**Chapter 5**

**Test Plan And Test Case**

Test Plan:

 A test plan is a document that explains how software testing will be done.

 It includes goals, what to test, how to test, who will do it, and the time needed.

 It helps the team understand the testing process clearly.

 The plan gives all important details before testing starts.

 Companies may use standard formats or create their own test plan style.

Test Case:

 A test case shows what to test, how to test, and the expected result.

 It checks if a feature works properly.

 It includes input, action, and expected output.

 Test case formats can vary by company.

 **Note:** Test plan is high-level; test case is low-level.

**Explain QA activities**

**1. Pre-QA (Planning)**

* Set quality goals (like speed, reliability).
* Understand user expectations.
* Choose measurable quality targets.
* Balance quality with cost.
* Plan QA strategy and test methods.
* Prepare and organize test cases.

**2. In-QA (Test Execution)**

* Run tests as planned.
* Find and record bugs.
* Document test results using templates.

**3. Post-QA (Review & Improve)**

* Measure quality results.
* Analyze what worked or failed.
* Suggest improvements for future testing.

### ****Preparing a Test Suite (Macro-level)****

* A **test suite** is a group of test cases run together in a sequence.
* It’s built by organizing test cases using specific testing methods.
* Test cases from older versions can be reused — this is called **regression testing**.
* All test cases, new or reused, should work together as one complete test suite.

### ****Testing Teams: Organization & Management****

* **Customers/Users**: May help with usability or beta testing.
* **Independent Testers**: Hired third-party testers to provide unbiased results.
* **Team Structures**:
  + **Vertical Model**: Testers focus on one product.
  + **Horizontal Model**: Testers focus on one type of testing across many products.
  + **Mixed Model**: Combines both — used in large companies.

**Chapter 6(A)**

### ****Testing****

* A key part of QA that involves running the software to check how it behaves.
* If something goes wrong (a failure), the issue is investigated and fixed.
* If everything works, it increases confidence that the software does what it’s supposed to.

### ****Testing the System****

* Testing is a separate process with its own methods and strategies.
* It helps find defects and checks quality like reliability and performance.
* Starts early in the project and continues through development.
* Done by different people at different stages.
* Uses various techniques and tracks progress with metrics.
* Can be manual, automated, or both.
* Organizational policies affect how testing is done.

### ****7 Principles of Testing****

 Testing shows presence of defects but cannot prove absence of defects

 Exhaustive testing is impossible

 Early testing

 Defect clustering

 Pesticide paradox

 Testing is context dependent

 Absence-of-errors fallacy

1. **Defects exist**: Testing can show bugs are present, but never prove there are none.
2. **Can’t test everything**: Full testing isn’t possible—focus on high-risk areas.
3. **Start early**: The earlier you test, the easier and cheaper it is to fix issues.
4. **Defects cluster**: Most bugs are found in a few problem areas—focus there.
5. **Pesticide paradox**: Same tests stop finding new bugs—update tests regularly.
6. **Context matters**: Different types of systems need different testing approaches.
7. **No errors ≠ good software**: Bug-free doesn’t mean it meets user needs.

Testing Levels:

### ****1. Unit Testing****

* Done by developers
* Tests small parts of code (like functions) one at a time

### ****2. Integration Testing****

* Done by testers
* Checks if different parts/modules of the software work well together

### ****3. System Testing****

* Done by testers
* Tests the whole system to see if it works as expected (e.g., features, performance)

### ****4. Acceptance Testing****

* Done by end-users
* Checks if the software meets the user’s needs and expectations

### ****5. Regression Testing****

* Makes sure that new changes haven’t broken anything that worked before
* Uses old test cases (no need to create new ones)

### ****Testing vs. Debugging****

* **Testing**: Checking the software to find defects. It's a repeated activity.
* **Debugging**: Fixing defects found during testing. Done by developers.

### ****Developer Roles & Responsibilities****

* Implements requirements and builds software.
* Designs and programs the system.
* Success is creating a working product.
* Focuses on delivery.

### ****Tester Roles & Responsibilities****

* Plans and designs testing activities.
* Focuses on finding defects and errors in the software.
* Success is finding issues before release.
* Focuses on quality, from the user’s perspective.

### 🔹 ****TESTING KEY QUESTION****

#### ✅ **WHY**

* Demonstration of proper behavior or quality
* Defect-free software development (defect detection and removal)

#### ✅ **HOW**

* Techniques, activities, process, etc.
* Generic testing process:
  + Test planning and preparation
  + Test execution
  + Analysis and follow-up

#### ✅ **VIEW**

* Functional / external / black-box
* Structural / internal / white-box
* Gray-box (mixed black-box & white-box testing)

#### ✅ **EXIT**

* Functional coverage (white-box)
* Usage-based: quality / reliability goals

Testing Techniques

* High-level: Whole system ==> black-box (late in testing, e.g. system testing)
* Low –level: Individual statements, data, and other elements ==> white-box (test in small/early)
* Middle-levels of abstraction ==> Gray-box

### ****1. Black-box Testing****

* **What it tests**: Focuses on **inputs and outputs**. It doesn’t look at the code.
* **Example**: Testing a login page. You input a username and password, and the software either lets you in or shows an error. You don't care how it works behind the scenes, just that it works.

### ****2. White-box Testing****

* **What it tests**: Focuses on the **internal code** (how the software works).
* **Example**: Testing a function that checks if a number is prime. You’d check that the logic works, test all possible paths (like when the number is 1, 2, or a large number), and make sure it works for every condition.

### ****3. Gray-box Testing****

* **What it tests**: A mix of **black-box** and **white-box** testing. You check both the software's behavior and some internal parts.
* **Example**: Testing an online store’s checkout process. You test the user interface (black-box), but also check how the payment gateway connects to the backend (white-box).

WBT VS BBT

**1. Perspective**

**BBT**: Focuses on testing **inputs and outputs** without looking at internal code.  
**WBT**: Focuses on testing **internal code and implementation details**.

**2. Objects Tested**

**BBT**: Used for **large systems** or the **whole software**.  
**WBT**: Used for **small parts** or **individual units** of software.

**3. Timeline**

**BBT**: Used in **later stages**, like system and acceptance testing.  
**WBT**: Used in **early stages**, like unit and component testing.

**4. Defect Focus and Fixing**

**BBT**: Focuses on fixing **external functional issues** that customers will see.  
**WBT**: Focuses on fixing **internal implementation issues** before they affect the software.

**5. Defect Detection and Fixing**

**BBT**: Detects **interface issues** and functional problems.  
**WBT**: Detects **internal logic issues**, but may miss design or omission problems that BBT would catch.

**6. Techniques**

**BBT**: Focuses on **external functions** and how they behave.  
**WBT**: Focuses on the **internal implementation** of code.

**7. Tester**

**BBT**: Usually done by **dedicated testers** or **third-party testers**.  
**WBT**: Often done by **developers** who wrote the code.

**When to stop testing?**

* **Not finding defects**: Don't stop just because no defects are found. **User acceptance** matters most at the end.
* **Resource-based**: Stopping when time or money runs out isn’t ideal for quality.
* **Quality-based**: Stop when **quality goals** (like usability and reliability) are met.
* **Activity completion**: Stop when all planned tests are done.

Manual VS Automated Testing

| **Manual Testing** | **Automated Testing** |
| --- | --- |
| Done by human testers | Done using software tools |
| Slow | Fast |
| Hard to repeat | Easily repeatable |
| Less reliable | More reliable |
| Costly | Saves cost over time |
| Time-consuming | Time-saving |
| Labour intensive | Less manual effort |
| Not reusable | Reusable test scripts |
| No programming needed | Requires programming for test scripts |

**Test Automation**

### 🔑 ****Key Points****

* Pick tools that fit the testing needs
* Make sure testers are trained
* Consider total cost: tools, training, support
* Check if tools affect schedule/resources

### ✅ ****When to Automate****

* System is stable and clear
* Test cases are clear and repeatable
* Tools and setup are ready
* Testers have automation experience
* Budget is available

### 🤖 ****What to Automate****

* Repeated tests (e.g., regression tests)
* Tests with many data sets
* Tests needing app internals (like GUI info)
* Load/stress tests

### ❌ ****What NOT to Automate****

* Usability tests (e.g., user-friendliness)
* One-time or urgent tests
* Random/ad hoc tests
* Tests with unclear results
* Design/spec document checks

**Chapter 6(B)**

**Code Review Process**

**What is Code Review?**

It's a process where team members check each other’s code to find mistakes and suggest improvements.

**Steps with Examples**

**1. Readiness**  
The code writer checks if the code works and is easy to read.  
*Example:* Aisha writes a grade calculator and makes sure it runs correctly.

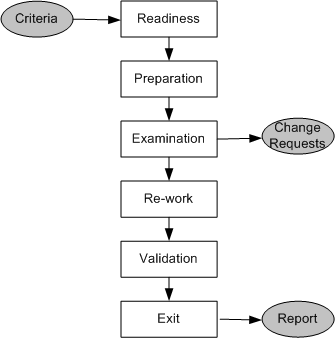
**2. Preparation**  
Reviewers read the code and note questions or improvements.  
*Example:* Rafiq checks Aisha’s code and wonders what happens if no grades are entered.

**3. Examination**  
They meet to discuss the code and note any issues.  
*Example:* In the meeting, Rafiq suggests fixing the empty input issue.

**4. Re-work**  
The code writer fixes problems and makes improvements.  
*Example:* Aisha updates her code to handle missing grades.

**5. Validation**  
Someone checks if the updates work.  
*Example:* Rafiq tests the new code and confirms it works.

**6. Exit**  
A summary of changes is shared with the team.  
*Example:* The team leader sends an update that Aisha’s code is ready.



### ****Change Request (CR)****

A **Change Request (CR)** includes:

1. **Issue Description**: Brief explanation of the problem.
2. **Priority Level**: Major or minor.
3. **Assigned Person**: Who will handle it.
4. **Deadline**: When it should be fixed.

### ****Dynamic Unit Testing****

Execution-based unit testing is called dynamic unit testing.  
A program unit runs by itself, and we watch the results.  
The unit's environment is copied and tested alone.  
The IDE checks each line for errors while coding.  
The caller unit is called a test driver.  
A test driver runs the unit under test (UUT).  
It gives input to the UUT and shows the result.  
Units called by the UUT are copied as stubs.  
A stub is a small program that stands in for a called unit.  
The test driver and stubs together are called scaffolding.  
The low-level design document helps choose test input.

### 🧪 Dynamic Unit Testing

**🔁 Control Flow Testing**

* Draw a flow chart of the code (CFG).
* Pick a path to test.
* Make input that follows that path.

**🔄 Data Flow Testing**

* Track how data moves in the code (DFG).
* Test like control flow.
* Example: Grade calculation in VUES.

**🗂️ Domain Testing**

* Checks for wrong paths caused by certain inputs.
* Example: Leave app —
  + 1–5 days → Team Lead
  + 6–10 days → PM

**⚙️ Functional Testing**

* Test input-output match.
* Example: 90 marks → A+ in VUES.

### ✅ What is Dynamic Unit Testing?

Dynamic unit testing means **running a program unit (like a function or module)** to see if it gives the **correct result**.

* The unit is tested **alone** (in isolation).
* We **watch the output** when we give it different inputs.
* The parts around the unit (like other functions it calls) are **faked or replaced**, so we can test just this one unit.

### 📌 Important Parts:

* **Test Driver**: A small program that **calls the unit under test (UUT)** and gives it input.
* **Stub**: A fake or simple version of a unit that the UUT **calls**.
* **Scaffolding**: The **test driver and stubs together** used for testing.
* **Low-level design document**: Helps pick good test input values.

### 🧪 Example: Login Button on a Website

* You are testing the **login** function (the unit).
* You use a **test driver** to enter a username and password.
* The real server is not ready, so a **stub** gives a fake reply: “Login OK”.
* You check if the login works.

If it shows “Welcome”, the test **passes**.  
If not, it **fails**.

👉 This is **dynamic unit testing** because the code runs with real input.

### ****Dynamic VS Static Unit Testing****

### 🔄 ****Dynamic Unit Testing****

* **Code is executed** during testing.
* Finds **runtime errors**, logic issues, and incorrect outputs.
* Needs a **test driver**, **stubs**, and sometimes **real inputs**.
* Example: Running a login function to check if it accepts correct credentials and blocks wrong ones.

### 🧾 ****Static Unit Testing****

* **Code is not executed**.
* Done by **reading**, **reviewing**, or using **tools** to check the code.
* Finds **syntax errors**, **style issues**, or **possible bugs** early.
* Example: Using an IDE or tool to highlight missing semicolons or unused variables while coding.

### 📊 Key Differences:

| **Feature** | **Dynamic Unit Testing** | **Static Unit Testing** |
| --- | --- | --- |
| Code execution | Yes | No |
| Catches | Logic & runtime errors | Syntax, style, and early errors |
| Tools used | Test drivers, stubs, scaffolding | IDE, linters, code analyzers |
| When done | After writing the code | While or just after writing the code |
| Example | Running a function with inputs | IDE shows error in undeclared variable |

### ****Debugging Techniques****

Fixing defects found during testing. Done by developers.

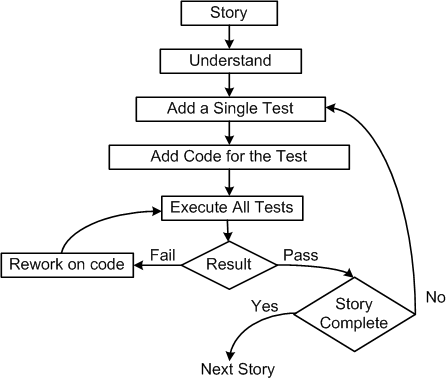
Identify the error and potential cause of error (hypothetical hidden errors)

1. **Brute Force Testing**
   * Most common but least efficient.
   * Involves using memory dumps, run-time traces, and output statements to find errors.
2. **Backtracking**
   * Tracing the code backward manually to locate the error.
   * Works well for small programs.
3. **Cause Elimination**
   * Create a "cause hypothesis" and test it.
   * Refine tests to isolate the bug if the initial hypothesis seems correct.

### ****Unit Testing in Extreme Programming (XP)****

In **Extreme Programming (XP)**, unit testing is closely integrated with the development process. Here's how it works:

1. **Pick a Requirement**: Choose a user story to work on.
2. **Write a Test Case**: Create a test that checks a small part of the requirement and set it to fail initially.
3. **Write Code**: Implement the code to make the test pass.
4. **Execute Tests**: Run all tests to check if the code works.
5. **Rework Code**: Adjust the code and run tests until all pass.
6. **Repeat**: Continue this process until the user story is fully implemented.



### UNIT TESTING

* **Mutation:** A small, allowed change in the code.
* **JUnit:** A tool to test Java programs.

