What is Software Testing?

Testing is a **core part of QA (Quality Assurance)**.
It helps check if the software works correctly **by running it** and **observing the output**.

✅ **Key Idea:**

- **Run the software ➡️ Watch what happens ➡️**
  - If something **fails**, you try to **find the cause (fault/bug)**
  - If **nothing fails**, it builds **confidence** that the software is doing its job

---

📘 **Example:**

Imagine you're testing a **calculator app**:

- Input: 2 + 2
- Expected Output: 4
- If it shows 5, that's a **failure**, and you start debugging
- If it shows 4, then the app is **working for that input**

🛠️ **TESTING THE SYSTEM**

Testing is not just clicking buttons — it's a **structured and organized process** with techniques, tools, and strategies.

📌 **Key Points Explained:**

1. 🧩 **Testing is a Distinct Process**
   - Has its own **strategies**, **techniques**, and **goals**
   - Not the same as regular development work
2. 🐞 **Purpose of Testing**
   - To **find defects (bugs)**
   - To **assess software quality** — especially things like:
     - Reliability

        - Performance
        - Usability
        - Security

3. 🕒 **Testing Starts Early**
    - Not just at the end — testing begins during planning and continues throughout development
      (e.g., reviewing requirements, writing testable code, unit testing, etc.)

4. 👥 **Who Tests?**
    - Different people at different stages:
        - Developers → Unit Testing
        - Testers → Integration/System Testing
        - Users → Acceptance Testing

5. 🧪 **Different Strategies at Different Levels**
    - **Unit Testing** → Test small functions/modules
    - **Integration Testing** → Test how modules work together
    - **System Testing** → Test the whole system
    - **Acceptance Testing** → Final check by users/customers

6. 📏 **Testing Metrics**
    - Used to **measure** how testing is going
        - Number of test cases passed/failed
        - Defect density
        - Test coverage
        - Time spent per test

7. 🏢 **Organizational Policies Matter**
    - Some companies prefer **developers to test their own code**
    - Others assign **dedicated testers**
    - Some do both (mixed team)

8. 🤖 **Manual + Automated Testing**

- **Manual Testing** → Human performs the tests
- **Automated Testing** → Scripts or tools run the tests
  Best practice: Use a combination of both depending on the context

---

🔁 **Quick Summary:**

Concept	Description
Testing Goal	Find bugs and verify software quality
Starts Early	Begins with project planning
Who Tests	Developers, Testers, Users
Test Levels	Unit, Integration, System, Acceptance
Metrics	Help track testing progress
Manual vs Automated	Combine both for best results
Policies	Affect how testing is organized in a company

Want me to continue with the next slides from this chapter, or prepare quick revision notes/flashcards for this one?

🧠 PRINCIPLES OF TESTING

These are **fundamental truths** that guide how we approach testing.

🔹 Principle 1 – Testing shows presence of defects, not absence

We test to find **bugs**, not to **prove perfection**.

- Even if your tests pass, it doesn't **guarantee** that your software is bug-free.
- There's always a chance of **undetected bugs**.

📄 **Example:**
You tested your app 100 times and found 0 issues — but maybe you didn't test **all** scenarios. Bugs could still be hiding.

🔹 Principle 2 – Exhaustive testing is impossible

You **can't test everything** — not all input combinations, all devices, all browsers, etc.

- Instead, **focus** on the **most critical and risky areas**.
- Use **risk-based testing**, **priorities**, and **cost/time analysis**.

📜 **Example:**
Testing a login form: You can't test **every** combination of input (e.g., 5000+ types of invalid emails). Focus on key ones.

🔹 Principle 3 – Early testing

The earlier you find a bug, the **cheaper and easier** it is to fix.

- Start testing **from the beginning** of the project — even during **requirements** and **design**.

📜 **Example:**
Finding a bug in a plan is much cheaper than fixing it after the full app is built.

🔹 Principle 4 – Defect clustering

Most bugs are found in **a few specific areas** of the system.

- 80% of bugs are often in **20% of the modules** (Pareto Principle).
- Focus more testing effort on **those risky areas**.

📜 **Example:**
If bugs are always found in the payment module, test it harder — it's a hotspot.

🔹 Principle 5 – Pesticide paradox

Running the **same tests again and again** eventually stops finding bugs.

- You need to **change** or **add new test cases** regularly to find **new bugs**.

📜 **Example:**
Like insects become resistant to pesticides, software can seem clean if you use only the same tests over time.

🔹 Principle 6 – Testing is context dependent

Different types of software need **different types of testing**.

- A **medical device app** needs stricter testing than a **blog site**.
- You must **adapt your testing** based on the **project type**.

📜 **Example:**
Testing for a **banking app** focuses more on **security** and **accuracy** than a social media app.

🔹 **Principle 7 – Absence-of-errors fallacy**

Fixing all bugs doesn't mean the software is **useful or successful**.

- The software must also **meet user needs** and be **usable**.
- A **bug-free product** that nobody wants is still a failure.

📜 **Example:**
If your app has no bugs but no one can figure out how to use it — that's a big problem.

🔁 **Quick Summary Table:**

Principle	**Main Idea**
1. Presence ≠ Absence	Testing finds bugs, but can't prove they're all gone
2. Exhaustive ≠ Practical	Can't test everything — prioritize wisely
3. Early Testing	Find bugs early when they're cheap to fix
4. Defect Clustering	Most bugs are found in a few specific places
5. Pesticide Paradox	Repeating same tests won't find new bugs — update test cases regularly
6. Context Dependent	Testing approach depends on the type of software
7. Absence-of-errors Fallacy	No bugs ≠ usable or successful product

TESTING LEVELS

Software testing is done in **different phases**, depending on the **scope** and **who performs it**.

🔹 1. Unit Testing

👨‍💻 **Done by:** Developer
🧠 **Focus:** Testing individual components like functions, methods, or classes in isolation.

- The **smallest testable part** of the software.
- Ensures **each piece works correctly** on its own.

📜 **Example:**
Testing a calculateTotal() function in a shopping cart module — before integrating it with the full cart system.

🔹 2. Integration Testing

👨‍🔬 **Done by:** Tester
🧠 **Focus:** Test how **modules work together**.

- After unit testing, developers combine modules and test **interactions** between them.
- Catches issues like **data mismatches**, **API miscommunications**, etc.

📜 **Example:**
Testing whether the login module correctly passes data to the dashboard module.

🔹 3. System Testing

👨‍🔬 **Done by:** Tester
🧠 **Focus:** Test the **complete system** as a whole.

- Covers **functionality**, **performance**, **security**, **UI/UX**, etc.
- Mimics a **real-world environment**.

📜 **Example:**
Simulating a full user journey from account creation to checkout in an e-commerce site.

🔹 4. Acceptance Testing

👨‍⚖️ **Done by:** End-User / Customer
🧠 **Focus:** Does the system meet **customer's expectations**?

- Validates the system **from a user's perspective**.
- Sometimes called **User Acceptance Testing (UAT)**.

📄 **Example:**
A client tests your developed school management app to verify if it works the way they wanted.

🔁 **Regression Testing**

Ensures that **new changes** didn't **break anything** in the old code.

- No new test cases are designed — instead, **existing test cases** are **re-run**.
- It's done whenever you **add new features**, **fix bugs**, or **make enhancements**.

📄 **Example:**
After fixing a bug in the search feature, you re-run older tests to make sure the homepage and cart still work fine.

👷 **Testing vs. Debugging**

Understanding the difference between **testers and developers** is crucial:

Activity	Tester	Developer
🎯 **Goal**	Find bugs	Fix bugs
👓 **Perception**	Destructive (break the system)	Constructive (build the system)
🎨 **Known/Unknown**	Works like an **external user** (unknown system)	Knows how the system works (internal structure)
🔁 **Work**	Plans and executes tests	Writes code, debugs, and fixes issues
🏆 **Success means**	Finding the most bugs	Creating a working, bug-free product

🔁 **Tester:**

- Plans and executes **test cases**
- Focus: **Finding faults** that the developer missed

🔁 **Developer:**

- Writes and debugs code
- Focus: **Fixing faults** and ensuring the product works

📄 **Example:**
A tester reports that clicking “Add to Cart” crashes the app.
The developer then **debugs the code** and finds a missing null-check in the item object.

TESTING KEY QUESTION FRAMEWORK

This section answers **why**, **how**, and **what kind** of testing we do — and when we can **stop** testing.

❓ WHY do we test?

1. ✅ **Demonstrate proper behavior or quality**
   - Show that the software behaves as **expected** and meets its **requirements**.
2. 🐞 **Detect and remove defects**
   - The main goal is to **find bugs early** and ensure **defect-free software** before release.

📄 **Example:**
Testing ensures your online banking app **doesn’t crash** when transferring money and performs **accurately and securely**.

🔧 HOW do we test?

1. 🔍 **Using techniques, processes, and tools**
   - This includes **manual testing**, **automated scripts**, **unit tests**, etc.
2. 📋 **Generic Testing Process:**

Step	Description
Test Planning & Preparation	Define what to test, how, and with what data
Test Execution	Run the test cases and log results
Analysis & Follow-up	Check the results, fix issues, and re-test

📄 **Example:**
You plan test cases for a login system, run them, find that the “Forgot Password” feature fails — fix it, and test again.

👁️ **VIEW – What kind of testing approach?**

1. 🧊 **Black-box (Functional/External)**
    - Tester doesn’t see the internal code; focuses on **inputs and outputs**.

📄 *Example:* Checking if entering valid credentials logs the user in.

2. 🔓 **White-box (Structural/Internal)**
    - Tester knows the code and tests **internal paths**, loops, and logic.

📄 *Example:* Ensuring all branches in an if-else block are executed during tests.

3. ⚪ **Gray-box (Mixed)**
    - Tester has **partial knowledge** of internal logic and uses both input-output and code insight.

📄 *Example:* Testing form validation knowing there’s both frontend and backend validation rules.

🏁 **EXIT – When to stop testing?**

- **Functional Coverage (White-box):**
  Stop when all code paths or branches are tested (e.g., 100% branch coverage).
- **Usage-based/Quality Goals:**
  Stop when you've achieved the desired **reliability, performance, or user experience** level.

📄 *Example:*
If 95% of real-user scenarios are tested successfully, and system uptime is at target level, testing may be stopped.

🧪 **TESTING TECHNIQUES**

There are two major types of testing techniques — **White-box testing** and **Black-box testing**. Both serve different purposes and are used at various stages of the development process.

☐White-box Testing (Implementation-based / Structural Testing)

- **Focus:** This type of testing focuses on the **internal workings** of the software, particularly the **source code**.

Key Areas in White-box Testing:

- **Control Flow:**
  - This refers to how control (or instructions) **flows** from one part of the code to another.
  - **Example:** A simple if condition in the code will control which block of code gets executed based on a decision.
- **Data Flow:**
  - This involves **tracking** how data is passed around and used between variables or constants in the program.
  - **Example:** How a value in one variable (like x) is transferred or modified in another variable (like y).

Test Case Goals in White-box Testing:

1. ✅ **Exercise all independent paths:**
   - Ensure that all possible execution paths through a module (or function) are tested at least once.

📄 *Example:* If there are multiple decision points in your code, ensure each path is tested, like both "yes" and "no" outcomes from if conditions.

2. ✅ **Exercise all logical decisions:**
   - Test both the **true** and **false** outcomes of logical decisions (like if statements).

📄 *Example:* Check what happens when a value is greater than a threshold and when it's less.

3. ✅ **Loop Testing:**
   - Ensure that all **loops** in the code are tested at their **boundaries** and **within normal bounds**.

📄 *Example:* If a for loop runs 10 times, ensure it's tested for 0 iterations (boundary), and 10 iterations (normal condition).

4. ✅ **Internal Data Structures:**
    - Verify that internal data structures (like arrays or linked lists) work correctly.

📜 *Example:* If using an array, check that it can correctly store, retrieve, and update values.

2️⃣ Black-box Testing (Specification-based / Functional Testing)

- **Focus:** Black-box testing examines the **external behavior** of the software without needing to know the internal code structure.

Key Areas in Black-box Testing:

- You only test what is **visible** at the **external interface level**, meaning inputs are given to the program, and outputs are observed.
- **Example:** Testing a login form where you give valid and invalid credentials and see whether the system correctly logs the user in or displays an error message.

Error Categories Black-box Testing Focuses On:

1. 🚫 **Incorrect or Missing Functions:**
    - Check whether all **required features** are implemented correctly.

📜 *Example:* A shopping cart checkout feature should correctly calculate the total price and apply discounts if necessary.

2. 🚫 **Interface Errors:**
    - Ensure the software interacts correctly with other systems or users (like APIs or databases).

📜 *Example:* A "submit" button should properly submit a form, and not leave the user stuck.

3. 🚫 **External Database Access Errors:**
    - Make sure the software **accesses** external databases properly and returns accurate data.

📜 *Example:* A website displaying product prices from an external database should update correctly without errors.

4. 🚫 **Behavior or Performance Errors:**
    - Test whether the software behaves as expected in various conditions and performs optimally.

📜 *Example:* A mobile app should not crash or lag when opening a heavy image gallery.

5. 🚫 **Initialization and Termination Errors:**
    - Verify that the software **starts up** and **shuts down** correctly.

📄 *Example:* A program should not leave open files or memory leaks when closed.

Summary:

- **White-box testing** looks **inside** the program and checks for logical errors in the code structure (like flow control, loops, and data handling).
- **Black-box testing** checks what is **visible to the user**, ensuring that the software works correctly from an **external perspective** without worrying about how it's coded.

Both techniques are essential, and they complement each other in ensuring software quality. Let me know if you'd like more detailed examples or further clarification!

WBT vs. BBT: Key Differences

🔲Perspective:

- **White-box Testing (WBT):**
    - Views the software as a **glass-box**. The internal workings of the system (such as code and logic) are visible and tested.
    - Testers have access to the source code and can design tests based on the structure and logic of the system.
- **Black-box Testing (BBT):**
    - Views the software as a **black-box**. Testers focus on the **input-output behavior** without knowing the internal details of the system.
    - Tests are designed based on the **requirements and specifications** of the software, focusing only on what the system does, not how it does it.

2️⃣Objects Tested:

- **WBT:** Typically used for testing **smaller units** or specific components of a system, like individual modules or functions.
    - **Example:** Testing a single function in a program to ensure it behaves as expected based on the code.

- **BBT:** Used for testing **larger systems** or substantial parts of the system as a whole, where the interactions and overall functionality need to be evaluated.
    - **Example:** Testing a complete website to ensure it performs all its expected functions, like logging in or adding items to a shopping cart.

3️⃣ Timeline of Use:

- **WBT:** More commonly used in the **early stages** of the development cycle, such as during **unit testing** and **component testing**.
    - Focuses on individual components to catch issues early in the coding process.
- **BBT:** Used later in the development cycle, such as during **system testing** and **acceptance testing**.
    - Focuses on the software as a whole to ensure that it meets the customer's needs and behaves as expected in real-world usage.

4️⃣ Defect Focus and Fixing:

- **BBT:** Focuses on detecting defects related to **external functions** (what the software does for the user). Failures in this area are easier to observe, and the system's interactions with external systems or the user are examined.
    - **Key Focus:** Ensures that the software delivers the expected functionality to the customer.
- **WBT:** Focuses on detecting defects in the **internal implementation** of the software, such as issues in the logic, control flow, or data handling.
    - **Key Focus:** Ensures that the software is robust internally, reducing the chances of failures in the later stages.

5️⃣ Defect Detection & Fixing Efficiency:

- **WBT:** Defects found through white-box testing are typically **easier to fix** because they are related to specific internal issues in the code.
    - However, WBT may **miss certain defects** like **design issues** or **missing functionality** that BBT is better suited to catch.
- **BBT:** While BBT can be effective in detecting problems with the external functionality and user interfaces, it may **miss internal issues** that affect performance or functionality, which would be detected by WBT.

- **In summary:**
  - **BBT** is better for detecting issues related to **interfaces, interactions, and user behavior**.
  - **WBT** is better for catching issues within **smaller units** and focusing on logic and code-related problems.

6️⃣ Techniques:

- **BBT:** Testing is performed by modeling the **external functions** of the system. This involves creating test cases based on the system's **functional requirements** and **expected outputs**.
- **WBT:** Testing involves modeling the **internal implementation**, such as **control flow**, **data flow**, and logic. Test cases are based on the structure and **code** of the system.

7️⃣ Testers:

- **BBT:** Typically performed by **dedicated testers** or **third-party testers** (such as in independent verification and validation, IV&V). The focus is on functional behavior from an **end-user perspective**.
- **WBT:** Often performed by **developers** themselves, who test the software as they develop it, checking for internal issues and ensuring the code meets its specified requirements.

Summary:

- **White-box testing (WBT)** focuses on testing the internal structure and logic of software, typically during the early stages of development. It's used to identify defects within small code units and is often done by the developers themselves.
- **Black-box testing (BBT)**, on the other hand, focuses on testing the software from the user's perspective, ensuring that it works as expected based on the requirements. It is used in the later stages of development and is often performed by dedicated testers.

Both testing types are complementary and help ensure that the software is both functionally correct and internally robust.

When to Stop Testing?

Exit Criteria:

- **Not finding defects** is not an appropriate reason to stop testing. **User Acceptance** is the final test that determines whether the product is ready.

Resource-Based Criteria (Not Ideal):

- **Stop when you run out of time or money**: This approach may lead to unfinished testing or insufficient quality assurance. Relying solely on resources for exit criteria can compromise product quality and lead to a poor user experience.

Quality-Based Criteria (Better Approach):

- **Stop when quality goals are reached**: Quality goals may include usability, reliability, and meeting actual customer needs. Testing should conclude when the product meets these quality standards, and it has been validated against its requirements.
- **Activity Completion**: Another substitute for exit criteria could be to stop testing when all planned test activities have been completed, ensuring that thorough testing has been conducted.

Manual Testing vs. Automated Testing

Manual Testing:

- **Oldest and Most Rigorous Type**: Manual testing requires human involvement to perform test operations, which makes it:
    - **Hard to Repeat**: Each test execution may differ based on the tester.
    - **Not Always Reliable**: Human error can occur.
    - **Costly and Time-Consuming**: Manual testing requires more effort and time.
    - **Labor-Intensive**: It requires significant human resources, especially for large-scale tests.

Automated Testing:

- **Testing with Software Tools**: Automated testing uses tools to execute tests without manual intervention, making it:
    - **Fast**: Automated tests can be run quickly, saving time.
    - **Repeatable**: Tests can be executed multiple times with consistency.
    - **Reliable**: Automated tests are less prone to human error.
    - **Reusable and Programmable**: Test scripts can be reused across different test cycles.

- **Saves Time and Costs**: After initial setup, automation reduces manual intervention and accelerates testing.

Key Issues Related to Test Automation:

- **Need for Automation**: Ensure that the system is stable and its functionality is well defined.
- **Tool Selection**: Choose the right tools based on the testing needs and system requirements.
- **Training and Effort**: Consider the time and resources required for training testers to use the automation tools.
- **Overall Cost**: Including tool acquisition, training, and support.
- **Impact on Project Management**: The automation effort must align with resource allocation and the project schedule.

Pre-Requisites for Test Automation:

- **System Stability**: The system should be stable, with well-defined functionalities.
- **Clear Test Cases**: The test cases to be automated should be clear and unambiguous.
- **Test Tools in Place**: Proper automation tools and infrastructure should be available.
- **Experienced Test Engineers**: Automation requires skilled testers familiar with tools and processes.
- **Adequate Budget**: Ensure the budget covers the costs for procurement and maintenance of testing tools.

Which Tests/Test Cases to Automate?

- **Tests that should be run for every build** (e.g., regression testing).
- **Data-Driven Tests**: When the same test is executed with multiple data values.
- **GUI Attribute Testing**: Tests that need detailed internal information (like checking GUI components).
- **Stress/Load Testing**: Automated tests are useful for simulating heavy load on the system.

Automated Testing CANNOT Replace Manual Testing:

- **Usability Testing**: Can't be automated because it involves subjective human judgment.
- **Logical Errors**: Often require human intervention to identify and diagnose.

- **Specification/Design Documents**: Automation cannot test design documents or specifications.
- **One-Time Testing**: Testing that is needed urgently or for a unique, time-sensitive issue ("ASAP").
- **Ad-Hoc/Random Testing**: When tests are based on intuition or experience, they can't always be predefined or automated.

Summary:

- **Manual Testing** is essential for tasks that require human judgment and flexibility, such as usability testing, and it's necessary for certain unique or one-time tests.
- **Automated Testing** excels in repetitive tasks and scenarios that require consistent and efficient execution, such as regression and data-driven testing. It helps save time and improve reliability, but it can't replace manual testing for subjective assessments or highly dynamic test situations.