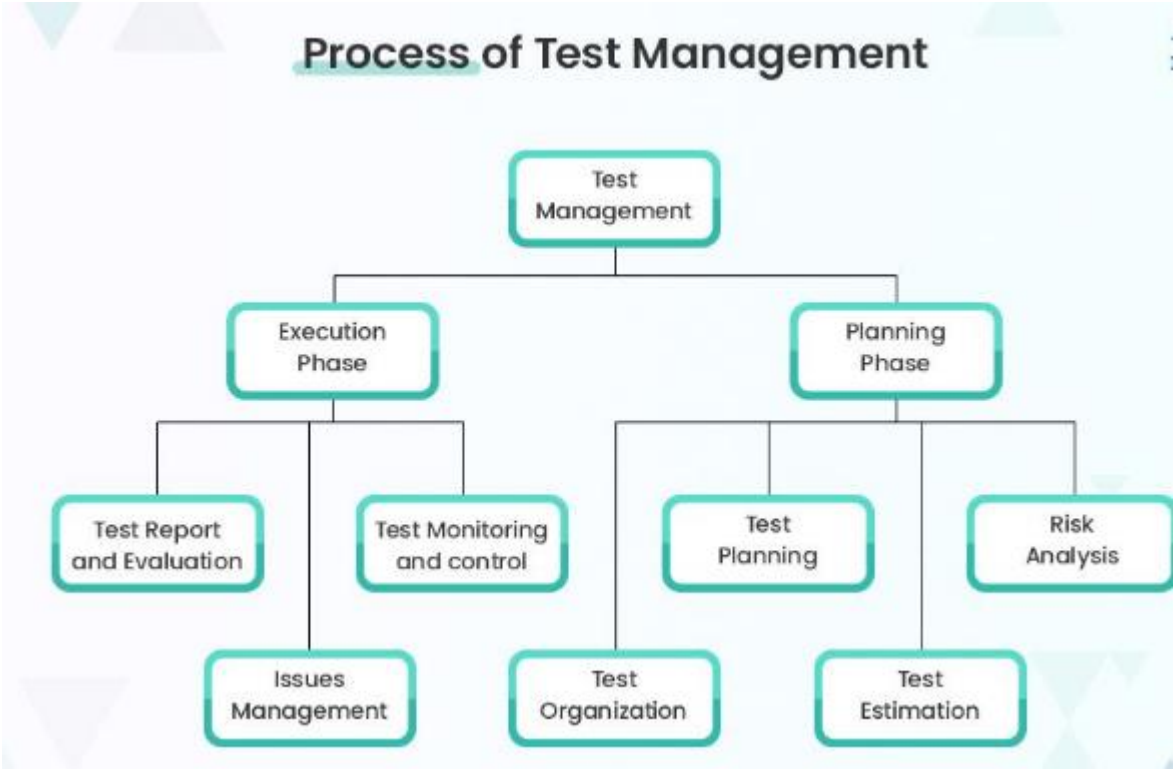
Test Management is the process of planning, organizing, controlling, and monitoring all testing activities in a software project to ensure that the final product meets quality standards and requirements.

It involves managing:

- Resources: Testers, tools, environments, and budgets.
- Processes: Test planning, execution, defect tracking, and reporting.
- People: Roles and responsibilities of QA teams, test engineers, and leads.
- Tools: Software for tracking test cases, defects, and progress (e.g., JIRA, TestRail).

Key Objectives

- Ensures testing is systematic and efficient.
- Helps achieve complete coverage of requirements.
- Tracks defects, risks, and test progress throughout the project.
- Improves communication and collaboration among QA, developers, and management.
- Enables better decision-making using metrics and reports.

- 

Key Activities in Test Management:

1. **Test Planning:**
    - Define objectives, scope, strategy, and resources.
    - Prepare a Test Plan outlining schedules, responsibilities, and entry/exit criteria.
2. **Test Design and Preparation:**
    - Develop test cases, scripts, and test data.
    - Select techniques like Equivalence Partitioning and Decision Table Testing for efficiency.
    - Set up environments and decide on automation where applicable.
3. **Test Execution and Monitoring:**
    - Execute manual or automated tests.
    - Log results, defects, and track progress using test metrics like coverage and defect density.
4. **Defect Management:**
    - Identify, prioritize, and track defects until closure.
    - Assign severity and priority to defects to manage resolution efficiently.

5. **Test Reporting and Closure:**
    - Generate summary reports, dashboards, and lessons learned.
    - Archive all test artifacts for future audits and compliance.

Test Management Tools

Common tools used in the workplace for managing testing activities include:

- Testsigma
- JIRA / Azure DevOps
- TestRail
- HP ALM (Application Lifecycle Management)
- Zephyr
- Bugzilla

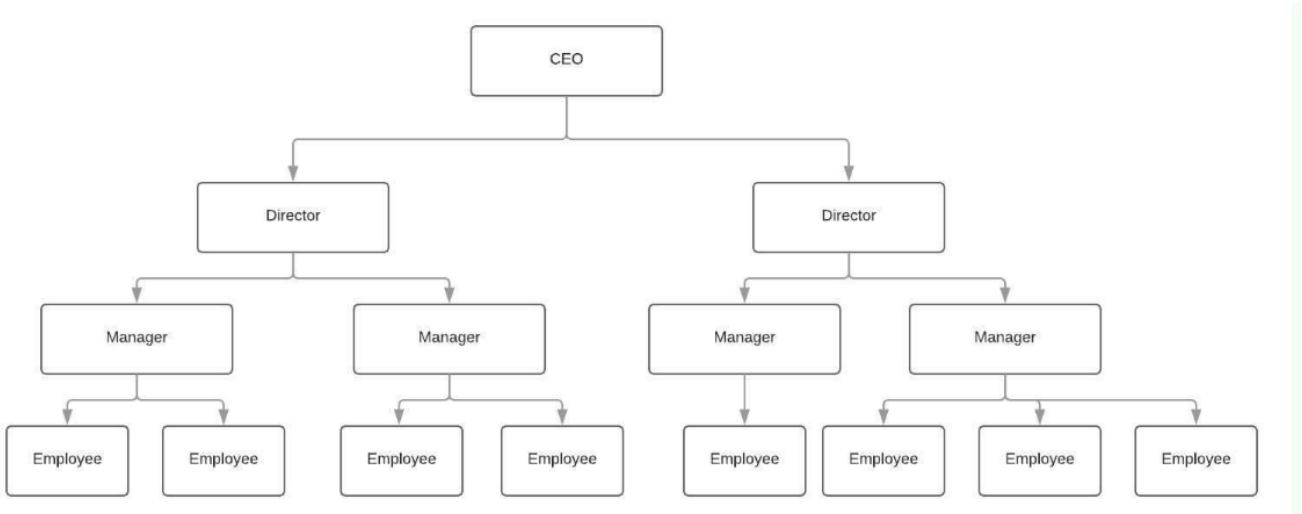
These tools help in planning, tracking, defect management, and reporting efficiently.

Organizational Structure

Organizational Structure refers to the way a company or team is arranged to define roles, responsibilities, reporting lines, and communication channels.

In the context of software testing and QA, it defines:

- Who does what: Assigning tasks to managers, leads, testers, and specialists.
- Who reports to whom: Clear reporting hierarchy to ensure accountability.
- How teams communicate and coordinate: Facilitates collaboration between QA, developers, and management.



Typical Roles:

1. **QA/Test Manager:**
    - Oversees all testing activities, allocates resources, manages schedules, and ensures standards.
    - Acts as a bridge between development and management.
2. **Test Lead / Coordinator:**
    - Manages day-to-day operations, assigns tasks, and monitors progress.
3. **Test Engineers / QA Analysts:**
    - Design and execute test cases, report and retest defects.
4. **Automation Testers:**
    - Develop automated scripts and maintain testing tools.
5. **Performance / Security Testers:**
    - Focus on performance, load, and security testing.
6. **Configuration Manager:**
    - Maintains test environments and controls versions of test artifacts.

Software Quality Assurance Metrics

SQA Metrics **are** quantitative measures used to assess the quality of software and the effectiveness of the software development and testing process**.** They help in monitoring, controlling, and improving software quality throughout the development lifecycle.

Metrics provide **objective data** for decision-making, defect prevention, and process improvement.

- **Purpose**:
    - **Monitor Quality**: Track if software meets requirements (e.g., <1% defect rate in production).
    - **Drive Improvements**: Use data for retrospectives (e.g., reduce testing time by 20%).
    - **Decision-Making**: Help managers allocate resources (e.g., prioritize high-defect modules).
    - **Benchmarking**: Compare against industry standards (e.g., defect density <0.5 per KLOC – thousand lines of code).

Why QA Metrics Matter

- **Track Progress:**

They provide data points to monitor the effectiveness of quality assurance efforts and track progress over time.

- **Improve Quality:**

Metrics help identify areas of the software or process that are underperforming, allowing teams to focus their efforts on improvement.

- **Inform Decisions:**

QA metrics provide insights that support strategic decisions regarding test strategies, resource allocation, and release readiness.

- **Measure Performance:**

They offer objective data for evaluating the overall health and performance of the software development lifecycle (SDLC).

- **Enhance Customer Experience:**

By ensuring software reliability and functionality, metrics help align the product with user expectations and satisfaction.

Six Sigma

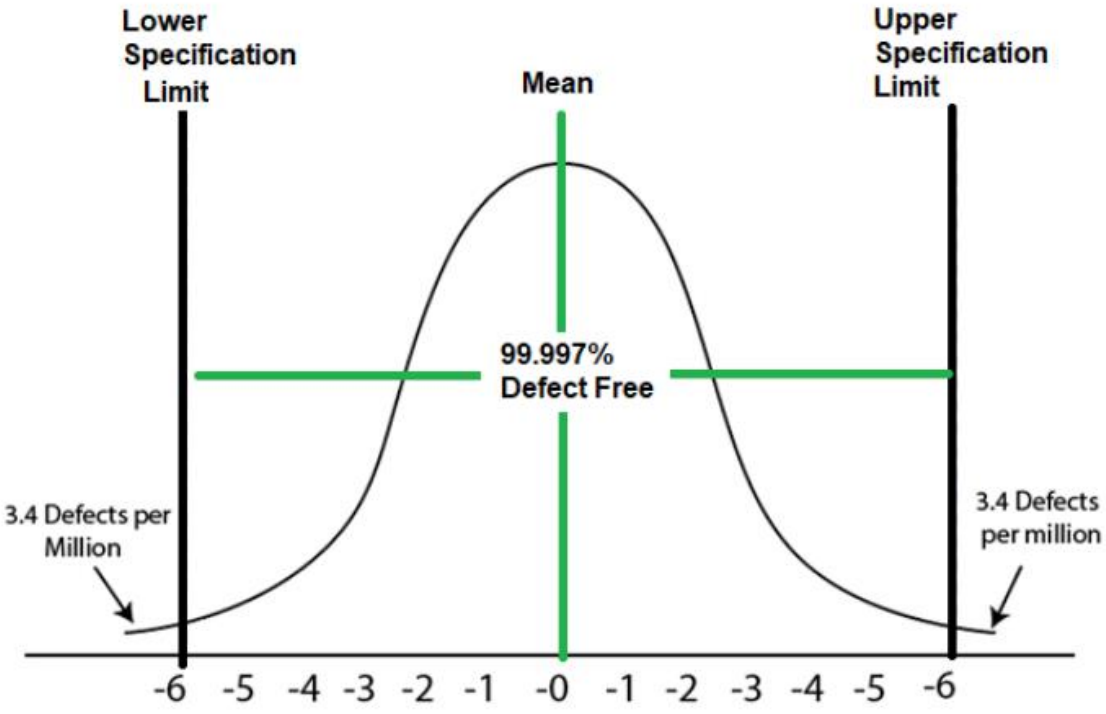
Six Sigma is a data-driven methodology used to improve business and software processes by reducing errors, minimizing variations, and enhancing overall quality and efficiency. It aims to achieve near-perfect performance, targeting no more than 3.4 defects per million opportunities (DPMO) — which means 99.99966% defect-free output.

Six Sigma focuses on process improvement through statistical analysis and emphasizes customer satisfaction by ensuring that products and services consistently meet expectations.

Six Sigma defines and reduces the variation in any process that leads to defects or inefficiency.

It works in two major phases:

1. Identification – Detecting the root cause of process defects.
2. Elimination – Removing the causes of defects to achieve consistency and quality improvement.



Six Sigma Curve

The curve you see is a bell curve (normal distribution) that represents the variation in any process — for example, how consistent your software testing results or production output are.

- The centre line (Mean) is the average performance.

- The spread (width) of the curve shows how much variation there is from the mean.

Sigma (σ) Concept

- The term "sigma (σ)" means standard deviation, a measure of how much variation exists from the mean.
- In Six Sigma, each sigma level represents how far a process result can be from the mean before it is considered a defect.

The Six Sigma Principle

- Six Sigma means the process variation is within six standard deviations (±6σ) from the mean.
- This ensures that almost all outputs (99.9997%) fall within acceptable limits.

So, the Lower Specification Limit (LSL) and Upper Specification Limit (USL) mark the boundaries within which the process output is acceptable.

Numbers Mean

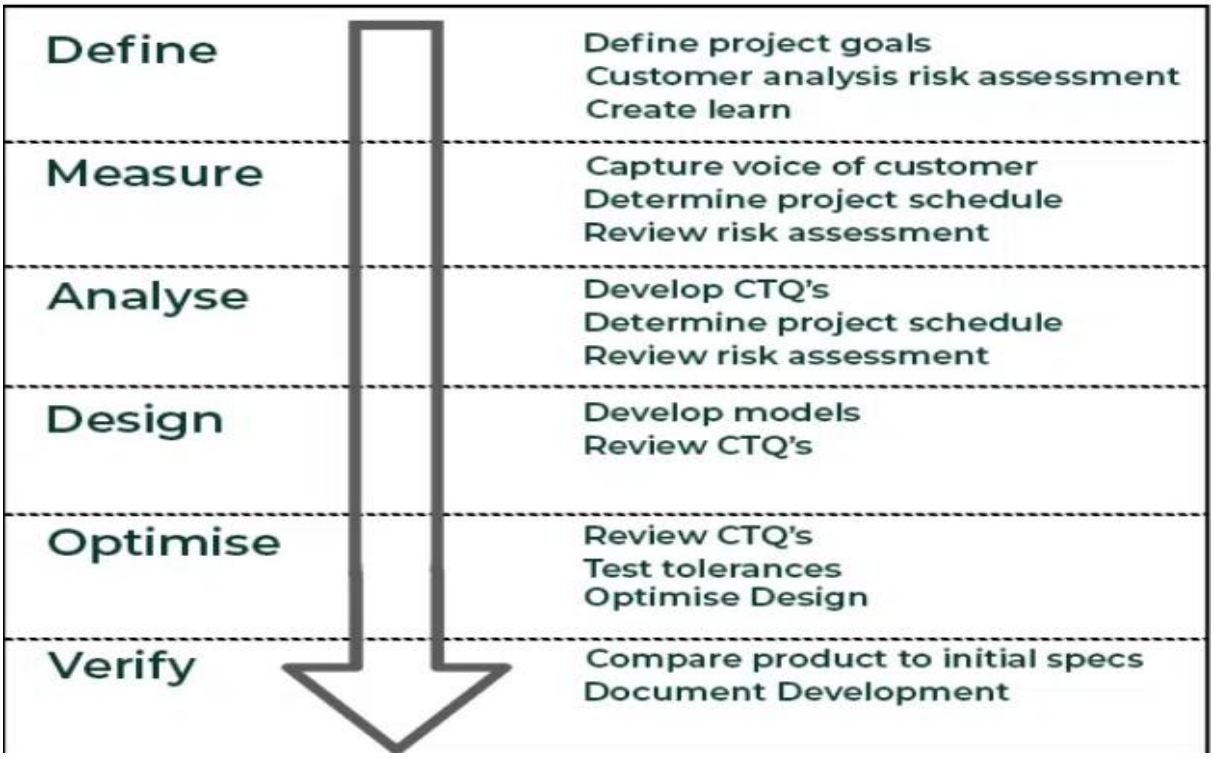
- Between -6σ and +6σ, the process produces results that are considered defect-free.
- Only a tiny fraction of results (3.4 defects per million opportunities) fall outside these limits.

That's why Six Sigma quality is considered world-class performance — the process is 99.9997% defect-free.

Key Points on the Graph

Term	Meaning
Mean (μ)	The average or center point of the process.
±6 Sigma Limits	Boundaries showing the extreme ends of acceptable variation.
Lower Specification Limit (LSL)	Minimum acceptable performance or quality.
Upper Specification Limit (USL)	Maximum acceptable performance or quality.
3.4 Defects per Million	Indicates extremely high process quality (near perfection).

Methodology



DMADV – Six Sigma Design Framework

1. Define

- Establish project goals aligned with customer needs and business objectives.
- Conduct customer analysis and risk assessment to understand expectations and potential challenges.
- Create a learning plan for the project team to understand the process scope and objectives.

Purpose: To clearly define what success looks like and set measurable targets.

2. Measure

- Capture the voice of the customer (VoC) to identify key requirements and expectations.
- Determine the project schedule, milestones, and resource needs.
- Review risks that might affect timelines, cost, or quality.

Purpose: To collect data that defines current capabilities and customer needs.

3. Analyse

- Develop **CTQs (Critical to Quality)** — parameters that directly impact customer satisfaction.
- Reassess the **project plan and risk factors** based on data collected.
- Identify **gaps or variations** in the process that can affect quality.

Purpose: To find root causes and understand what drives process performance.

4. Design

- **Develop models** or prototypes that meet CTQ requirements.
- Review and refine designs to ensure they align with customer expectations.

Purpose: To design a solution or product that eliminates identified issues.

5. Optimise

- Review CTQs and **test tolerances** to ensure the process or product performs efficiently.
- **Optimize the design** for performance, cost, and reliability.

Purpose: To fine-tune the process for maximum efficiency and minimal variation.

6. Verify

- **Compare the final product** or process output against the initial specifications.
- **Document the development** process and verify that all requirements are met.

Purpose: To validate that the new process achieves Six Sigma quality standards before full-scale implementation.

The 6 Key Principals of Six Sigma

1. Customer-Focused Improvement

The main idea behind Six Sigma is to keep the customer at the center of everything. It focuses on understanding what customers truly need through the Voice of the Customer (VoC). Once the organization knows what the customer expects, it works on improving its processes to meet or exceed those expectations. This leads to better customer satisfaction, loyalty, and ultimately, higher profits.

2. Continuous Process Improvement

Six Sigma is not a one-time activity; it’s a continuous journey. Organizations using this method always look for areas where they can make improvements. After one process is

enhanced, they move to another. The goal is to keep refining operations to reach a high accuracy level (up to 99.99966%) while maintaining efficiency and financial stability.

3. Reducing Process Variation

Every process naturally has some variation, which can lead to mistakes and defects. Six Sigma aims to reduce these variations to ensure consistency and quality. For example, in a software team, developers may have different coding habits. By following coding standards, doing code reviews, and using automated testing, the team can reduce variation and make the final product more reliable.

4. Eliminating Waste

Six Sigma encourages organizations to identify and remove unnecessary steps or resources that do not add value to the final product or service. Waste could be in the form of extra materials, time delays, or repetitive tasks. By eliminating waste, the organization can save time, reduce costs, and improve productivity.

5. Employee Empowerment

Successful Six Sigma implementation depends on involving and empowering employees. Workers who handle the process daily are trained and given tools to maintain improvements. Usually, a team of managers and experts first plans the improvements, and then employees are encouraged to take ownership and sustain those changes in their regular work.

6. Process Control and Monitoring

After improvements are made, it's important to ensure that processes stay under control. Six Sigma uses data, measurements, and statistical tools to monitor performance and maintain consistency. Regular checks and employee training help prevent the process from slipping back into an unstable state, ensuring long-term success.

Six Sigma is a structured methodology used by organization to improve processes by reducing inherit variation and defects. Six Sigma helps the organization in improving the efficiency, quality and customer satisfaction by reducing variation and defects in processes.

CMM (Capability Maturity Model)

The Capability Maturity Model (CMM) is a framework designed to help organizations evaluate and improve their software development processes. It was created by the Software Engineering Institute (SEI) at Carnegie Mellon University in 1987. Rather than being a software development model itself, CMM serves as a guideline for process improvement that enables organizations to produce high-quality software consistently.

CMM helps companies understand how well their current processes are defined, managed, and optimized. It provides a step-by-step pathway for improving efficiency, productivity, and product quality.

Key Benefits of Implementing Capability Maturity Model (CMM)

The Capability Maturity Model (CMM) plays a vital role in improving the quality and efficiency of software development within an organization. It provides a structured approach to assess how well the software processes are managed and how they can be enhanced to achieve better performance and reliability.

1. Improves Process Quality

CMM helps organizations define and standardize their development processes. This ensures consistency, reduces errors, and improves the overall quality of the software product.

2. Enhances Project Management

By following CMM guidelines, organizations can plan, monitor, and control projects more effectively. It enables better estimation of cost, time, and resources, leading to successful project delivery.

3. Promotes Continuous Improvement

The model encourages organizations to continuously review and improve their processes. As they move through the maturity levels, they adopt new methods and technologies to enhance performance and efficiency.

4. Increases Customer Satisfaction

High-quality and reliable software products built using mature processes lead to greater customer trust and satisfaction. This strengthens the organization's reputation and long-term relationships with clients.

Principles of Capability Maturity Model (CMM)

1. People as a Key Competitive Asset

In software development, competition arises when multiple organizations perform similar tasks. The true competitive advantage comes from the skills, knowledge, and efficiency of the people working in the organization.

2. Alignment with Business Objectives

The capabilities of individuals and teams should align with the organization's business goals. Every effort to improve skills and processes must directly contribute to achieving these objectives.

3. Investment in People's Development

An organization must continuously invest in building the knowledge, skills, and capabilities of its employees. Skilled and motivated people are essential for maintaining quality and ensuring long-term success.

4. Management Responsibility

It is the responsibility of management to create an environment that supports growth, provides resources for learning, and encourages continuous improvement among employees.

5. Improvement as a Continuous Process

Enhancing people's capabilities should be treated as an ongoing process. This involves following proper procedures, adopting best practices, and reviewing progress regularly to ensure consistent development.

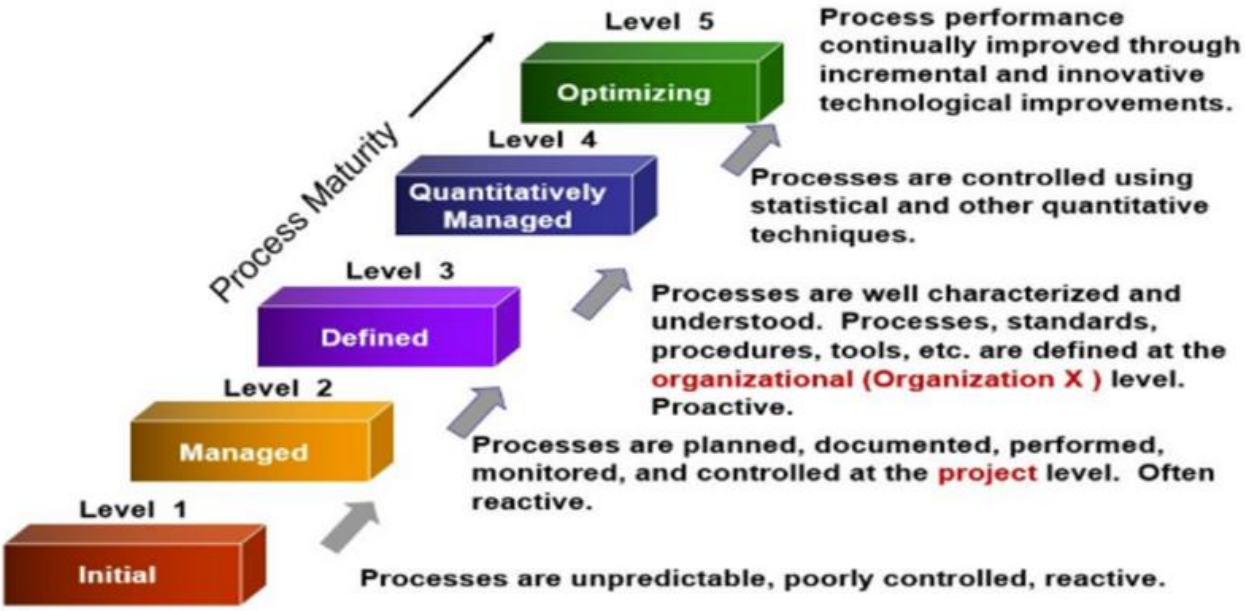
6. Providing Opportunities for Growth

Organizations should actively create opportunities for employees to learn, experiment, and improve. This empowers people to apply their knowledge and contribute more effectively.

7. Adapting to Change and Innovation

With new technologies and practices emerging rapidly, organizations must stay updated and flexible. Continuous improvement and skill development help them remain competitive and innovative.

Levels of Capability Maturity Model



Level 1: Initial

At this stage, processes are ad hoc and undefined, with no KPIs or formal project management. Development heavily relies on individual skills, leading to an unstable environment. Communication, coordination, and training are minimal, tools and automation are rarely used, and projects are at high risk of failure or delays.

Level 2: Repeatable

Basic project management policies are established, leveraging experience from previous projects. Key practices include project planning (defining resources, goals, and constraints), configuration management, requirements management (incorporating customer feedback), subcontract management, and software quality assurance.

Level 3: Defined

Processes are documented and standardized across the organization. Practices include peer reviews (walkthroughs, inspections), intergroup coordination, organization-wide process definition and focus, and structured training programs to enhance team skills and efficiency.

Level 4: Managed

Quantitative goals are set for both software processes and products. Measurement and analysis allow prediction and control of product and process quality. Key aspects include software quality management and quantitative project performance management.

Level 5: Optimizing

The organization focuses on continuous process improvement using quantitative feedback. Emphasis is on process and technology change management, adoption of new tools and techniques, and defect prevention to enhance quality, productivity, and reduce development time.

ISO 9000

The ISO 9000 model in software engineering is a family of international quality management standards that provides a framework for establishing and maintaining an effective Quality Management System (QMS). It is not a single model but a set of guidelines to ensure consistent quality, customer satisfaction, and continuous process improvement.

Key Points:

1. **Application in Software Engineering:**
    - Software companies can apply ISO 9001 to document, implement, and improve their processes.
    - It helps in managing the entire software lifecycle, including development, supply, maintenance, and support.

- Focuses on process quality rather than the product itself, ensuring repeatable and reliable outcomes.

2. **Purpose and Scope:**
   - ISO 9000 standards aim to satisfy customers, comply with regulatory requirements, and achieve continuous improvement.
   - Emphasizes both operational and organizational aspects, including responsibilities, reporting structures, and accountability.
   - Serves as a formal contract between independent parties, providing confidence in the quality of services and deliverables.
3. **Principles of ISO 9000 in Software:**
   - **Customer Focus:** Meeting or exceeding customer expectations.
   - **Leadership Commitment:** Management must actively support quality initiatives.
   - **Process Approach:** Viewing software development as interconnected processes to enhance efficiency.
   - **Continuous Improvement:** Regularly improving processes, tools, and practices.
   - **Evidence-Based Decision Making:** Decisions are driven by data and measurable outcomes.
   - **Supplier and Stakeholder Management:** Ensures external inputs meet quality standards.
4. **Documentation and Standardization:**
   - Requires well-documented processes for design, development, testing, and maintenance.
   - Encourages use of metrics and records for quality audits, tracking defects, and performance evaluation.
5. **Benefits for Software Organizations:**
   - Improved software process reliability and repeatability.
   - Higher customer confidence and satisfaction.
   - Better regulatory compliance and risk management.
   - Streamlined workflows and reduced errors through standardized procedures.
   - Enhanced organizational credibility and competitive advantage.
6. **Integration with Other Models:**

- ISO 9000 can be used alongside models like CMM/CMMI to combine process maturity with formal quality management.
- Supports software process improvement initiatives and structured project management.

Why ISO Certification is Needed in Software Industry

1. **International Recognition:** Helps software companies participate in global projects and bidding.
2. **High-Quality Software:** Ensures software is reliable and can be developed repeatedly with good quality.
3. **Proper Documentation:** Encourages recording processes, making projects easier to manage and track.
4. **Better Processes:** Helps design efficient workflows and measure software quality.
5. **Customer Confidence:** Shows clients that the company follows quality standards.

How to Get ISO 9000 Certification

1. **Application:**
   - The organization decides to obtain ISO 9000 certification and submits an application to an ISO registrar or certification body.
   - The application typically includes basic details about the organization, its processes, and objectives.
2. **Pre-Assessment:**

- The registrar performs an initial evaluation of the organization's quality management system (QMS).
- This helps identify gaps, areas of improvement, and readiness for formal certification.

3. **Document Review and Adequacy Check:**
    - The registrar reviews all submitted documents, including process manuals, quality procedures, and records.
    - Suggestions are given to improve documentation or align it with ISO 9000 standards.
4. **Compliance Audit:**
    - The registrar conducts a detailed audit to verify whether the organization has implemented the recommended changes and is following the documented processes.
    - This ensures that the QMS is effectively operating as per ISO standards.
5. **Registration / Certification:**
    - If the organization successfully passes all audits, the registrar issues the ISO 9000 certification.
    - The certification formally recognizes that the organization maintains a quality management system meeting international standards.
6. **Continued Surveillance / Inspection:**
    - After certification, the registrar periodically conducts inspections to ensure the organization continues to comply with ISO 9000 standards.
    - This helps maintain quality standards, supports continuous improvement, and ensures long-term adherence.

Advantages of ISO 9000 Certification:

Some of the advantages of the ISO 9000 certification process are following :

- Business ISO-9000 certification forces a corporation to specialize in "how they are doing business". Each procedure and work instruction must be documented and thus becomes a springboard for continuous improvement.
- Employees morale is increased as they're asked to require control of their processes and document their work processes
- Better products and services result from continuous improvement process.

• Increased employee participation, involvement, awareness and systematic employee training are reduced problems

Software Engineering Standards

Software engineering standards are agreed-upon rules, formats, and techniques that guide software development, testing, maintenance, and management. They provide a framework for processes, helping improve software quality, reliability, interoperability, and team collaboration.

Purpose of Standards:

- Ensure consistent quality across software products and projects.
- Provide a common framework for software development and maintenance.
- Facilitate communication among teams, clients, and stakeholders.
- Help in compliance with legal, regulatory, and contractual requirements.

Types of Software Engineering Standards:

- Process Standards: Guidelines for software development processes (e.g., ISO 9000, ISO/IEC 12207).
- Product Standards: Specifications for software products, including functionality, performance, and quality metrics.
- Documentation Standards: Rules for creating and maintaining clear, consistent project and technical documentation.
- Testing Standards: Guidelines for software testing, verification, and validation (e.g., IEEE 829 for test documentation).

Popular Standards in Software Engineering:

- ISO 9000 Series: Quality management systems for consistent process quality.
- ISO/IEC 12207: Life cycle processes for software development, maintenance, and operation.
- IEEE Standards: Various standards for software design, testing, documentation, and reliability.
- CMMI (Capability Maturity Model Integration): Framework to assess and improve software process maturity.

Importance of Software Engineering Standards

1. **Quality Assurance:**
    - Ensure software meets defined quality attributes like functionality, reliability, and usability.
2. **Interoperability:**
    - Allow different software systems to communicate and work together using common protocols and data formats.
3. **Collaboration:**
    - Provide a common language and processes for teams, improving coordination and cohesion.
4. **Efficiency:**
    - Standardized processes and coding practices streamline development, reduce errors, and simplify maintenance.
5. **Risk Reduction:**
    - Help identify and mitigate risks early in the software development lifecycle, ensuring smoother project execution.

Benefits of Using Standards:

- Enhances software quality and reliability.
- Improves team coordination and communication.
- Reduces development time and costs through repeatable processes.
- Facilitates audits, certifications, and compliance with industry norms.
- Encourages continuous improvement in software processes.