Question Answering Assignment

1. Search about these topics (Parallel Programming and Concurrency - Unit Testing and Test-Driven Development (TDD) - Asynchronous Programming with async and await)

Parallel Programming and Concurrency

Parallel programming and concurrency are related but distinct concepts in computer science.

- Concurrency refers to managing multiple tasks at the same time, but not necessarily executing them simultaneously. It creates an illusion of parallelism by interleaving task execution on a single processing unit through context switching. It is commonly used to improve responsiveness, such as handling multiple user requests. Tasks start, run, and complete in overlapping time periods without running simultaneously on multiple processors. Debugging concurrent programs can be difficult due to non-deterministic control flow.
- Parallelism means executing multiple tasks simultaneously on multiple processing units (e.g., multiple CPU cores). It divides a task into subtasks that run in parallel to improve throughput and computational speed. It is deterministic and often used for high-performance applications.
- Parallelism requires multiple processors, while concurrency can be achieved on a single processor by rapid switching between tasks. Parallelism is a subset of concurrency.

Example:

```
using System;
using System.Threading.Tasks;

class Program
{
    static void Main()
    {
        // Using Parallel.Invoke to run multiple tasks in
parallel
        Parallel.Invoke(
            () => DoWork("Task 1"), // Task 1 does some work
concurrently
            () => DoWork("Task 2"), // Task 2 runs
simultaneously
            () => DoWork("Task 3")  // Task 3 runs
simultaneously
        );
    }

    static void DoWork(string taskName)
    {
        Console.WriteLine($"{taskName} starting..."); //
Indicate task start
        System.Threading.Thread.Sleep(1000);          //
Simulate a 1-second workload
        Console.WriteLine($"{taskName} done.");        //
Indicate task completion
    }
}
```

- Parallel.Invoke runs the lambda expressions concurrently, allowing the three tasks to start and run simultaneously.

- Each DoWork call represents an independent unit of work simulated by a 1-second delay.
- The output order for start and end messages is not guaranteed due to parallel execution.

Unit Testing and Test-Driven Development (TDD)

- Unit Testing involves writing test cases that validate the functionality of the smallest parts of a program (units or components). The goal is to ensure each unit works as expected.
- Test-Driven Development (TDD) is a software development practice where automated unit tests are written before the actual code. The development follows a short repetitive cycle called Red-Green-Refactor:
    a. Write a test case that initially fails (Red).
    b. Write just enough code to make the test pass (Green).
    c. Refactor the code to improve structure without changing behavior.

This cycle is repeated for each new piece of functionality, ensuring code quality and reliability from the start. Writing tests first drives the design and helps keep code decoupled and maintainable.

Example:

```
// Calculator.cs - Production Code
public class Calculator
{
    // A simple method to add two integers
    public int Add(int a, int b)
```

```
    {
        return a + b;
    }
}

// CalculatorTests.cs - Unit Test (NUnit Framework)
using NUnit.Framework;

[TestFixture] // Marks this class as a test suite
public class CalculatorTests
{
    [Test] // Marks this method as a test case
    public void Add_SimpleValues_CalculatesCorrectly()
    {
        var calculator = new Calculator();       // Arrange:
create Calculator instance
        int result = calculator.Add(2, 3);       // Act: call
Add method
        Assert.AreEqual(5, result);               // Assert:
check if result is 5
    }
}
```

- The test *A*dd_Simple*V*alues_CalculatesCorrectly is written first, initially failing.
- Then the *A*dd method is implemented to make the test pass – this follows the TDD cycle.
- NUnit attributes [TestFixture] and [Test] identify test classes and methods respectively.

Asynchronous Programming with async and await

- Asynchronous programming allows a program to perform tasks without blocking the execution flow waiting for those tasks to complete, improving responsiveness and efficiency.
- The keywords async and await facilitate asynchronous programming by allowing methods to run asynchronously. An async method runs synchronously until it hits the first await keyword, at which point it yields control to the caller until the awaited task completes.
- Using await pauses the execution of the async method, allowing other work to happen meanwhile, and resumes it once the awaited operation finishes. This approach helps write asynchronous code in a sequential style, making it easier to understand and maintain.

Example:

```
using System;
using System.Threading.Tasks;

class Program
{
    static async Task Main()
    {
        await SayHelloAsync(); // Await the asynchronous method
without blocking Main thread
    }

    static async Task SayHelloAsync()
    {
        Console.WriteLine("Hello");          // Print
immediately
        await Task.Delay(1000);               // Non-blocking
wait for 1 second
        Console.WriteLine("World");          // Prints after the
```

```
delay
    }
}
```

- async keyword marks the method as asynchronous, allowing await inside it.
- await Task.Delay(1000) asynchronously pauses the method for 1 second without blocking the main thread.
- This allows better responsiveness and concurrency in applications.