**NUnit Theory**

**What is Unit Testing?**

Unit testing is a software development practice where individual pieces of code, usually functions or methods, are tested in isolation to ensure they work correctly. It focuses on validating one small "unit" of behavior at a time without relying on external systems like databases or APIs. The goal is to catch bugs early in development, which makes debugging easier and improves code quality. Unit tests are typically written by developers and are fast to execute, which makes them ideal for running frequently during development.

**Difference Between Unit Testing and Functional Testing**

While unit testing checks the behavior of individual methods, functional testing focuses on verifying that a complete feature or system behaves as expected. For example, unit tests might test the Add() method of a calculator, while functional tests might check if the entire calculator UI performs correctly. Functional testing often involves multiple units working together and may depend on real or simulated external components. In contrast, unit testing avoids such dependencies, often using mocks or stubs to isolate the test.

**Types of Testing**

There are several types of testing used in software development:

Unit Testing: Tests individual methods or classes.

Functional Testing: Verifies the functionality of an application from the user's perspective.

Automated Testing: Uses scripts and tools to run tests automatically, including unit and functional tests.

Performance Testing: Measures how well a system performs under load, checking speed, responsiveness, and stability.

Each type of testing plays a different role in the quality assurance process.

**Benefits of Automated Testing**

Automated testing brings many benefits to modern development workflows. It allows teams to run tests frequently and consistently without manual effort. This leads to quicker feedback, reduces human error, and makes continuous integration and delivery (CI/CD) pipelines more reliable. Automated tests also improve regression testing, ensuring that new code changes don’t break existing functionality. In the long run, it saves time, reduces costs, and boosts developer confidence.

**Loosely Coupled and Testable Design**

A loosely coupled design means your classes and components are not tightly bound to each other. This is crucial for writing testable code. For instance, instead of hardcoding a dependency inside a class, we inject it using an interface. This way, in tests, we can replace real implementations with mock versions to control behavior and isolate the logic being tested. Writing code that follows principles like Dependency Injection and the SOLID design principles results in a system that’s easier to test, maintain, and extend.

**Understanding [TestFixture], [Test], [SetUp], [TearDown], and [Ignore]**

In NUnit, [TestFixture] marks a class as containing unit tests. Inside it, [SetUp] is used to prepare the environment before each test runs — for example, initializing an object. [TearDown] is used to clean up afterward. [Test] is used to define a test case method. Sometimes you may want to skip a test temporarily — for that, you use [Ignore], optionally with a reason like "under development" or "blocked by bug".

**Benefits of Parameterized Test Cases ([TestCase])**

Rather than writing multiple test methods for similar logic, parameterized test cases let you test multiple inputs and expected outputs using a single test method. The [TestCase] attribute in NUnit makes this possible. For example, you can test a calculator’s Add method with inputs (2, 3) expecting 5, (0, 0) expecting 0, and (-1, 1) expecting 0 — all in one test. This improves readability, reduces code duplication, and ensures thorough testing of edge cases.