## 1. Unit Testing vs Functional Testing

**Unit Testing:**

**Tests:** Smallest units (methods/function

**Dependencies:** Mocked/stubbed

**Who:** Developers

**When:** During development

**Purpose:** Verify code logic works as designed

**Functional Testing:**

**Tests:** Entire features/user workflows

**Dependencies:** Real implementations

**Who:** QA/Test Engineers

**When:** After integration

**Purpose:** Verify system meets requirements

**Key Difference:**  
Unit tests isolate smallest units with mocked dependencies; functional tests check end-to-end behavior with real components.

## 2. Types of Testing

**Unit Testing:** Test individual components in isolation (e.g., a method in a class)

**Functional Testing:** Verify features against requirements (e.g., login workflow)

**Automated Testing:** Scripted tests that run without manual intervention (e.g., Selenium)

**Performance Testing:** Measure system under load (e.g., JMeter)

## 3. Benefits of Automated Testing

**Faster feedback:** Run tests in seconds

**Higher accuracy:** Eliminate human error

**Cost reduction:** Save time on regression testing

**Continuous delivery:** Enable CI/CD pipelines

**Documentation:** Tests serve as living specs

## 4. Loosely Coupled & Testable Design

**Principles:**

**Dependency Injection:**

csharp

**public** **class** PaymentProcessor { **private** **readonly** IPaymentGateway \_gateway; **public** PaymentProcessor(IPaymentGateway gateway) { \_gateway = gateway; *// Dependency injection* }}

**Avoid class data dependencies:**

csharp

**public** List<User> GetUsers(IUserRepository repo) { **return** repo.GetAll(); *// Decoupled from data source*}

**Benefits:**

Easier mocking for unit tests

Components testable in isolation

More flexible to change

## 5. First Test Program (Calculator)

**Production Code:**

csharp

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

**Test Code (NUnit):**

csharp

[TestFixture]**public** **class** CalculatorTests { **private** Calculator \_calculator; [SetUp] **public** **void** Setup() { \_calculator = **new** Calculator(); } [Test] **public** **void** Add\_TwoNumbers\_ReturnsSum() { *// Arrange* **int** a = 5, b = 3; *// Act* **var** result = \_calculator.Add(a, b); *// Assert* Assert.AreEqual(8, result); }}

## 6. Test Attributes

|  |  |  |
| --- | --- | --- |
| **Attribute** | **Purpose** | **Example** |
| [SetUp] | Prepares test environment | Create test objects |
| [TearDown] | Cleans up after test | Dispose database connection |
| [Ignore] | Temporarily skip a test | [Ignore("WIP")] |
| [TestCase] | Parameterized testing | See below |

## 7. Parameterized Testing

**Traditional Test:**

csharp

[Test]**public** **void** Add\_PositiveNumbers\_ReturnsSum() { **var** result = \_calculator.Add(2, 3); Assert.AreEqual(5, result);}

**Parameterized Version:**

csharp

[TestCase(2, 3, 5)][TestCase(0, 0, 0)][TestCase(-1, 5, 4)][TestCase(100, 200, 300)]**public** **void** Add\_VariousInputs\_ReturnsCorrectSums(**int** a, **int** b, **int** expected) { **var** result = \_calculator.Add(a, b); Assert.AreEqual(expected, result);}

**Benefits:**

Less code duplication

Clearer test data

One failure doesn’t block others

Easy to add edge cases

## 8. Key Takeaways

**Unit tests** verify small units in isolation using mocks.

**Testable design** needs loose coupling and dependency injection.

**Parameterized tests** reduce duplication and improve coverage.

**Automated testing** enables continuous delivery and reduces costs.