**UNIT TESTING**

**Definition:**

* Unit Testing is a way to test *one small piece* (or *unit*) of software, like a single function or method in a program.
* This is usually done by the programmer *before* combining all units together (before **Integration Testing**).

**Example:**

* Imagine you're building a calculator app. You'd test just the **add()** function first to make sure it works on its own, like add(2, 3) should return 5.

**Performed by the Individual Programmer**

* The person who wrote the code is usually the one who tests it.
* This helps catch and fix errors early, while the code is still fresh in their mind.

**Uses White Box Testing**

* **White Box Testing** means the tester knows how the code works inside.
* So the programmer tests all possible paths, conditions, and logic inside the code.

**Example:**

* If there’s an if statement inside the function, you make sure to test both the if and the else parts.

**Lower Level / Initial Level Testing**

* Unit testing happens early in the software testing process.
* It's the first level of actual code testing.

**Types of Unit Testing:**

**1. Static Unit Testing**

* This checks the code **without actually running it**.
* Focus is on reviewing the code, checking for syntax errors, or using tools to find bugs.

**Example:**

* A programmer reading through the function to spot mistakes.
* Or using tools like linters or static analyzers.

**2. Dynamic Unit Testing**

* This **runs** the code to see if it behaves as expected.
* Tests the actual output for given inputs.

**Example:**

* Running add(2, 3) and checking if the result is 5.

**Static and Dynamic Analysis are Complementary**

* Both methods work **together** to make the testing more complete.
* Static testing catches mistakes without execution, while dynamic testing confirms behavior during execution.

**Recommendation**

* Do **static testing first** to clean up obvious errors.
* Then do **dynamic testing** to verify that the code behaves correctly.

**STATIC UNIT TESTING (REVIEW CODE)**

**What is it?**

* Static unit testing means checking the code **without running it**.
* It's like proofreading your code — you’re looking at the code itself to find problems before even pressing “Run”.

**Code is examined at compile time**

* This happens **before** the code is actually executed.
* The goal is to catch problems that could happen **when** the code runs.

**Example:**

* You might spot a logic error like dividing by zero or missing a return statement — just by looking at the code.

**Code is validated against requirements**

* You check: **Does the code do what it’s supposed to?**
* The programmer or a reviewer compares the code with the **unit’s requirements**.

**Example:**

* If a function is supposed to calculate a discount, you check whether the logic matches what the business rules say.

**Techniques Used in Static Unit Testing:**

**🔹 Walkthrough (Informal)**

* The programmer walks others (like team members) through the code.
* It's a casual, informal review.

**Example:**

* “Here’s my function — it takes the price and applies the discount. Let me show you how I’ve written it...”

**🔹 Code Inspection (Formal)**

* A more **structured** and **formal** review.
* A team checks the code based on a checklist (e.g., naming conventions, logic errors, consistency).

**Example:**

* A reviewer might go: “Line 12 doesn’t handle negative values — that’s a problem based on the requirements.”

**Black-box and White-box Testing in Static Testing?**

* Even though **static testing** doesn’t run code, you can still use both **views** to check:
  + **Black-box view**: Focuses on what the code is supposed to do (without caring how it works inside).
  + **White-box view**: Focuses on the actual structure and logic inside the code.

**Example:**

* Black-box: “The function should return a valid discount.”
* White-box: “Let me check if the if-else conditions inside the function cover all cases.”

**DYNAMIC UNIT TESTING (EXECUTE CODE)**

**What is it?**

* This type of testing happens **when the code is actually run**.
* You give input to the code, run it, and check if the output is correct.
* It helps you find *run-time errors* — the kinds of problems that only show up when the program is running.

**Execute at Run Time**

* Unlike static testing (which is done just by looking at the code), dynamic testing runs the program to **see how it behaves**.

**Example:**

* You run a function like calculateTax(5000) and check if the result is correct — say it should return 250.

**Black-box Testing**

* In dynamic unit testing, **black-box testing** is often used.
* This means: you **don’t look at the code inside**, you only care about **input and output**.

**Example:**

* You test a login function:
  + Input: username = "user1", password = "pass123"
  + Expected output: "Login successful"
  + You don't care how the function checks the password — only that the result is correct.

**🔁 Quick Comparison (for clarity):**

|  |  |
| --- | --- |
| **Static Unit Testing** | **Dynamic Unit Testing** |
| Code is **not executed** | Code is **executed** |
| Focuses on code structure | Focuses on behavior/output |
| Done at compile time | Done at run time |
| Techniques: Walkthroughs, inspections | Techniques: Test cases, actual runs |

**🔍 STEPS IN THE CODE REVIEW PROCESS**

Code review is a team activity where the written code is examined before it goes live. The main goal is to **catch bugs, improve quality, and share knowledge**. Here's how it works step-by-step:

**✅ Step 1: Readiness**

* The **author (programmer)** checks if the code is ready to be reviewed.
* The code should meet certain conditions like:
  + ✔ **Completeness** – the function/module does what it’s supposed to
  + ✔ **Minimal functionality** – unnecessary code is avoided
  + ✔ **Readability** – clean and easy to understand
  + ✔ **Complexity** – not too complicated
  + ✔ Matches the **Requirements and Design documents**

**Think of it like:** "Is my code clean, complete, and easy for others to understand?"

**📝 Step 2: Preparation**

* **Reviewers** get ready by:
  + Writing down **questions** about the code
  + Listing possible **Change Requests (CRs)** – things that might need fixing
  + Suggesting **improvements** – better ways to do things

**👀 Step 3: Examination**

* The actual **code review meeting** happens. Everyone has roles:

| **Role** | **What they do** |
| --- | --- |
| **Author** | Presents the code and explains it |
| **Reviewer** | Reads the code and finds issues or improvements |
| **Record Keeper** | Writes down all CRs and key notes from the meeting |
| **Moderator** | Keeps the meeting on track |

**🔧 What is a CR (Change Request)?**

A **CR** is a formal request to change something in the code.

Each CR includes:

* A short **description** of the issue
* A **priority** (e.g., *Major* or *Minor*)
* Who will **fix it**
* A **deadline** for fixing it

**Example:**

CR: "Function doesn't handle empty input" Priority: Major Assigned to: John Deadline: 2 days

**🔄 Step 4: Re-work**

* After the meeting, the **Record Keeper** prepares a **summary**.
  + All CRs are listed
  + Any **types of maintenance** needed are noted:
    - **Corrective** (fixing bugs)
    - **Perfective** (improving code)
    - **Adaptive** (adjusting to environment)
    - **Preventive** (avoiding future issues)
* Then the **author** goes back and fixes the issues.

**✅ Step 5: Validation**

* Once changes are made, they are **independently checked**.
* This may include **regression testing** to make sure old features still work.

**📤 Step 6: Exit**

* A **final summary report** is shared with:
  + Everyone who **requested** the review
  + People who will **work with or use** the reviewed code

**📊 CODE REVIEW METRICS**

**What are metrics?**  
Metrics are numbers or data used to **measure performance**, **efficiency**, and **cost** during the code review process.

These metrics help teams **track productivity**, **estimate budgets**, and **spot problems early**.

**💰 1. Identify the Cost of Development and Change**

This part is about understanding **how much it costs to write and review code**.

**📌 How is cost calculated?**

* **Cost per line of code** =  
  Developer’s hourly pay ÷ Number of lines they can write in an hour

**Example:**  
If a developer writes 100 lines/hour and is paid $50/hour:  
👉 Cost per line = $50 ÷ 100 = **$0.50 per line**

**📏 Lines of Code (LOC) reviewed per hour**

* Tells you how fast reviewers can go through code.
* Helps measure **reviewer efficiency**.

**Example:**  
If a reviewer reads 300 lines in 1 hour → 300 LOC/hour.

**⚠️ CRs per KLOC (Change Requests per 1000 Lines of Code)**

* Tells you how **many problems** are found in a certain amount of code.
* More CRs = potentially lower code quality

**Example:**  
If 10 CRs are found in 2000 lines of code →  
(10 ÷ 2) = **5 CRs per KLOC**

**🔍 2. Identify the Cost of Testing Process**

This focuses on how much time and effort is spent **finding and fixing issues** during reviews.

**⏱️ CRs generated per hour**

* Shows how many issues the team finds in each hour.
* More CRs/hour might mean lots of bugs… or very sharp reviewers!

**📅 Total hours spent on code review**

* Helps track how much **time (and money)** is being spent.
* Useful for budgeting and project planning.

**Example:**  
If a team spends 10 hours reviewing a module and finds 20 CRs:

* Cost of testing time = 10 hours × reviewer’s hourly rate
* Review efficiency = 2 CRs/hour

**Summary Table:**

|  |  |  |
| --- | --- | --- |
| **Metric** | **What it Measures** | **Why It’s Useful** |
| Cost per LOC | Cost of writing code | Budgeting, resource planning |
| LOC reviewed/hour | Reviewer speed | Efficiency tracking |
| CRs per KLOC | Code quality | Bug density |
| CRs/hour | Testing effectiveness | Review productivity |
| Total review hours | Time cost | Planning and management |

**⚙️ DYNAMIC UNIT TESTING**

**✅ What is it?**

* Dynamic Unit Testing means **testing code by actually running it**.
* You focus on **one unit (like a single function or module)** at a time.
* You run it, give it input, and **observe the output** to see if it behaves correctly.

**Example:**  
You test a function like calculateInterest() by passing some values and checking if the result is correct.

**🔄 Unit Executed in Isolation**

* Each **unit (function or method)** is tested **separately**, not as part of the full program.
* You **simulate** its environment if needed.

**Example:**  
If calculateInterest() depends on a getRate() function, you can replace getRate() with a fake version (a **stub**) just for testing.

**🛠️ IDE Assistance**

* Many modern IDEs (like Visual Studio, Eclipse, or IntelliJ) help catch errors while writing code.
* They point out issues like syntax errors or type mismatches as you type.

**📞 Caller Unit: Test Driver**

* A **Test Driver** is a small program written just to **call and test** the unit under test (UUT).
* It sends **input data** and checks the **output**.

**Example:**  
If you're testing addNumbers(a, b), your test driver might look like:

python

CopyEdit

result = addNumbers(2, 3)

print("Test Result:", result)

**🧩 Stub: Fake Called Unit**

* A **Stub** is a dummy function that **replaces another unit** which is called by the unit you're testing.
* It's used when the real unit isn’t ready or isn’t needed for this test.

**Example:**  
If your unit calls a database, the stub just returns fake data instead of making a real DB call.

python

CopyEdit

def getRate(): # Stub version

return 0.05 # Fixed interest rate

**🧱 Scaffolding = Test Driver + Stubs**

* Both the **Test Driver** and **Stubs** are part of a test setup called **Scaffolding**.
* They create a temporary “test environment” around your unit.

**📄 Low-Level Design Document**

* This design document gives **details about each function**, what input it needs, and expected behavior.
* It helps in selecting **realistic test data** to test your unit properly.

**📌 Summary of Key Terms:**

|  |  |
| --- | --- |
| **Term** | **Meaning** |
| Unit | A small, testable part of the program (like a function) |
| Test Driver | A program that calls the unit and checks the result |
| Stub | A fake function that replaces a real one the unit depends on |
| Scaffolding | The setup (test driver + stubs) used to test the unit |
| Low-Level Design | Document that helps choose correct test inputs |

**⚙️ DYNAMIC UNIT TESTING – Types and Techniques**

When you run a unit of code to test it (dynamic testing), there are **different ways** to check how it behaves. These are techniques based on what you're focusing on — like control flow, data flow, input ranges, or outputs.

**🔁 1. Control Flow Testing**

**What it means:**

* You're checking **which paths** your program takes when different conditions are true or false.
* You use something called a **Control Flow Graph (CFG)** — a visual of all the paths your code can take (like if-else, loops, etc.).

**Steps:**

1. Draw a CFG showing all possible execution paths.
2. Choose which paths to test (criteria).
3. Identify test inputs that will **make the program follow those paths**.

**Example:**

python

CopyEdit

if marks > 90:

grade = "A+"

elif marks > 75:

grade = "A"

else:

grade = "B"

You want to test all 3 branches — so you pick:

* marks = 95 → A+
* marks = 80 → A
* marks = 70 → B

**🔄 2. Data Flow Testing**

**What it means:**

* Focuses on how **data (variables)** are used in the code.
* You create a **Data Flow Graph (DFG)** to track where variables are:
  + **Defined** (assigned a value)
  + **Used** (read or changed)

**Why it's useful:**

* Helps find issues like using a variable before it’s assigned.

**Example (Marks Grading):**

python

CopyEdit

marks = input()

if marks >= 90:

grade = 'A+'

You track how marks and grade are **defined and used**, and make sure all important combinations are tested.

**🌐 3. Domain Testing**

**What it means:**

* You test the **input ranges** (domains) and check if the program handles them correctly.
* A **domain error** happens when an input that should be valid causes the program to go in the wrong direction.

**Example (Leave Application):**

* If someone applies for leave:
  + 1–5 days → approved by **Team Lead**
  + 6–10 days → approved by **Project Manager**

You test:

* 3 days → should go to Team Lead ✅
* 8 days → should go to PM ✅
* 0 or 15 days → **invalid or misdirected** ❌ (test for errors)

**📥 4. Functional Program Testing**

**What it means:**

* This is basically **black-box testing**: You test if **input → expected output** works correctly.
* You don’t care *how* it works internally, just whether the **right result comes out**.

**Example (Grading System):**

* If input marks are 90, output should be “A+”
* If input is 85, it should be “A”

You define **input/output domains** and check if the unit returns correct values for each.

**🧠 Quick Summary Table:**

|  |  |  |
| --- | --- | --- |
| **Technique** | **Focus** | **Example** |
| Control Flow Testing | Test different execution paths | if-else, loops |
| Data Flow Testing | Track variables and their use | marks = input() |
| Domain Testing | Input ranges and boundaries | Leave approval by days |
| Functional Testing | Input → Output correctness | Grading system in VUES |

**🐞 DEBUGGING**

**💡 What is Debugging?**

* Debugging is the process of **finding and fixing errors (bugs)** in your code.
* You also try to **identify what might be causing the error**, even if the error is not visible yet — this is called finding **hypothetical or hidden errors**.

**🧰 DEBUGGING TECHNIQUES**

There are different strategies developers use to debug their programs. Let's look at the main ones:

**🔨 1. Brute Force Debugging**

**What it is:**

* The most common method, but also the **least efficient**.
* You add **print statements** or use tools to **dump memory** or **trace what the program is doing** during execution.

**How it works:**

* Print out values at various points to see where things go wrong.

**Example:**

python

CopyEdit

print("Value of x:", x)

print("Reached here")

**Use case:** Helpful when you're not sure where to start — just throw in output and see what's happening.

**🔁 2. Backtracking**

**What it is:**

* Mostly used in **small programs**.
* You start from the place where the error was noticed and **trace the code backward** to figure out what went wrong.

**How it works:**

* Look at the logic and variables **step-by-step in reverse**.

**Example:**

If your result is wrong, go backward and check where the input or calculations could have gone off-track.

**Use case:** Best when there are **not too many lines of code**, and you can manually trace it.

**❌ 3. Cause Elimination**

**What it is:**

* You come up with **hypotheses** (guesses) about what might be causing the bug.
* Then you **test those hypotheses** to see which one is the real issue.

**How it works:**

* If a test disproves one hypothesis, you eliminate it.
* Keep refining the test and input data to isolate the actual problem.

**Example:** Let’s say a function crashes only when input is between 10–20.  
You **assume** it’s a loop issue or array boundary error, and you **test different cases** to confirm that.

**Use case:** Useful for **complex bugs** where the cause is not obvious.

**📌 Summary Table:**

|  |  |  |
| --- | --- | --- |
| **Technique** | **How It Works** | **Best For** |
| Brute Force | Add print/logs, dump memory, trace output | Quick checks, any program |
| Backtracking | Manually trace code from the error point | Small programs |
| Cause Elimination | Make guesses, test and eliminate causes | Complex or hidden bugs |

**UNIT TESTING in EXTREME PROGRAMMING (XP)**

**Extreme Programming (XP)** is a software development approach that focuses on:

* **Team collaboration**
* Writing **tests first** (called *Test-Driven Development*)
* Continuous improvement

**🤝 Pair Programming in XP**

* Two developers work **together** at one computer.
  + One writes the code (**Driver**)
  + The other reviews and guides (**Observer**)

**🧪 Steps in XP Unit Testing:**

1. **Pick a Requirement ("Story")**
   * Example: “User should be able to log in with a username and password.”
2. **Write a Test Case First** (This is key!)
   * Write a small test that will **fail** because the code isn’t written yet.
   * Example (in JUnit):

java

CopyEdit

assertEquals(true, login("user", "pass"));

1. **Write the Code to Pass the Test**
   * Now you write just enough code to make that test pass.
2. **Execute All Tests**
   * Run all existing and new tests. Some may still fail.
3. **Rework the Code Until All Tests Pass**
   * Fix any issues, improve the code, and make sure everything works.
4. **Repeat Steps 2–5 Until the Story is Complete**
   * Keep writing tests and code for each part of the feature.

**🧠 Why do this?**

* Keeps the code clean, tested, and working at all times.
* Bugs are caught early.

**🧬 Mutation Testing**

* A **mutation** = small change made intentionally in the code to **see if the tests can catch it**.
* If the tests **still pass** after the mutation, it means your tests may be **too weak**.

**Example:** Original:

java

CopyEdit

if (a > b) { return true; }

Mutation:

java

CopyEdit

if (a >= b) { return true; }

→ If your tests **don’t fail**, you might be missing important test cases.

**🛠️ TOOLS for UNIT TESTING**

Here’s a breakdown of common tools used during unit testing:

| **Tool** | **What It Does** |
| --- | --- |
| **Code Auditor** | Checks code for issues, bad practices |
| **Documenters** | Generates documentation from code/comments |
| **Interactive Debuggers** | Helps run code line-by-line to find bugs |
| **Static Code Analyzer** | Checks code **without running** it (e.g. path analysis) |
| **Software Inspection Support** | Helps teams review code together |
| **Test Coverage Analyzer** | Shows how much of the code is tested |
| **Test Data Generator** | Creates input values for tests |
| **Test Harness** | Manages and runs unit tests |
| **Performance Monitors** | Measures how fast or efficient the code is |
| **Network Analyzers** | Analyzes data sent/received over networks |
| **Simulators/Emulators** | Imitates real systems (e.g., mobile device) for testing |
| **Version Control** | Tracks changes in the code (e.g., Git) |

**✅ Example: JUnit (for Java)**

**JUnit** is a popular **unit testing framework** for Java.

Simple example:

java

CopyEdit

@Test

public void testAddition() {

assertEquals(5, add(2, 3));

}

This test checks if the add() method returns 5 when passed 2 and 3.