UNIT - 1

PHASES OF SOFTWARE PROJECT

A **software project** is completed through a set of well-defined steps called **phases**.
Each phase has a specific goal and output.

The main phases are:

1. **Requirements Gathering and Analysis**
2. **Planning**
3. **Design**
4. **Development (Coding)**
5. **Testing**
6. **Deployment and Maintenance**

1.Requirements Gathering and Analysis

This is the **first and most important phase**.

Meaning:

In this phase, the developer collects information about what the customer wants the software to do.

Activities:

- Meet users or clients
- Understand their needs
- Find functional and non-functional requirements
- Remove ambiguities and conflicts

Output:

- All requirements are written in a document called **SRS (System Requirements Specification)**

Importance:

- The SRS acts as a **contract** between customer and developer.
- If requirements are wrong, the entire project will fail.

2.Planning

This phase decides **how** and **when** the project will be completed.

Activities:

- Decide project scope
- Estimate cost and time
- Decide number of developers and testers
- Create schedules and milestones

Output:

- **Project Plan**
- **Test Plan**

Importance:

- To ensure that the project is completed **within time, cost and quality limits**.

3. Design

This phase decides **how the system will be built**.

Purpose:

To convert the SRS into a technical solution.

Two levels of design:

1. **High Level Design (HLD)** – System architecture
2. **Low Level Design (LLD)** – Detailed module design

Output:

SDD – System Design Description

Importance:

This document tells programmers:

- What to build
- How to build it

4.Development (Coding)

This is the phase where **actual software is created**.

Activities:

- Write program code
- Follow the design
- Create user manuals and documentation

Output:

- A working software product.

Importance:

- Design acts as a **blueprint**, and coding builds the software from it.

5.Testing

Testing checks whether the software works correctly.

Meaning:

Testing is the process of executing the software to find errors.

Objectives:

- Find defects

- Ensure requirements are satisfied
- Check quality

Types:

- Unit testing
- Integration testing
- System testing
- Acceptance testing

Testing reduces the risk of failure after release.

6.Deployment and Maintenance

Deployment:

After testing, the software is delivered to the customer and installed in their system.

Maintenance:

After users start using the software, problems may appear. These must be fixed.

Types of maintenance:

1. **Corrective Maintenance**
   Fixing bugs reported by users
2. **Adaptive Maintenance**
   Modifying software for new OS, hardware, etc.
3. **Preventive Maintenance**
   Improving software to prevent future problems

QUALITY, QUALITY ASSURANCE, AND QUALITY CONTROL

Quality:

- Quality means the software meets customer requirements correctly and consistently.
- Requirements are converted into software features.
- Each feature is tested using test cases.
- A test case defines:
    - Environment
    - Inputs
    - Processing
    - Internal changes
    - Expected outputs
- If actual results match expected results, the test passes.
- If not, the software has a defect.

Quality Control (QC)

- Quality control focuses on the **product**.
- It checks whether the built software works correctly.
- Errors are found by testing.
- If defects are found, they are fixed and tested again.
- QC is **defect detection and correction oriented**.
- It is usually done by a **testing or QC team**.
- Examples: unit testing, system testing, acceptance testing.

Quality Assurance (QA)

- Quality assurance focuses on the **process** of software development.
- It aims to **prevent defects** before they occur.
- It ensures that correct methods and standards are followed.
- QA activities include:
    - Design reviews
    - Code reviews
    - Audits
    - Coding standards
- QA is performed throughout the software life cycle.
- It is the responsibility of everyone in the project.

Difference between Quality Assurance and Quality Control

Quality Assurance (QA)	**Quality Control (QC)**
Focuses on process	Focuses on product
Prevents defects	Finds and fixes defects
Done throughout the project	Done after product is built
Staff responsibility	Testing team responsibility
Examples: audits, reviews	Examples: testing

TESTING, VERIFICATION, AND VALIDATION

Testing

- Testing is the process of executing software to find defects.
- It is usually done after coding and before deployment.
- Testing checks whether the actual output matches the expected output.
- The main aim of testing is to **find errors**, not to prove the software is perfect.
- Effective testing increases customer satisfaction and confidence.
- Testing is carried out by a team whose goal is to detect defects before the product reaches the customer.
- Testing is part of **Quality Control**.

Why Testing Is Important ?

- Defects can occur in any phase of software development.
- Finding defects early reduces cost and effort.
- Testing ensures the software behaves as expected in real-life usage.
- It improves the chances of customer acceptance.

Verification

- Verification checks whether each development phase is done correctly.
- It answers the question: **"Are we building the product right?"**
- It focuses on the **process** and documents.
- Verification is done during each phase of the life cycle.
- Examples:
    - Requirement reviews

    - Design reviews
    - Code reviews
- Verification is a **proactive** activity.
- It is part of **Quality Assurance**.

Validation

- Validation checks whether the final product meets customer requirements.
- It answers the question: **"Are we building the right product?"**
- It focuses on the **actual software**.
- Validation is done by testing the software.
- Examples:
    - Unit testing
    - Integration testing
    - System testing
- Validation is a **reactive** activity.
- It is part of **Quality Control**.

Relationship between QA, QC, Verification, and Validation

- **Quality Assurance = Verification**
- **Quality Control = Validation = Testing**

Comparison Table

Verification	Validation
Are we building the product right?	Are we building the right product?
Focuses on process and documents	Focuses on actual software
Done throughout development	Done after or during development
Examples: reviews, audits	Examples: testing
Part of Quality Assurance	Part of Quality Control

PROCESS MODEL TO REPRESENT DIFFERENT PHASES (ETVX MODEL)

Process Model

- A process model is a way to describe how each phase of software development is carried out.
- It helps in **verification and validation** of work done in each phase.
- It reduces the time between **defect creation and defect detection**.
- Each phase is clearly defined to avoid confusion and errors.

ETVX Model

ETVX stands for:

- **E – Entry criteria**
- **T – Tasks**
- **V – Verification**
- **X – Exit criteria**

Each phase of software development follows these four steps.

Components of ETVX Model

1. Entry Criteria

- Defines when a phase can start.
- Specifies what inputs are required for that phase.
- Ensures that the previous phase is completed properly before starting the next.

2. Tasks

- Describes the activities to be performed in that phase.
- Includes measurements to track progress and quality.
- Reduces confusion by clearly stating what to do.

3. Verification

- Checks whether tasks are done correctly.
- Helps in early detection of defects.
- Prevents errors from moving to the next phase.

4. Exit Criteria

- Defines when a phase can be considered complete.
- Specifies the outputs of the phase.
- Ensures that only correct and complete work is passed to the next phase.

Advantages of ETVX Model

- Prevents starting a phase too early.
- Helps in detecting defects early.
- Reduces misunderstanding of tasks.
- Improves quality through clear validation.
- Ensures smooth transition between phases.

LIFE CYCLE MODELS

Meaning

- A life cycle model describes how the different phases of a software project are organized and connected.
- It explains how a project moves from requirements to maintenance.
- While the **ETVX model** describes a single phase, the **Life Cycle Model** describes the **entire project**.

Attributes of a Life Cycle Model

1. Activities Performed

- Describes all the activities carried out in the project.
- Includes both technical and non-technical tasks.

2. Deliverables from Each Activity

- Each activity produces an output.
- Examples:
    - Requirements phase → **SRS**
    - Design phase → **SDD**
    - Coding phase → **Program code**

3. Validation of Deliverables

- Checks whether each output meets its goals.
- Ensures that documents and software are correct and useful.

4. Sequence of Activities

- Defines the order in which activities are performed.
- Some steps may be repeated until correct results are obtained.
- Example: requirements may be revised and revalidated.

5. Verification and Communication

- Ensures that errors are traced back to their source.
- Helps different phases communicate and coordinate.
- Makes sure mistakes are corrected in the right phase.

Purpose of Life Cycle Models

- To organize and control software development.
- To improve quality through verification and validation.
- To reduce errors and delays.

DIFFERENT LIFE CYCLE MODELS

1. **Waterfall Model**
2. **Prototyping and Rapid Application Development (RAD) Models**
3. **Spiral or Iterative Model**
4. **V Model and Modified V Model**

1. WATERFALL MODEL

- The Waterfall model is one of the earliest software life cycle models.
- In this model, work flows step by step from one phase to the next, like a waterfall.
- Each phase must be completed before moving to the next phase.

Phases in Waterfall Model:

1. Requirements gathering
2. Design
3. Development (Coding)
4. Testing

5. Deployment / Maintenance

Each phase starts only after the previous phase is finished.

Working of Waterfall Model:

- First, requirements are collected and written in **SRS**.
- Next, the design is prepared in **SDD**.
- Then, coding is done according to the design.
- After coding, the software is tested.
- Finally, the product is delivered to users.

Main Features:

1. The project is divided into **separate phases**.
2. Each phase gives output to the next phase.
3. Errors are sent back to previous phases for correction.

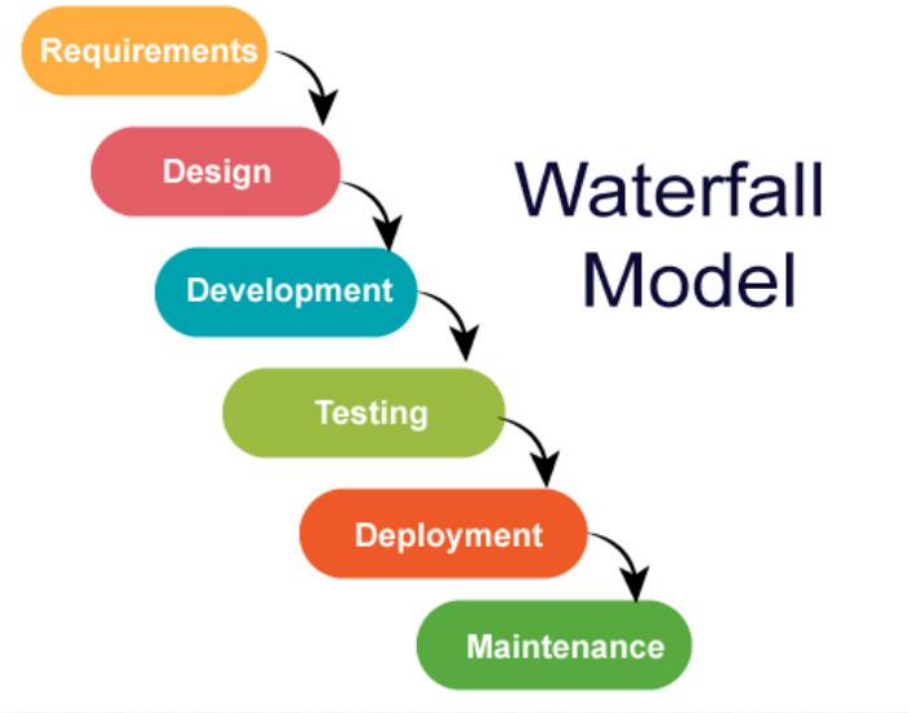
Advantages:

- Very simple and easy to understand.
- Easy to manage because phases are clearly defined.
- Suitable when requirements are clear and stable.

Disadvantages:

- Feedback is slow.
- Errors are found late.
- Changes are difficult to make.

- Not suitable for complex or changing projects.



2.Prototyping and Rapid Application Development (RAD) Models

Why these models are used ?

- Users may not clearly know their requirements at the start.
- Changes are common during software development.
- Early and frequent user feedback improves product quality.

a). Prototyping Model

- A **prototype** is a small working model of the software.
- It is built to understand user requirements clearly.

Steps:

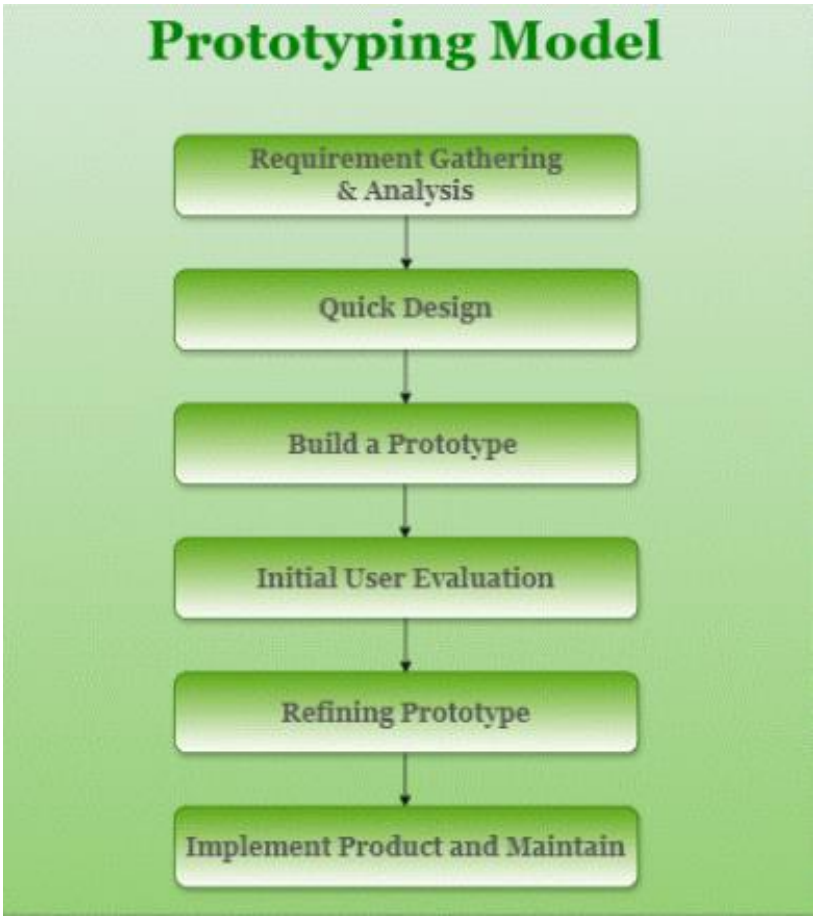
1. Developers talk with customers to understand needs.
2. A prototype is created (sample screens, reports, and workflow).
3. Customers review the prototype and give feedback.
4. Changes are made based on feedback.
5. Final **SRS** is prepared.
6. The prototype is discarded.
7. Real software is built using the SRS.

Advantages:

- Helps users understand what they want.
- Reduces misunderstandings.
- Improves accuracy of requirements.

Disadvantages:

- Prototype may be wrongly used as final product.
- Prototype is built quickly using shortcuts, which may cause defects.
- Not suitable for general-purpose products.



b). <u>Rapid Application Development (RAD) Model</u>

- RAD is a fast version of the Prototyping model.
- The **actual software** is built, not just a prototype.
- The product is not discarded.

Features:

- Uses **CASE tools** (Computer Aided Software Engineering).
- CASE tools help in:
    - Collecting requirements
    - Storing data
    - Creating design

- Generating code automatically

Advantages:

- Faster development.
- Better verification and validation.
- Suitable for general-purpose applications.

Limitations:

- CASE tools are expensive.
- More suitable for application software than system software.

Difference between Prototyping and RAD

Prototyping Model	**RAD Model**
Builds a sample model	Builds the actual product
Prototype is discarded	Product is kept
Less formal	Uses CASE tools
Limited validation	Strong verification & validation

3. SPIRAL OR ITERATIVE MODEL

- In the Spiral (or Iterative) model, software is developed in **small parts**.
- Requirements, design, coding, and testing are done **again and again** until all requirements are completed.
- Each cycle is called an **iteration** or **spiral**.

How it works ?

- Requirements are taken in small sets.
- Each set goes through:
    - Requirements
    - Design
    - Coding
    - Testing
- After one cycle, a **working version** of the software is produced.
- New requirements are added in the next cycle.

Important Features:

- Different requirements can be in different phases at the same time.
- One requirement may be in design, another in testing, another in coding.
- If a defect is found, only that requirement goes back to an earlier phase.

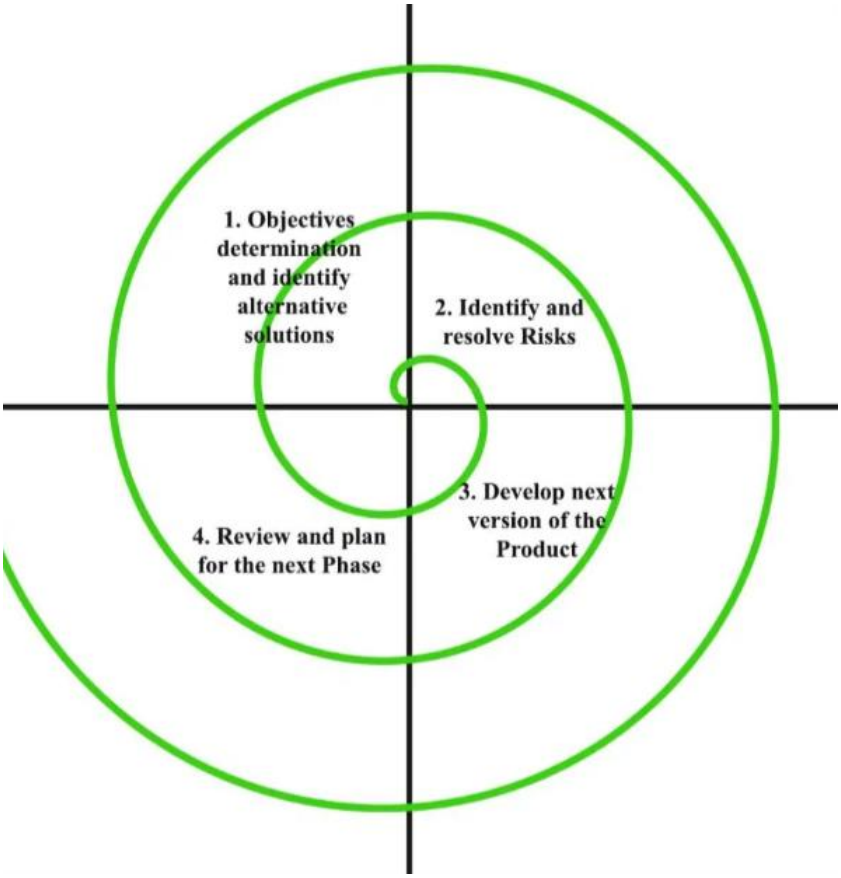
Advantages:

- Software is built **step by step**.
- Users can see and use parts of the software early.
- Customer feedback is taken regularly.
- Risk of major failure is reduced.
- Progress is easy to see.

Disadvantages:

- It is difficult to decide the final release date.

- Needs good planning and management.
- More complex than the Waterfall model.



4. V-MODEL

- The V-Model is an extension of the Waterfall Model.
- It shows that **testing is planned along with development**.
- Each development phase has a **matching testing phase**.
- Development is on the **left side** of the "V".
- Testing is on the **right side** of the "V".
- Test **design** is done early, but test **execution** is done after coding.

Development Phases and Their Testing Phases

Development Phase	**Corresponding Test**
Business Requirements	Acceptance Testing
Software Requirements	System Testing
High Level Design	Integration Testing
Low Level Design	Component Testing
Coding	Unit Testing

How the V-Model Works ?

- Requirements are written → Acceptance tests are designed.
- System design is done → System tests are designed.
- High-level design is done → Integration tests are designed.
- Low-level design is done → Component tests are designed.
- Coding is done → Unit tests are executed.

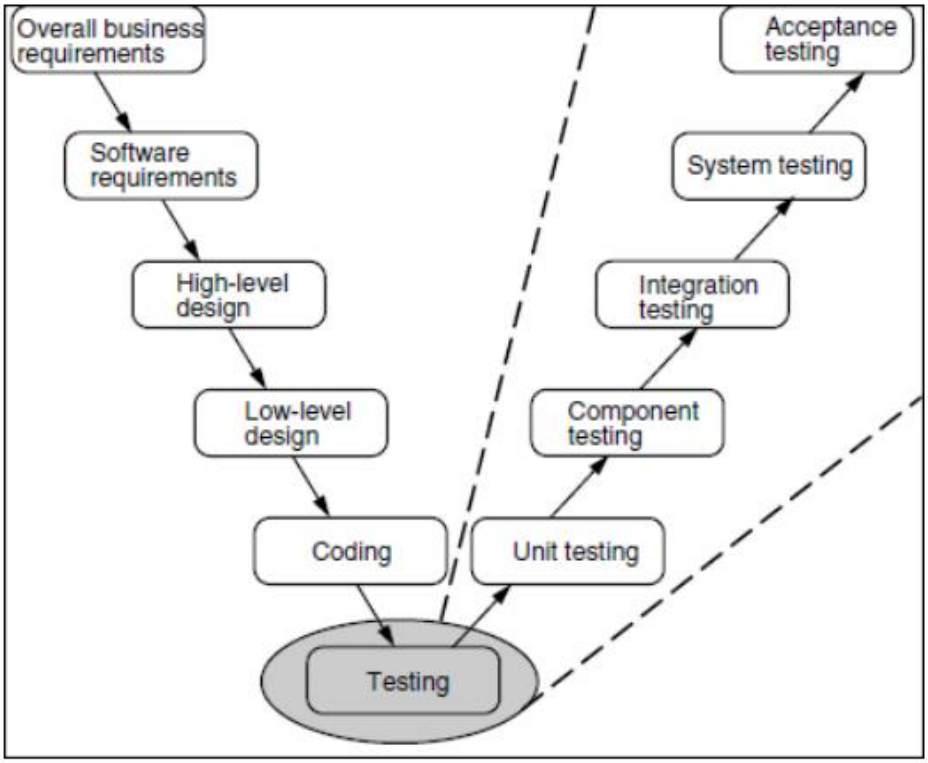
Key Features:

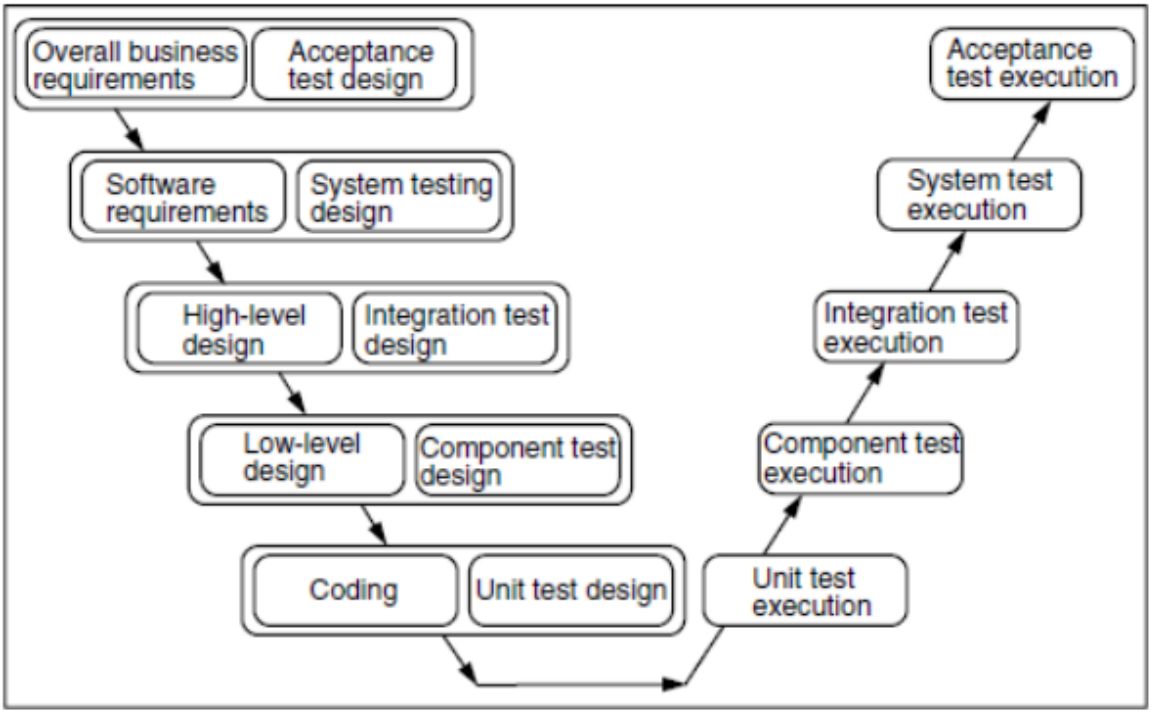
1. Testing is divided into **test design** and **test execution**.
2. Test design is done **early** in the project.
3. Different tests are done at different levels.
4. Errors can be found early through planned testing.

Advantages

- Reduces last-minute surprises.
- Testing is well planned.

- Quality is improved.
- Saves time at the end of the project.





Modified V Model

- It is an improved version of the V Model.
- Allows **different parts of a product** to be tested at the same time.
- Each unit or component moves to the next testing phase when **ready**, without waiting for others.
- Testing phases for parts:
    1. Unit Testing
    2. Component Testing
    3. Integration Testing
    4. System Testing
- **Entry criteria**: Check if the part is ready for testing.
- **Exit criteria**: Check if testing is complete for the part.
- Integration testing can start once some components are ready.
- System testing starts when **all components are complete**.
- **Advantages:**
    - Saves time by testing in parallel.
    - Ensures proper quality at each stage.
- Difference from V Model: Traditional V waits for the whole product; Modified V allows parts to move independently.

Comparison Table

Model	When to Use	Testing Information
Waterfall	- Clear steps - Each step must finish before next	- Testing happens late - Fixing errors takes more time
Prototyping	- User feedback is available	- Helps check requirements early - Reusing prototype may cause problems
RAD	- User feedback and modeling tools available	- Feedback happens quickly - Tools help with documentation and testing
Spiral	- Product made in small increments	- Can check and fix at each step - Frequent demos and releases possible
V Model	- Tests can be planned while coding	- Tests designed early - Helps validate each step on time - Reduces delay
Modified V Model	- Product can be split into independent parts	- Each part tested in parallel - Faster overall testing

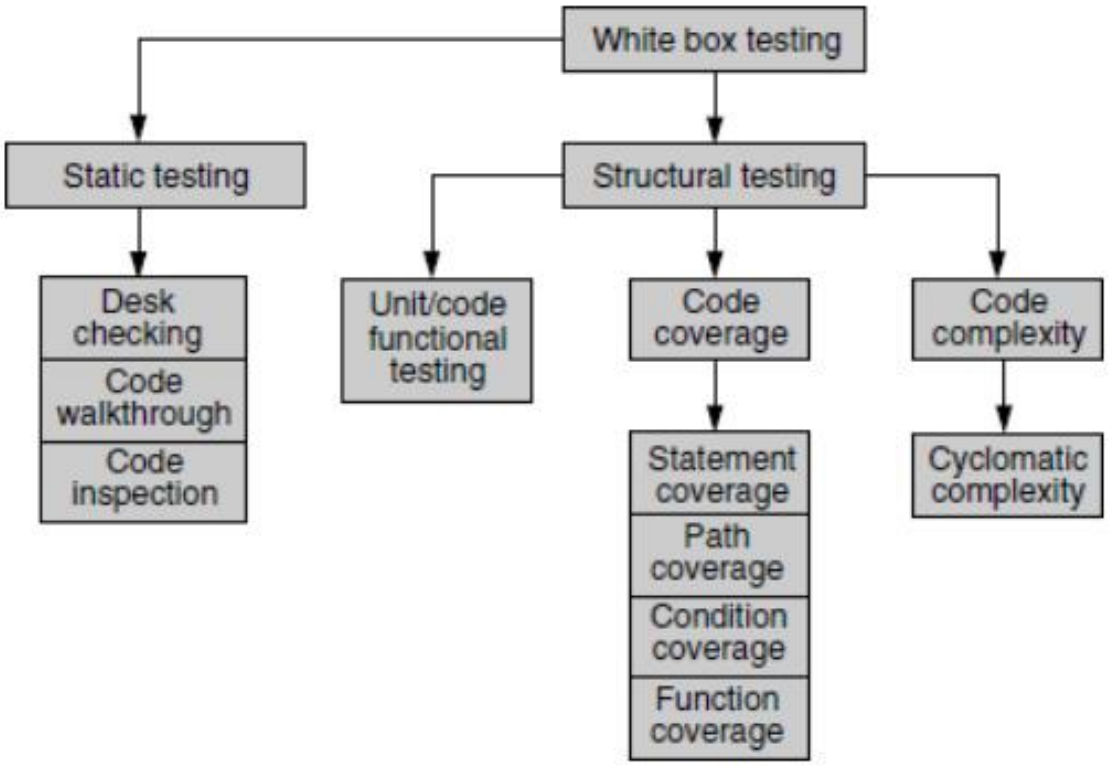
WHITE BOX TESTING

White Box Testing (also called **Clear Box / Glass Box / Open Box Testing**) is a testing method in which the **internal structure, logic, and code of the program are examined**.

The tester knows:

- How the program is written
- The program logic
- Flow of control
- Data structures

It checks **how the software works**, not just what it produces.



Static Testing	Structural Testing
Testing **without executing** the program	Testing **by executing** the program
Finds errors in code structure	Finds errors in program behavior
Done by reviewing code	Done by running test cases

---

1. STATIC TESTING

Static testing means checking the **code manually without running it**.

(a) Desk Checking

- The programmer checks the code line by line.
- Errors like:
    - Wrong logic
    - Missing statements
    - Incorrect formulas

    are found.

Example:
Checking whether total = a + b is written correctly.

(b) Code Walkthrough

- The programmer explains the code to a group.
- Others listen and point out mistakes.

Goal:
To find logical errors and improve understanding.

(c) Code Inspection

- A **formal review** done by a team.

- Code is checked using a checklist.
- More systematic than walkthrough.

Finds:

- Syntax errors
- Standard violations
- Wrong logic

2. STRUCTURAL TESTING

Structural testing checks **how the code actually runs**.

It includes:

(a) Unit / Functional Testing

Each small unit (function, module) is tested.

Example:
Testing a calculateTotal() function separately.

(b) Code Coverage

Checks **how much of the code is executed** during testing.

Types of Code Coverage

Type	Meaning
Statement Coverage	Every line of code should run at least once
Condition Coverage	Every condition (true/false) must be tested
Path Coverage	All possible execution paths must be tested
Function Coverage	All functions must be called at least once

Example:

```
if (a > b)
  print("A");
else
  print("B");
```

Condition coverage means testing:

- a > b is TRUE
- a > b is FALSE

(c) Code Complexity

Measures how **complex** the program logic is.

Cyclomatic Complexity

Cyclomatic Complexity tells **how many independent paths** exist in a program.

It shows:

- How difficult the code is
- How many test cases are needed

Formula:

$$M = E - N + 2$$

Where:

- E = Number of edges in flow graph
- N = Number of nodes

Or simply:

$$M = \text{Number of decision statements} + 1$$

Example:

If a program has:

- 2 if statements

  Cyclomatic complexity = 2 + 1 = **3**

So, **3 test cases** are needed.

UNIT - 2

BLACK BOX TESTING

What is Black Box Testing?

- Black box testing is a software testing method in which the tester checks the system using only requirements and specifications.
- The tester does not see the program code.
- Testing is done from the user's point of view.
- The tester checks whether the actual output matches the expected output.
- It focuses on what the system does, not how it works internally.

Why Black Box Testing?

- It verifies the overall functionality of the software.
- It ensures the software works according to the requirements.
- It helps identify missing or incorrect requirements.
- It considers the end-user perspective.
- It checks both valid inputs and invalid inputs.
- It helps in finding user-level defects.

When to do Black Box Testing?

- It starts from the requirements phase.
- Test scenarios can be prepared during the design phase.
- Test execution is done after the software is developed.
- It is also used during regression testing.

How to do Black Box Testing?

Black box testing is done using the following techniques:

1. Requirements based testing
2. Positive and negative testing
3. Boundary value analysis
4. Decision table testing
5. Equivalence partitioning
6. State based testing
7. Compatibility testing
8. User documentation testing

1.Requirements Based Testing

- Test cases are prepared from the Software Requirement Specification (SRS).
- Both explicit and implicit requirements are tested.
- A Requirements Traceability Matrix (RTM) is used to link requirements and test cases.
- It ensures that all requirements are tested.

Example: If the requirement says “User must be able to log in using a valid username and password”, the tester checks whether the system really allows the user to log in with correct details.

2.Positive Testing and negative testing

Positive Testing: Checks whether the system works for valid inputs. Confirms that the system gives the correct expected result.

Example: The tester enters a valid email and correct password and checks whether the login is successful.

Negative Testing: Checks the system using invalid or unexpected inputs. Ensures the system does not crash or fail.

Example: The tester enters a wrong password and checks whether the system shows an error message.

3.Boundary Value Analysis

Tests are performed at the edges of input ranges.

It focuses on values like:

- minimum
- maximum
- just below
- just above the limits

It helps detect boundary-related defects.

Example: If a form accepts age from 18 to 60:

The tester checks using:

18

60

17

61

to see whether the system accepts and rejects correctly.

4.Decision Table Testing

- Used when the output depends on multiple conditions.
- Conditions and actions are written in a table form.
- Each row of the table represents a test case.
- It reduces confusion in rule-based systems.

Example: If a system gives a discount only when user is a member and purchase amount is above ₹1000

The tester prepares a table and tests all combinations such as:

member + above ₹1000

member + below ₹1000

not member + above ₹1000

not member + below ₹1000

5.Equivalence Partitioning

- Input data is divided into groups (classes).
- All values in one group behave the same way.
- One value from each group is tested.
- It reduces the number of test cases.

Example: If marks are accepted from 0 to 100:

The tester divides them as:

valid group → 0 to 100

invalid group → less than 0

invalid group → more than 100

Then one value is tested from each group, for example: 50, –5, and 120

6.State Based Testing

- The system is tested based on its states and transitions.
- Test cases are created using state transition diagrams.
- It is useful for workflow and transaction systems.

Example: An ATM card can be in these states:

card inserted

PIN entered

transaction completed

The tester checks whether the system moves correctly from one state to another.

7.Compatibility Testing

Checks whether the software works correctly with:

- different hardware
- operating systems
- browsers
- databases

It ensures the software works in different environments.

Example: The tester checks whether a website works properly on:

Chrome browser

mobile phone

Windows and Android devices.

8.User Documentation Testing

Checks whether:

- the product matches the user manual
- the manual correctly explains the product

It also verifies:

- spelling
- grammar
- correctness of steps

Example: If the user manual says

“Click the Save button to store the file”,

the tester checks whether clicking the Save button really saves the file.

INTEGRATION TESTING

Integration Testing as a Type of Testing

Integration testing as a type of testing mainly focuses on testing interfaces between modules.

It verifies whether different modules or components work correctly when connected together.

Two kinds of interfaces are tested:

✓ Internal interfaces

Interfaces used **inside the product** between internal modules.

✓ External (exported) interfaces

Interfaces exposed to outside users or third-party developers.

Integration testing checks:

- internal and external interfaces
- explicit and implicit interfaces

Explicit interfaces – documented interfaces
Implicit interfaces – not documented but known by developers

Integration testing uses test cases to:

- check data flow between modules
- check communication between components
- verify behaviour when modules interact

Although integration testing is generally considered a black-box testing approach, sometimes testers refer to design or code to find missing interfaces.

So, in practice it becomes:

black box + white box → gray box testing

Integration testing as a type mainly answers:

➞ *Are all interfaces working correctly?*

Integration Testing as a Phase of Testing

Integration testing is also treated as a separate testing phase.

This phase starts when:

- two components are available

and ends when:

- all interfaces among all components are tested

The purpose of this phase is to:

- test all interactions among modules
- handle the complexity caused by multiple modules and systems
- verify that integrated modules work correctly together

The final round of integration involving all components is called:

Final Integration Testing (FIT)
or
System Integration

Integration testing is considered a phase because:

- interactions are complex
- multiple teams develop different modules
- interfaces become available at different times

Scenario Testing

Scenario testing in integration testing means:

testing realistic usage situations where multiple modules and interfaces are used together.

It focuses on:

- different combinations of interface usage
- different ways in which internal and external interfaces are used
- real interaction paths across modules

Because modern interfaces are generic and reusable, the number of:

- usage combinations
- interaction paths

is very high.

This increases:

➞ the number of scenarios

➞ the complexity of integration testing

So, scenario testing checks:

“Does the system behave correctly in real interface usage situations?”

Defect Bash

Defect bash is an exploratory and focused testing activity carried out during integration testing.

It is done when:

- multiple components are already integrated
- testers try to discover interface related problems quickly

In defect bash:

- testers freely explore interfaces
- try unusual flows between modules
- try unexpected sequences of actions
- try incorrect or boundary interactions

The main goal is:

➞ to quickly find hidden interface defects
➞ especially those caused by interaction between modules

]

Defect bash is very useful in integration testing because:

- many defects occur only when components interact
- some implicit interfaces are not documented
- complex interaction paths are difficult to predict using only test cases

Important

Topic	Main focus
Integration testing as a type	Testing internal, external, explicit and implicit interfaces
Integration testing as a phase	Dedicated phase to test interactions among components
Scenario testing	Testing real-life usage scenarios across multiple interfaces
Defect bash	Exploratory testing to quickly find interface and interaction defects

UNIT - 3

SYSTEM AND ACCEPTANCE TESTING

SYSTEM TESTING

System testing is the testing of the **complete and integrated software system**.
It is performed after component testing and integration testing.
In system testing, the product is tested as a whole in an environment similar to the real user environment.
The main goal of system testing is to verify that the system satisfies **both functional and non-functional requirements**.
System testing checks the system from the **user's point of view** and not from the internal code structure.

Why System Testing is Done

System testing is done to ensure that the entire system works correctly as one unit.
It verifies that all modules work together without errors.
It checks whether the system behaves correctly under real usage conditions.
It helps to find defects that cannot be found in individual modules.
It confirms that the product is ready for customer evaluation.

Functional versus Non-Functional Testing

Functional testing verifies **what the system does**.
Non-functional testing verifies **how well the system works**.
Functional testing focuses on features and business functions.
Non-functional testing focuses on quality attributes such as performance, reliability and stability.

a). Functional Testing

Functional testing checks whether the software performs the required functions correctly.
It verifies that the system produces correct outputs for given inputs.
It ensures that business rules and workflows are followed.
Functional testing is normally performed using black-box testing.

Examples:
A user should be able to log in successfully.
A transaction should be completed correctly.
A report should display correct data.
A calculation should produce the correct result.

b) Non-Functional Testing

Non-functional testing checks the quality and behavior of the system.
It verifies how the system behaves under different operating conditions.

i. Reliability testing

Reliability testing checks whether the system can run for a long time without failure.
It measures failure rate and mean time between failures.
It ensures that CPU, memory and network usage remain stable during long execution.
It ensures that repeated operations do not create side effects such as memory buildup.

ii. Stress testing

Stress testing checks the behavior of the system beyond its normal limits.
The system is deliberately overloaded to create resource shortage.
It verifies that the system degrades gracefully and does not crash.
It also measures how quickly the system recovers from failure using MTTR.

iii. Performance and resource testing

This testing checks response time, throughput and resource utilization.
It ensures that the system performs within acceptable limits.

iv. Interoperability testing

Interoperability testing verifies that two or more products can exchange data and work correctly together.
It checks that information is correctly understood and processed by all participating systems.

ACCEPTANCE TESTING

Acceptance testing is performed after system testing.
It is usually carried out by customers or their representatives.
Its main purpose is to decide whether the product is ready for real use.
Acceptance test cases are limited in number and are based on real business scenarios.
Acceptance testing is not mainly intended to find defects.
It is intended to verify that the product meets the defined acceptance criteria.

Acceptance Criteria

Acceptance criteria define the conditions under which the product is accepted.

Product acceptance criteria: These criteria verify that required features, performance and legal requirements are satisfied.

Procedure acceptance criteria: These criteria verify that documents, release media and training are delivered as agreed.

Service level agreement criteria: These criteria verify commitments such as downtime limits and defect-fixing timelines.

Selection of Test Cases for Acceptance Testing

Acceptance test cases include end-to-end business scenarios.
They include domain-specific and real user scenario tests.
They include basic sanity tests to verify existing behavior.
They include test cases for newly added features.

They include a small number of important non-functional tests.
They include test cases related to legal and contractual requirements.

Execution of Acceptance Testing

Acceptance testing is done by people who understand the business usage of the product.
Testing team members support the acceptance team with test data and defect reporting.
Critical and high-priority defects must be fixed before release.
If major defects are found, the release may be delayed.

Summary of Testing Phases

Software testing is performed in multiple phases such as unit, component, integration, system and acceptance testing.
Each phase has defined entry and exit criteria.
Entry and exit criteria decide when a testing phase can start and finish.
The purpose of these criteria is to maintain quality and allow parallel execution of test activities.
Very loose criteria can reduce product quality.
Very strict criteria can delay the release.
A balanced approach helps achieve both quality and timely delivery.

UNIT – 4

PERFORMANCE TESTING & REGRESSION TESTING

PERFORMANCE TESTING

Performance testing not only checks speed, but also verifies:

- how the system behaves under normal load
- how it behaves under heavy load
- how it behaves when the system is overloaded

It helps to find:

- memory leaks
- CPU over-utilization
- slow database queries
- server crashes

It is mainly used for:

- web applications
- mobile applications
- cloud systems
- enterprise systems

Main objectives of performance testing

- To verify response time requirements
- To identify performance bottlenecks
- To ensure system stability for long usage
- To verify scalability of the system

Common types of performance testing

- **Load testing** – checks behaviour under expected load
- **Stress testing** – checks behaviour beyond capacity
- **Endurance (soak) testing** – checks performance over long time
- **Spike testing** – sudden increase and decrease of users
- **Scalability testing** – ability to handle growth in users

Factors governing Performance Testing

In addition to basic factors, the following also govern performance:

1. Workload model

Defines:

- number of users
- type of user actions
- frequency of actions

Example:
70% users browse, 20% search, 10% submit forms.

2. Data volume

Large database size can affect:

- search speed
- report generation

3. Think time

Delay between two user actions.

Example:
User waits 5 seconds before clicking the next button.

4. Application architecture

- single tier
- multi-tier
- microservices

Each architecture behaves differently under load.

5. Third-party services

External APIs (payment, SMS, email services) also affect performance.

6. Browser and device variations

Different browsers and devices can show different performance behaviour.

Methodology of Performance Testing

1. Requirement analysis

Collect:

- SLA (Service Level Agreement)
- acceptable response times
- peak user expectations

2.Test planning (extended)

Decide:

- entry and exit criteria

- risk areas
- reporting format

3.Test scenario design (extended)

Include:

- business-critical scenarios
- background activities
- error scenarios

4.Script development

Create performance scripts using tools and parameterize:

- usernames
- passwords
- form data

5.Baseline testing

First run with small load to establish baseline performance.

6.Test execution cycles

Multiple cycles are run with:

- increasing load
- tuning in between cycles

7.Result analysis (extended)

Analyse:

- response time distribution
- error percentage
- resource usage trends

8.Reporting

Prepare reports showing:

- bottlenecks
- improvement suggestions
- comparison between runs

Tools for Performance Testing

Popular tools include:

- **Apache JMeter** – developed by
  Apache Software Foundation
  Supports web, database, REST, SOAP and FTP testing.
- **LoadRunner** – by
  OpenText
  Suitable for enterprise-level and large-scale testing.
- **Gatling** – developed by
  Gatling Corp
  Script based and suitable for CI/CD pipelines.
- **NeoLoad** – by
  Tricentis
  Good support for cloud and microservices environments

Process for Performance Testing

Step 1 – Define KPIs

Key performance indicators such as:

- average response time
- 90th percentile response time
- error rate
- throughput

Step 2 – Test data preparation

Prepare:

- realistic user data
- valid and invalid data

Step 3 – Load model creation

Define:

- number of concurrent users
- ramp-up and ramp-down time

Step 4 – Test execution and monitoring

Monitor:

- CPU
- memory
- disk I/O
- network traffic

Step 5 – Bottleneck identification

Identify whether bottleneck lies in:

- application code
- database
- web server
- network

Step 6 – Performance tuning

Activities include:

- SQL optimization
- caching
- configuration tuning
- thread pool adjustments

Step 7 – Re-execution and validation

After tuning, run the same tests again to validate improvements.

Challenges in Performance Testing

1. Limited production-like environment

It is difficult to reproduce real production scale in testing labs.

2. Tool complexity

Learning scripting, correlation and monitoring requires skilled testers.

3. High infrastructure cost

Multiple servers and monitoring tools are required.

4. Test data preparation

Large and realistic data sets are difficult to create.

5. Non-functional requirement ambiguity

Performance expectations are often unclear or missing.

6. Dynamic system behaviour

Cloud auto-scaling and dynamic resources make analysis complex.

REGRESSION TESTING

What is Regression Testing?

Regression testing confirms that:

- new code
- modified code
- bug fixes

do not introduce new defects into existing functionality.

It mainly focuses on:

- system stability
- backward compatibility

Regression testing is an important part of:

- maintenance testing
- continuous delivery environments

Types of Regression Testing

1. Unit regression testing

- Mostly done by developers
- Executed at module or function level
- Often automated

2. Partial regression testing

- Used when change impact is limited
- Helps reduce test execution time

3. Complete (full) regression testing

- Used when:
    - major release
    - architectural changes
    - critical bug fixes

4. Selective regression testing

- Based on:
    - risk
    - usage frequency
    - business importance

5. Progressive regression testing (extra)

Performed when new test cases are added to existing test suite.

When to do Regression Testing?

Regression testing is performed when:

- database schema is changed
- external service is replaced
- operating system or platform changes
- build is generated after code merge
- configuration values are updated

It is usually mandatory:

- before UAT
- before production deployment

How to do Regression Testing?

Step 1 – Change and impact analysis

Analyse:

- code changes
- impacted modules
- dependency areas

Step 2 – Regression test suite preparation

Create a dedicated suite containing:

- high-priority cases

- defect-prone areas
- core business flows

Step 3 – Automation support

Regression testing is usually automated using tools like Selenium.
The Selenium project is maintained by **SeleniumHQ**.

Step 4 – Execution in multiple environments

Run tests in:

- different browsers
- different operating systems
- different test environments

Step 5 – Defect analysis

Verify whether failures are:

- new defects
- old known issues
- environment related problems

Step 6 – Test result reporting

Prepare regression reports showing:

- total executed cases
- passed and failed cases
- blocked cases

Best Practices in Regression Testing

1. Maintain a stable and clean test suite

Remove:

- duplicate test cases
- outdated scenarios

2. Use automation frameworks

Use proper frameworks for:

- test maintenance
- reporting
- reusability

3. Integrate regression tests with CI/CD

Regression tests should run automatically after each build.

A popular CI tool is maintained by
Jenkins.

4. Perform risk-based regression testing

High-risk and critical features should be tested first.

5. Maintain traceability

Maintain mapping between:

- requirements
- test cases
- defects

6. Keep regression execution time under control

Apply:

- test case selection
- test case prioritization
- parallel execution

7. Store historical regression results

Helps in:

- identifying unstable areas
- analysing failure patterns

UNIT 5

1. Test Planning

Test planning is the activity of deciding what to test, how to test, who will test, when to test and what resources are needed.

It is the first and most important management activity in testing.

Objectives of Test Planning

- To define the scope of testing
- To select appropriate testing strategy
- To estimate time, cost and effort
- To assign roles and responsibilities
- To identify risks and prepare mitigation plans
- To define entry and exit criteria

Inputs to Test Planning

- Software Requirement Specification (SRS)
- Project plan
- Design documents
- Past project experience
- Organizational testing standards

Components of a Test Plan

A typical Test Plan Document includes

Section	Explanation
Introduction	It explains the purpose of testing and gives an overview of the testing work.
Test objectives	It explains what the testing is expected to achieve.
Scope	It lists the features that will be tested and the features that will not be tested.
Test strategy	It explains the overall method and approach used for testing.
Test levels	It shows the levels of testing such as unit, integration, system, and acceptance testing.
Test types	It shows the types of testing such as functional, performance, and security testing.
Test environment	It describes the hardware, software, and tools used for testing.
Test data	It specifies the data required to perform testing.
Schedule	It shows the planned dates and milestones for testing activities.
Resources	It lists the testers, tools, and infrastructure needed for testing.
Roles & responsibilities	It explains the duties of the test manager, testers, and developers.
Entry criteria	It states the conditions that must be satisfied before testing can start.
Exit criteria	It states the conditions that must be satisfied to stop testing.

Risks and mitigation	It identifies possible testing risks and the actions to reduce them.
Deliverables	It lists the documents and outputs such as test cases, reports, and logs.

Entry Criteria (examples)

- Requirements are approved
- Test environment is ready
- Test data is prepared

Exit Criteria (examples)

- All planned test cases executed
- No critical defects open
- Test summary report prepared

2. Test Management

Test management is the process of planning, organizing, monitoring and controlling all testing activities during the project.

Main Activities of Test Management

1. Test planning and scheduling
2. Resource management
3. Tool selection and management
4. Risk management
5. Defect management

6. Monitoring and control
7. Reporting and communication

Responsibilities of Test Manager

- Prepare test plan
- Assign tasks to testers
- Track test progress
- Monitor defect status
- Resolve testing issues
- Communicate with project stakeholders
- Ensure quality objectives are met

Test Management Tools (examples)

- Test case management tools
- Defect tracking tools
- Automation tools

These tools help in:

- tracking test cases,
- tracking defects,
- reporting metrics.

3. Test Process

The test process defines a systematic sequence of steps followed during testing.

Phases of Test Process

a). Requirement Analysis

- Study and understand requirements
- Identify testable requirements
- Identify test types and test levels
- Prepare requirement traceability

b). Test Planning

- Prepare test plan
- Define strategy, schedule and resources

c). Test Design (Test Case Development)

- Write test cases
- Prepare test data
- Review test cases
- Create test scenarios

d). Test Environment Setup

- Install software
- Configure hardware
- Verify tools
- Check test data availability

e). Test Execution

- Execute test cases
- Compare actual result with expected result
- Mark test cases as pass or fail
- Report defects

f). Defect Tracking and Control

- Log defects
- Assign defects to developers
- Track status
- Perform retesting and regression testing

g). Test Closure

- Check exit criteria
- Collect metrics
- Prepare test summary report
- Document lessons learned

Defect Life Cycle

Typical defect states:

New → Assigned → Open → Fixed → Retest → Closed
If issue is still present → Reopened

4. Test Execution

Test execution is the phase in which test cases are run on the software to verify whether it behaves as expected.

Activities in Test Execution

- Select test cases
- Prepare environment
- Execute test cases
- Record actual results
- Log defects
- Retest fixed defects
- Perform regression testing

Outputs of Test Execution

- Executed test cases
- Defect reports
- Test execution logs

5. Test Reporting

Test reporting is the activity of communicating the test status and results to stakeholders.

Purpose of Test Reporting

- To inform project team about quality level
- To support release decisions
- To show testing progress and coverage

Test Report – Typical Contents

Item	Description
Test scope	It clearly explains what parts of the software were tested.
Environment details	It describes the hardware and software used for testing.
Test execution summary	It shows the total number of test cases and how many passed and failed.
Defect summary	It shows how many defects are open and closed and their severity levels.
Coverage	It shows how many requirements or features were tested.
Risks and issues	It lists the remaining problems and concerns after testing.
Conclusion	It gives the final testing result and overall quality status of the software.

Types of Test Reports

- Daily / weekly status report
- Test execution report
- Defect report
- Test summary (closure) report

6. Best Practices in Testing

- Start testing early in the development life cycle
- Write clear and maintainable test cases
- Review test artifacts
- Maintain traceability between requirements and tests
- Prioritize test cases based on risk
- Automate repetitive and stable tests
- Maintain proper defect documentation
- Communicate frequently with development team
- Perform regular regression testing
- Use metrics for continuous improvement

Test Metrics and Measurements

Test Metrics

Test metrics are numerical measures used to evaluate the quality, progress and effectiveness of testing activities.

They help management to make informed decisions.

Objectives of Test Metrics

- Measure testing progress
- Measure product quality
- Improve testing efficiency
- Identify problem areas
- Support release decisions

a). Project Metrics

Project metrics measure the overall testing status of the project.

Examples

- Total number of test cases planned
- Total number of test cases executed
- Total defects detected
- Total defects fixed
- Total test effort used

Typical Project Metric Example

Test Execution Percentage

$$\text{Test Execution Percentage} = \frac{\text{Number of Executed Test Cases}}{\text{Total Number of Planned Test Cases}} \times 100$$

b). Progress Metrics

Progress metrics measure how much testing work has been completed compared to the plan.

Common Progress Metrics

Metric	Description
Test case execution progress	It shows the percentage of test cases that have already been executed.

Test design progress	It shows how many test cases have been written compared to how many were planned.
Defect discovery trend	It shows how many defects are found each day or each week.
Environment readiness	It shows whether the testing environment is ready and available for testing.

Example

$$\text{Test Design Completion} = \frac{\text{Test Cases Prepared}}{\text{Test Cases Planned}} \times 100$$

c). Productivity Metrics

Productivity metrics measure efficiency of the testing team.

Common Productivity Metrics

Metric	Explanation
Test cases designed per tester per day	Measures design productivity
Test cases executed per day	Measures execution speed
Defects detected per tester	Measures detection capability
Effort per test case	Time spent to execute one test case

Example

$$\text{Tester Productivity} = \frac{\text{Total Test Cases Executed}}{\text{Total Tester Effort (person-days)}}$$

d). Release Metrics

Release metrics indicate whether the software is ready for release.

Important Release Metrics

Metric	Description
Open defects count	It shows the number of defects that are still not fixed.
Severity distribution	It shows how many defects are classified as critical, major, and minor.
Defect leakage	It shows the number of defects found after the software is released.
Test coverage	It shows the percentage of requirements that have been tested.
Defect density	It shows the number of defects found per size of the software.

Example

$$\text{Test Coverage} = \frac{\text{Requirements Tested}}{\text{Total Requirements}} \times 100$$

Advantages of Using Test Metrics

- Improves visibility of testing
- Helps detect schedule risks early
- Improves planning accuracy
- Supports management decisions
- Improves process maturity