**Q1. EXPLAIN THE MEANING OF UNIT TESTING AND ITS DIFFERENCE ON COMPARISON WITH FUNCTIONAL TESTING.**

**Solution:**

**Unit Testing is a type of software testing where individual components or functions (called *units*) of a program are tested in isolation to ensure that they work as expected.**

* **It focuses on a single function, method, or class.**
* **Usually written and run by developers.**

**Functional Testing is a type of testing that validates the entire application or feature against the business requirements.**

* **It focuses on what the system does rather than how it does it.**

**Example:  
Testing a login feature by:**

* **Entering a username and password.**
* **Clicking "Login".**
* **Verifying that the user is redirected to the dashboard.**
* **Difference Between Unit Testing and Functional Testing**

| **Feature** | **Unit Testing** | **Functional Testing** |
| --- | --- | --- |
| **Scope** | Individual components (methods/functions) | Entire application features or modules |
| **Focus** | Internal logic | User-facing behavior |
| **Performed by** | Developers | Testers (or QA engineers) |
| **Level** | Low-level (code level) | High-level (system or acceptance level) |
| **Tools** | NUnit, JUnit, PyTest, xUnit | Selenium, QTP, Postman, etc. |
| **Speed** | Very fast | Slower (involves full flow) |
| **Dependency** | No or few external dependencies | Works with full system and all dependencies |
| **Automation** | Common and easy | Possible but more complex |

**Q2. LIST VARIOUS TYPES OF TESTING**

· **Unit Testing** – Tests individual methods or components.

· **Functional Testing** – Ensures the software meets functional requirements.

· **Automated Testing** – Uses scripts/tools to automatically run tests (can be unit or functional).

**Q3. UNDERSTAND THE BENEFIT OF AUTOMATED TESTING**

### **Benefits of Automated Testing**

* **Repeatability**: Run tests frequently and consistently.
* **Speed**: Faster than manual testing.
* **Reliability**: Reduces human error.

**Q4. EXPLAIN WHAT IS LOOSELY COUPLED & TESTABLE DESIGN**

**Loosely Coupled Design**

**A loosely coupled design means that different parts of a program (like classes or modules) are minimally dependent on each other.**

* **Components interact through interfaces or abstractions, not direct references.**
* **If one part changes, it doesn’t break or affect other parts.**
* **Makes the system more flexible, maintainable, and easier to modify.**

**Example:  
Instead of a class directly creating a DatabaseConnection, it should depend on an IDatabase interface. This way, you can swap the database type without changing the rest of the system.**

**✅ Testable Design**

**A testable design is a design that allows individual components to be easily tested (especially through automated unit tests).**

* **The code is written to allow injection of dependencies (using constructors or setters).**
* **Mocks and stubs can be used to replace real dependencies during testing.**
* **Follows the Single Responsibility Principle, keeping logic small and focused.**

**Example:  
If a class needs to send emails, don’t hardcode the email service. Inject an IEmailService interface so that during testing, you can use a fake or mock service.**

**🔁 Why They Go Together**

* **Loosely coupled systems naturally become more testable.**
* **Both promote clean architecture, faster testing, and better code quality.**

**CODE:**

using CalcLibrary;

namespace StudentGrades.nUnitTests

{

[TestFixture]

public class GradeCalculatorTests

{

private SimpleCalculator calculator;

[SetUp]

public void Setup()

{

calculator = new SimpleCalculator();

}

public void TearDown()

{

calculator = null;

}

[TestCase(50, 30,80)]

[TestCase(10, 20, 30)]

[TestCase(100, 14, 114)]

[TestCase(20, 30, 50)]

public void Addition\_EqualTest(double a, double b, double expected)

{

var actual = calculator.Addition(a, b);

Assert.AreEqual(expected, actual);

}

[TestCase(22,22,11)]

[TestCase(70, 10, 100)]

[TestCase(120, 1, 122)]

[TestCase(11, 10, 90)]

public void Addition\_NotEqualTest(double a, double b, double Notexpected)

{

var actual = calculator.Addition(a, b);

Assert.AreNotEqual(Notexpected, actual);

}

}

}

