## Unit Testing vs Functional Testing

****Unit Testing**** is a software testing method where individual components (units) of the software are tested in isolation from the rest of the system. The "smallest unit" typically refers to a single function, method, or class.

****Key characteristics of unit testing:****

* Tests individual components in isolation
* Uses mocking/stubbing for dependencies
* Fast execution
* Focused on code correctness at the lowest level
* Usually written by developers

****Functional Testing**** is a broader testing approach that verifies that the system functions according to requirements and specifications.

****Key differences:****

* Scope: Unit tests small pieces; functional tests complete features.
* Dependencies: Unit tests mock dependencies; functional tests use real ones.
* Speed: Unit tests are faster.
* Purpose: Unit tests verify code works; functional tests verify system behaves correctly.

## Types of Testing

****Unit Testing****: Testing individual components in isolation.

****Functional Testing****: Testing complete features/functionality.

****Automated Testing****: Tests that run without manual intervention.

****Performance Testing****: Testing system responsiveness/stability under load.

**Integration Testing:** Testing how components work together.

**System Testing:** Testing the complete system.

**Regression Testing:** Re-running tests to ensure new changes don't break existing functionality.

**Acceptance Testing:** Validating the system meets business requirements.

## Benefits of Automated Testing

* Faster feedback on code changes
* Early bug detection
* Reduced manual testing effort
* Enables continuous integration/deployment
* Improves code quality
* Provides documentation of expected behavior
* Makes refactoring safer
* Reduces regression bugs

## Loosely Coupled & Testable Design

****Loosely coupled design**** means components have minimal dependencies on each other, making them easier to test and maintain.

****Characteristics of testable design:****

* Dependencies are injected rather than hard-coded
* Classes don't create their own dependencies
* Follows Dependency Inversion Principle
* Uses interfaces/abstractions rather than concrete implementations
* Methods are small and focused

#### Example:

public class InvoicePrinter {

private IInvoiceService \_service;

public InvoicePrinter(IInvoiceService service) {

\_service = service;

}

}

### 5. ****Write Your First NUnit Test ([TestFixture], [Test])****

using NUnit.Framework;

[TestFixture]public class CalculatorTests

{

[Test]

public void Add\_2Plus3\_Returns5()

{

var calc = new Calculator();

int result = calc.Add(2, 3);

Assert.That(result, Is.EqualTo(5));

}

}

### 6. ****[SetUp], [TearDown], and [Ignore]****

[SetUp] — Runs **before each** test (initializes resources)

[TearDown] — Runs **after each** test (cleans up)

[Ignore] — Skips a test method

[SetUp]public void Init() { /\* run before each test \*/ }

[TearDown]public void Clean() { /\* run after each test \*/ }

[Test, Ignore("Under development")]public void TestToSkip() { }

### 7. ****Benefits of Parameterized Tests ([TestCase])****

#### Instead of writing multiple test methods:

[Test]public void Add\_2Plus3() => Assert.That(calc.Add(2,3), Is.EqualTo(5));

[Test]public void Add\_5Plus7() => Assert.That(calc.Add(5,7), Is.EqualTo(12));

#### Use [TestCase] to simplify:

[TestCase(2, 3, 5)]

[TestCase(5, 7, 12)]public void Add\_TestCases(int a, int b, int expected)

{

Assert.That(calc.Add(a, b), Is.EqualTo(expected));

}

**Benefits**:

* Cleaner code
* Easier to maintain
* More test coverage with fewer lines