## Unit Testing vs Functional Testing

- Unit Testing focuses on individual components or functions of the code in isolation. For example, testing just the Add() method in a calculator.
- Functional Testing checks the entire system's behavior to ensure it meets the specified requirements, like verifying the calculator UI produces correct results.

| Feature | Unit Testing | Functional Testing |
|---|---|---|
| Scope | Smallest part (methods) | Entire application flow |
| Tools | NUnit, xUnit | Selenium, Postman, etc. |
| Speed | Fast | Slower |
| Dependencies | Mocked | Real or integrated |

## Mocking Dependencies in Unit Testing

In Unit Testing, we test only one function or class in isolation. If that function depends on other services (like a database or web API), we use mocking to simulate those services.

Example: Use a fake service that returns dummy data instead of calling a real database.

## Types of Testing

1. Unit Testing: Testing small components (e.g., a method).
2. Functional Testing: Verifying business logic.
3. Automated Testing: Tests written and run by tools (e.g., NUnit, Selenium).
4. Performance Testing: Checks responsiveness under load (e.g., JMeter).

## Benefits of Automated Testing

- Faster test cycles
- Repeatable and consistent
- Easier debugging
- Enables continuous integration

## Loosely Coupled & Testable Design

- A loosely coupled design means components don't depend heavily on each other.
- For testability: avoid using hardcoded dependencies inside your classes. Instead, use interfaces or pass dependencies via constructors or setters.

**Example:**

```
public class Calculator {
   public int Add(int a, int b) => a + b;
}
```

This method is testable because it doesn't depend on any external class or data source.

## NUnit Attributes

| Attribute | Use |
|---|---|
| [TestFixture] | Declares a test class |
| [Test] | Marks a test method |
| [SetUp] | Code runs before each test |
| [TearDown] | Code runs after each test |
| [Ignore] | Skips a test (useful for debugging) |

## Parameterized Tests with [TestCase]

Instead of writing multiple tests for the same method with different inputs, we use [TestCase] to test many scenarios in one method.

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

[TestCase(0, 0, 0)]

[TestCase(-1, -1, -2)]

public void TestAdd(int a, int b, int expected) {

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

}
```

## Benefits:

- Reduces repetitive code
- Easier to manage input variations