Unit Testing Objectives and Implementation

# 1. Unit Testing vs Functional Testing

Unit testing focuses on testing the smallest piece of code in isolation, typically a single method or function.  
Functional testing evaluates the complete functionality of an application against its requirements.  
  
- Unit Testing: Tests individual units of code (e.g., methods), often with mocked dependencies.  
- Functional Testing: Tests complete features or user workflows using real or simulated environments.  
  
Example: Testing the `Addition()` method alone is unit testing, while testing the entire Calculator UI is functional testing.

# 2. Types of Testing

* Unit Testing: Tests individual components.
* Functional Testing: Validates business functionality.
* Automated Testing: Tests executed without manual steps.
* Performance Testing: Assesses application speed, scalability, and stability under load.

# 3. Benefits of Automated Testing

* Faster feedback loop during development
* Reduced manual effort and human error
* Easier regression testing
* Supports Continuous Integration (CI) and Continuous Deployment (CD)
* Improves software quality and maintainability

# 4. Loosely Coupled & Testable Design

A loosely coupled design allows components to work independently. This makes testing easier since dependencies can be mocked or substituted.  
  
Example: Instead of directly instantiating a class inside another, pass the dependency through a constructor or interface.  
  
Bad:  
var db = new Database();  
  
Good:  
public MyService(IDatabase db) { this.db = db; }

# 5. First Testing Program (Calculator Addition)

Example using NUnit:  
[TestFixture]  
public class CalculatorTests  
{  
 private SimpleCalculator calculator;  
  
 [SetUp]  
 public void Setup()  
 {  
 calculator = new SimpleCalculator();  
 }  
  
 [TearDown]  
 public void Teardown()  
 {  
 calculator = null;  
 }  
  
 [TestCase(2.0, 3.0, 5.0)]  
 public void Addition\_ShouldReturnExpected(double a, double b, double expected)  
 {  
 var result = calculator.Addition(a, b);  
 Assert.That(result, Is.EqualTo(expected).Within(0.0001));  
 }  
}

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

* [SetUp]: Runs before each test to prepare environment
* [TearDown]: Runs after each test to clean up
* [Ignore]: Skips the test temporarily

Example:  
[Test, Ignore("Not implemented yet")]  
public void UnusedTest() { }

# 7. Benefit of Parameterized Test Cases

* Eliminates repetitive test methods
* Tests multiple inputs in one method
* Improves readability and reduces maintenance

Example:  
[TestCase(1, 2, 3)]  
[TestCase(-1, -2, -3)]  
public void Add\_ShouldReturnSum(int a, int b, int expected)  
{  
 Assert.That(calculator.Add(a, b), Is.EqualTo(expected));  
}

## Summary of Attributes Used

| **Attribute** | **Purpose** |
| --- | --- |
| [TestFixture] | Declares the class is a test class |
| [SetUp] | Runs before each test to initialize objects |
| [TearDown] | Runs after each test to clean up |
| [TestCase] | Supplies input/output to a test method |
| Assert.That | Compares actual and expected result |