**A1.** We are maintaining Single Responsibility Principle here. There are 2 main responsibilities in the class Work. Reading data and displaying data. The DisplayData method is responsible for the UI part and the methods ReadFile and ReadDatabase are responsible for reading different formats of data. Each method has only one responsibility. If we need to add another method, for example, ReadJSONData which is specifically for data in JSON, then we do not need to disturb/ change these existing methods. The new method can be added and implemented independently and has different responsibilities than others.

**A2.** Important points to consider for LSP.

i) The substituting class (derived class) must not break any functionality of the original class (base class).

ii) The derived class object must work in the same way as the base class.

**A3.** Examples of interface segregation

Let us consider the following example. Here we have an interface IStartLaptop with various methods.

interface IStartLaptop

{

void loadDesktop();

void connectInternet();

void startChrome();

void StartGame();

}

Here, for any class implementing this interface, all of the methods need to be implemented. But if the user wants to use the laptop offline, he doesn’t need the connectInternet and startChrome methods. Thus, these 2 methods can be a part of a different interface. Also, the user might use the laptop for other purposes than playing games. So StartGame method can also be segregated into a different interface.

interface ILoadInternet

{

void connectInternet();

void startChrome();

}

interface IStartLaptop

{

void loadDesktop();

}

interface IPlayGame

{

void startGame();

}

class LoadOfflineGame : IStartLaptop, IPlayGame

{

void loadDesktop();

void startGame();

}

class LoadOnlineGame : IStartLaptop, IPlayGame, ILoadInternet

{

void loadDesktop();

void connectInternet();

void startChrome();

void startGame();

}

**A4.**

Example for Tight coupling

class Chrome

{

public void OpenChrome ()

{

Console.WriteLine("Opened chrome browser");

}

}

class Game

{

public void StartGame(Chrome page)

{

page.OpenChrome();

Console.WriteLine("Started the game");

}

}

class Program

{

static void Main(string[] args)

{

Chrome page = new Chrome();

Game game = new Game();

game.StartGame(page);

Console.ReadLine();

}

}

In the above example, we are playing an online game. The game needs to be opened on the browser. The browser used is Chrome only. Here ,if we want to add another browser like Firefox, we need to make changes in Game class. Hence, the Game class is tightly coupled with Chrome class.

Making it Loose coupling

class Chrome: IBrowserInterface

{

public void OpenBrowser()

{

Console.WriteLine("Opened chrome browser from interface");

}

}

class Firefox : IBrowserInterface

{

public void OpenBrowser()

{

Console.WriteLine("Opened firefox browser from interface");

}

}

class Game

{

public void StartGame(IBrowserInterface page)

{

page.OpenBrowser();

Console.WriteLine("Started the game");

}

}

class Program

{

static void Main(string[] args)

{

Chrome page = new Chrome();

Game game = new Game();

game.StartGame(page);

Console.ReadLine();

}

}

Here, the Main() code remains the same. And the Game class is not dependent on any class. The StartGame function takes a parameter of IBrowserInterface type. There is no dependency of the Chrome class or Firefox class in the final implementation. Hence, the classes are loosely couples now.

**A5.** The same above code example with slight modification can be used to demonstrate Open Closed Principle as well. To hide implementation behind abstraction, we add another layer. The OCPLayerClass.

class OCPLayer

{

public static IBrowserInterface GetBrowser()

{

return new Chrome();

}

}

static void Main(string[] args)

{

IBrowserInterface browser = OCPLayer.GetBrowser();

//Firefox page = new Firefox();

Game game = new Game();

game.StartGame(browser);

Console.ReadLine();

}

Since the Chrome and Firefox classes are implementing IBrowserInterface interface, we cannot remove the OpenBrowser() method. This shows that the classes are closed for modification. But they can be used by other implementations or inherited by other classes. This makes them Open for extensibility.