NUNIT

### ****What is Unit Testing?****

**Unit testing** is a type of software testing where **individual units or components** of code (usually functions or methods) are tested **in isolation** to verify that they work as expected.

A **unit** is the smallest testable part of a program (e.g., a method).

It helps ensure that each piece of code behaves correctly before integrating it into the larger system.

Developers typically write unit tests using testing frameworks like **JUnit** (Java), **NUnit** (C#), **pytest** (Python), etc.

**Example:** Testing a Calculator.Add(a, b) method to ensure it returns the correct sum.

**Difference Between Unit Testing and Functional Testing**

| **Feature** | **Unit Testing** | **Functional Testing** |
| --- | --- | --- |
| **Scope** | Tests a single method or function | Tests a complete feature or functionality |
| **Focus** | Internal logic and code correctness | System behavior from the user’s perspective |
| **Dependencies** | Uses **mocked** objects to isolate the unit | Uses **real** systems (e.g., databases, APIs) |
| **Written By** | Developers | QA testers or automation engineers |
| **Speed** | Very fast | Slower (due to setup, dependencies, UI, etc.) |
| **Purpose** | Ensure **each part** of the system works correctly | Ensure the **whole system** behaves as expected |

Smallest unit to test mocking dependencies

The **smallest unit** in software testing is typically a **method or function** within a class. This is the core of **unit testing**.

**Unit testing** focuses on verifying the correctness of individual methods **in isolation** from the rest of the system.

### ****Various Types of Software Testing****

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

**What it is:** Testing individual units of code (e.g., methods or functions) in isolation.

**Purpose:** Validate that each small piece of the application performs as intended.

**Tools:** JUnit (Java), NUnit (C#), pytest (Python), Mocha (JavaScript)

**Who writes it:** Developers

### 2. ****Functional Testing****

**What it is:** Testing the complete functionality of an application against business requirements.

**Purpose:** Ensure the software behaves correctly from the **end-user’s perspective**.

**Includes:** UI testing, API testing, end-to-end testing

**Tools:** Selenium, Postman, Cypress, QTP

**Who writes it:** QA/Test Engineers

### 3. ****Automated Testing****

**What it is:** Any test (unit, functional, etc.) that is executed automatically by scripts/tools.

**Purpose:** Reduce manual effort and enable rapid, repeatable testing (especially in CI/CD pipelines).

**Examples:** Automated regression tests, smoke tests, integration tests

**Tools:** Selenium, JUnit, TestNG, Jenkins, Playwright

### 4. ****Performance Testing****

**What it is:** Testing how the system performs under stress, load, and varying conditions.

**Purpose:** Ensure stability, scalability, and responsiveness under real-world usage.

**Types:** Load testing, stress testing, endurance testing

**Tools:** JMeter, LoadRunner, Gatling, k6

Summary Table:

| **Type** | **Focus** | **Tool Examples** | **Goal** |
| --- | --- | --- | --- |
| Unit Testing | Individual methods/functions | NUnit, JUnit, pytest | Validate code logic |
| Functional Testing | Entire application features | Selenium, Postman | Check business rules |
| Automated Testing | Any test automated via tools | Jenkins, Cypress, Playwright | Faster, repeatable testing |
| Performance Testing | Speed and reliability | JMeter, k6, LoadRunner | Measure response time, load |

### ****Benefits of Automated Testing****

Automated testing refers to running tests using scripts and tools without manual intervention.

#### ****Key Benefits:****

**Speed**

Tests run much faster than manual testing.

Saves time, especially in large projects or CI/CD pipelines.

**Repeatability**

Tests can be run repeatedly across different environments.

**Early Bug Detection**

Automated tests help catch bugs early during development.

**Regression Coverage**

Ensures that new changes don’t break existing features.

**Cost-Effective in the Long Run**

Reduces the cost of testing over time as tests are reused.

**Supports Agile & DevOps**

Essential for Continuous Integration (CI) and Continuous Delivery (CD).

### ****Loosely Coupled & Testable Design****

A **loosely coupled** design means components or classes do **not depend directly** on each other. Instead, they communicate via **interfaces or abstractions**, which makes the code **more flexible and testable**.

// Define an interface

public interface IOrderRepository

{

void Save(Order order);

}

// Service depends on abstraction, not concrete class

public class OrderService

{

private readonly IOrderRepository \_repository;

public OrderService(IOrderRepository repository)

{

\_repository = repository;

}

public void SaveOrder(Order order)

{

\_repository.Save(order); // Can be mocked in tests

}

}

[Test]

public void SaveOrder\_CallsRepositorySave()

{

var mockRepo = new Mock<IOrderRepository>();

var service = new OrderService(mockRepo.Object);

var order = new Order();

service.SaveOrder(order);

mockRepo.Verify(r => r.Save(order), Times.Once);

}

### Understanding [SetUp], [TearDown], and [Ignore] Attributes in Unit Testing (e.g., ****NUnit**** for C#)

[SetUp] – Runs **Before Each Test**

Initializes common objects or state.

Avoids repeating setup code in every test.

[TestFixture]

public class CalculatorTests

{

private Calculator \_calc;

[SetUp]

public void Init()

{

\_calc = new Calculator(); // Re-created before each test

}

[Test]

public void Add\_TwoNumbers\_ReturnsCorrectSum()

{

Assert.AreEqual(5, \_calc.Add(2, 3));

}

}

[TearDown] – Runs **After Each Test**

Cleans up resources (e.g., closing DB connections, releasing memory).

Ensures a **clean state** after each test run.

[TearDown]

public void CleanUp()

{

calc = null; // Optional cleanup

}

[Ignore] – **Skips** a Test

Used when a test is **incomplete, unstable, or temporarily irrelevant**.

Prevents test failures during development.

[Test]

[Ignore("Feature not implemented yet")]

public void Multiply\_TwoNumbers\_ShouldReturnProduct()

{

Assert.Fail("Not implemented");

}

Summary Table:

| **Attribute** | **Purpose** | **When to Use** |
| --- | --- | --- |
| [SetUp] | Prepare data/resources before tests | Initialize test dependencies |
| [TearDown] | Clean up after tests | Reset or dispose resources |
| [Ignore] | Temporarily skip a test | When test is broken, incomplete, or blocked |

Explain the benefit of writing parameterised test cases.

**Parameterized test cases** allow you to run the **same test logic** multiple times with **different sets of input values**—making your tests **cleaner, more efficient, and easier to maintain**.

**TextFixture and Test**

using NUnit.Framework;

using CalcLibrary; // Your Calculator class namespace

namespace CalcLibrary.Tests

{

[TestFixture]

public class CalculatorTests

{

private Calculator \_calculator;

// This runs before each test

[SetUp]

public void SetUp()

{

\_calculator = new Calculator();

}

// This runs after each test

[TearDown]

public void TearDown()

{

\_calculator = null;

}

// Parameterized test for addition

[TestCase(2, 3, 5)]

[TestCase(-1, 4, 3)]

[TestCase(0, 0, 0)]

[TestCase(-2, -3, -5)]

public void Add\_WhenCalled\_ReturnsCorrectSum(int a, int b, int expected)

{

var result = \_calculator.Add(a, b);

Assert.That(result, Is.EqualTo(expected));

}

}

}