**SOLID Design Principles**

1. **S - Single Responsibility Principle**

A class or method should have only one [reason to change + single responsibility]

1. **O - Open Closed Principles**

Software entities such as classes, modules, methods, etc. should be an open **extension** but closed for **modification**.

1. **L - Liskov Substitution Principles**
2. **I - Interface Segregation Principles**

Clients should not be forced to depend upon the interfaces that they do not use

1. **D - Dependency Inversion Principles**

**Introduction:**

SOLID is an acronym for five important design principles when doing OOP (Object Oriented Programming).

It takes some time to understand, but if you write your code following SOLID it will improve your code quality

**Why SOLID – summary: If we do not follow SOLID principles**

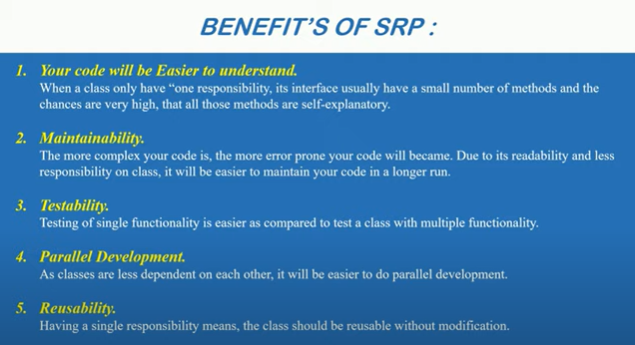
1. Our code might become tightly coupled.
2. Tight coupling creates problems while implementing any new requirement, features, or bug fixes.
3. End up with duplicate & non-testable code.
4. End up creating a new bug while implanting new features or fixing an existing bug

**SOLID helps us:**

1. Reduce the complexity of code
2. It makes your code extensible, logical, and easier to read
3. It makes your code loosely coupled

**S - Single Responsibility Principle**

A class or method should have only one [reason to change + single responsibility]



**How to implement SRP**

Project Requirement:

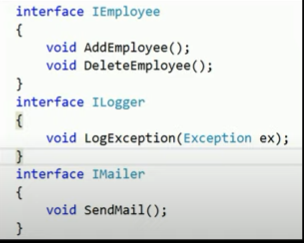
1. The application should be able to Add employee
2. The application should be able to Delete employee
3. With every adds and deletes one email should be send
4. In case of exception, a text log file will be generated

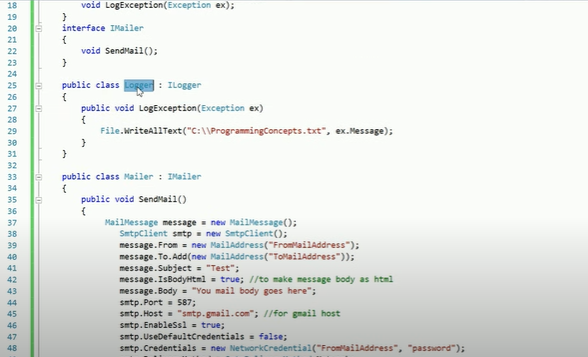
Solution 1: Without SRP Principles

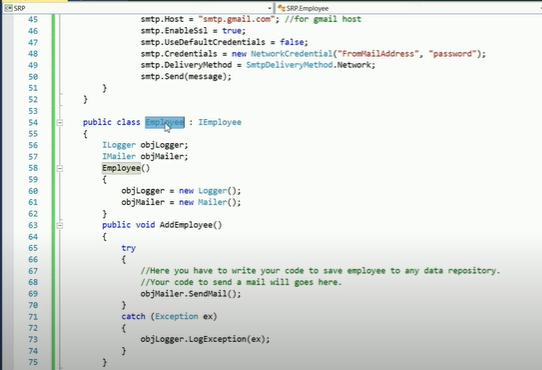
We can send an email and create a log of both Add & Delete methods separately. But if there are arise any issues we have to solve both methods which is not a good practice.

Solution 2: Using SRP Principles

1. So we create Ilogger and IMailer interfaces and implement these interfaces with two classes.
2. Mailer class takes responsibility for sending email
3. Logger class takes responsibility for creating the log
4. Instance create from employee constructor and then calling email-sender & logger method using those instance





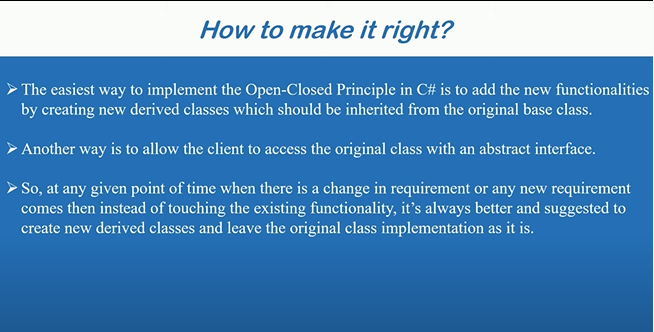


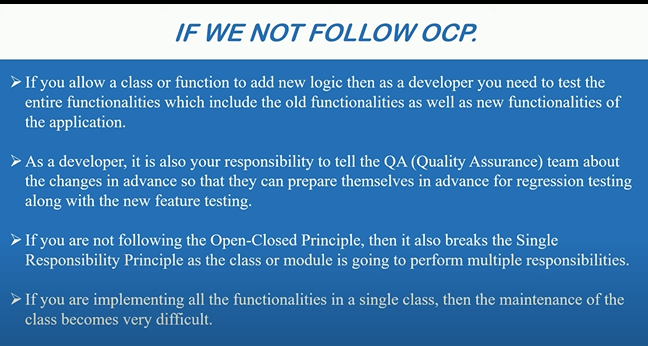
**O - Open Closed Principles**

Software entities such as classes, modules, methods should be open for **extension** but closed for **modification**.

**Open for an extension means** we need to design the software modules/class in such a way that the new functionalities should be added easily when new requirements come.

**Closed for modification means** we should not modify the class/modules until we find the bug





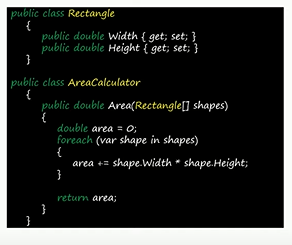
Project Requirement:

1. Clients want to build software by which they will be capable of calculating the sum of the area of a rectangle.

Summarize requirements:

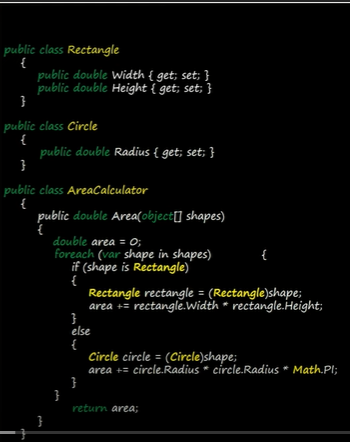
1. Need to calculate the sum of the Area of the rectangle
2. This means if we send two rectangles it should return the sum of both the rectangles.

Solution 1: To find out the Area of the rectangle



But after a few days client wants to calculate the area of circles

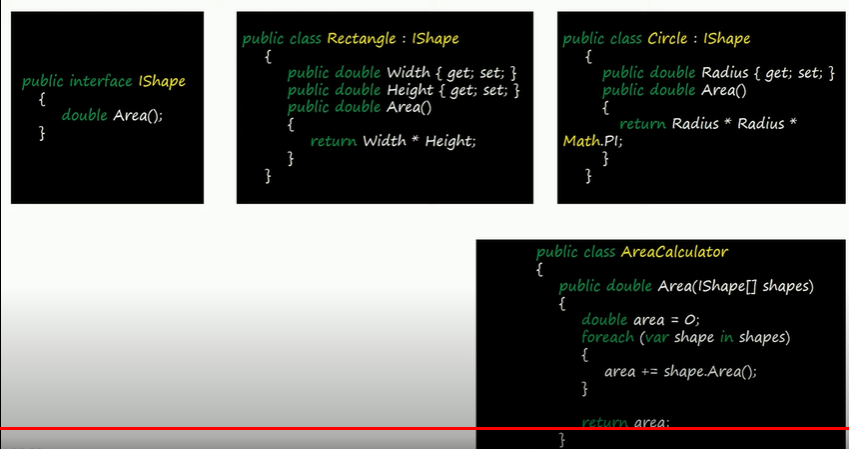
Solution 2: To find out Area of the rectangle and circle. Refactor the previous code

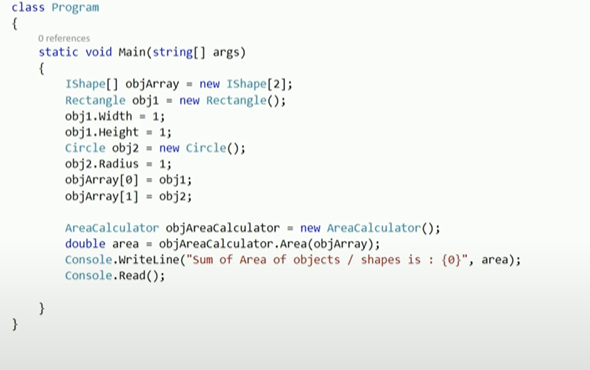


2nd requirement is also solved, but the Team lead says this code is not following the OCP. If clients request to add another option for calculating area, you have to again modify your code

So let’s implements the OCP

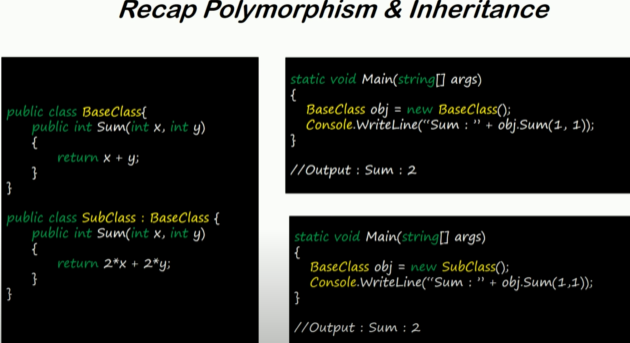






**L - Liskov Substitution Principles**

**Inheritance Example:**



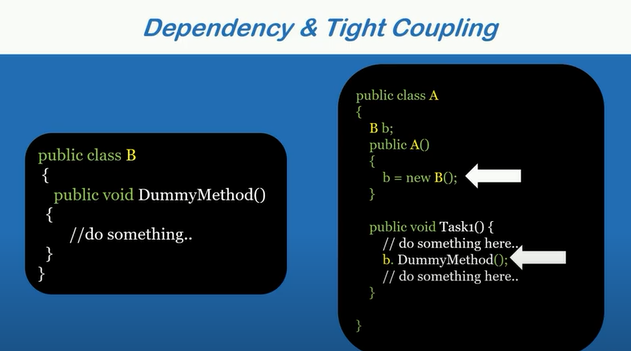
**Comparison of**

* **Inversion of control (IOC)**
* **Dependency Inversion Principle (DIP) and**
* **Dependency Injection (DI)/ IOC Container**

**Question:**

1. **How Dependency Injection help in Unit Testing**
2. **How web.config can be your best example when explaining IOC**
3. **Why should we use DI when we already have DIP**

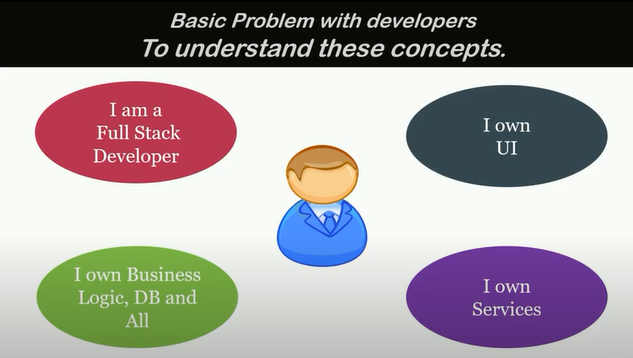
**Dependency & Tight Coupling:**



1. Here A-Class is dependent on B-Class for **Dummy Method**
2. B-Class is declared tightly inside the constructor

Basic Problem with Developers to understand these problems

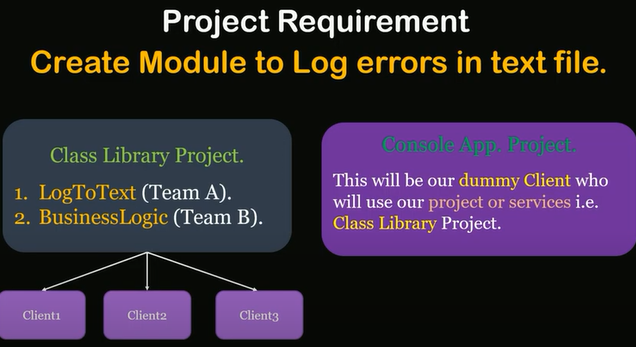
1. I am a Full Stack developer
2. I own UI, Services, Business Logic, DB and all

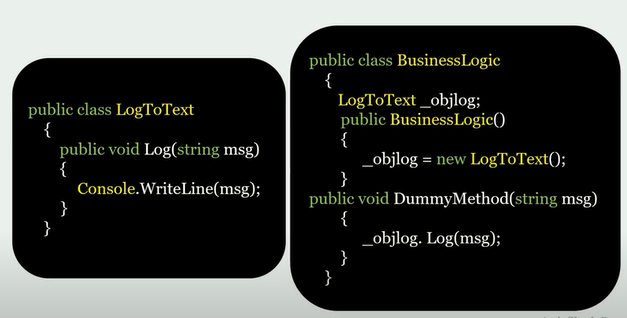


Solution

Assumption: You do not own even a single class in your project

Project Requirement: Create a module to Log errors in a text file





Note:

1. Business Logic is dependent on **LogToText** Class
2. Business logic is tightly coupled with **LogToText** Class

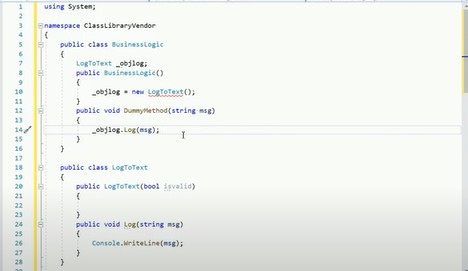
Question: Is there any problem with this implementation?

Answer: In general Not a big problem for a small project But Problems Occurs in Big Projects

Problems:

1. In case of change on dependent class **LogToText**, we need to make changes in all the places where we are referencing this class. Here we add a parameterized constructor.

**In a small project, we can solve the error easily. But big projects we do not have overall control to modify the errors.**

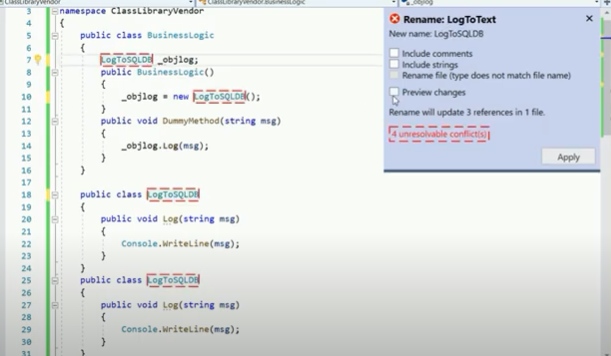


1. In case of future updates **LogToDatabase** we need to make modifications in all our dependent classes.

Note: we can rename the name using visual studio. If you think this way, you are still on Full Stack mode.

You think you do not have control over **others** Classes or Project.

This Dll also can use other applications which is not access from your company.



So let’s see how we can overcome those problems

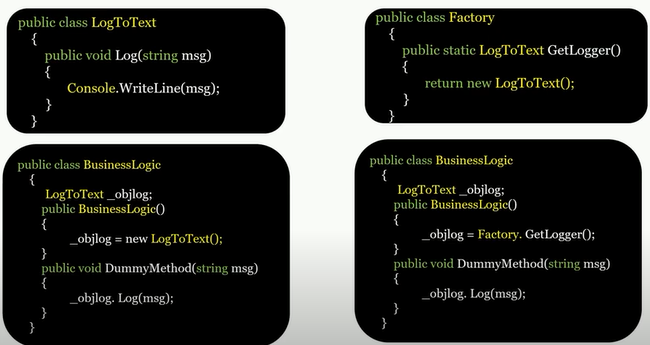
**Inversion of control (IOC):**

**IOC is a design principle used to convert different types of controls in object-oriented design to achieve loose coupling**

**The main objective of IOC is to remove dependencies between the objects of an application which makes the application more decoupled and maintainable**

**IOC is a principle, which means it is a general guideline and end-user can choose the way how they want to implement it.**

**Example 1:**



**The above examples solve problem No 1: (**In case of change on dependent class **LogToText**, we need to make changes in all the places)

**We cannot solve all the problems. It is just an example of how IOC works (invert the control).**

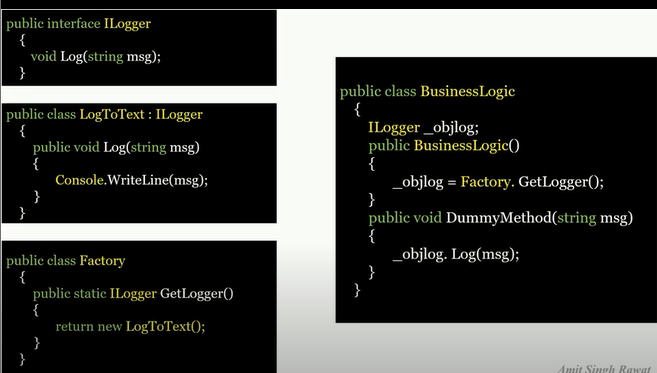
**Example 2: connection string**

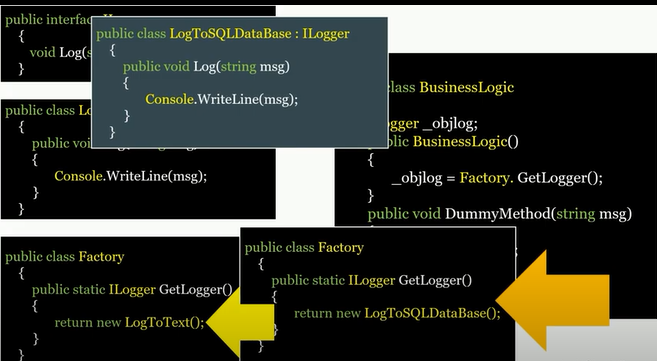
With IOC, the general idea is to give responsibility to someone who can handle it better than the current implementation. When we create a class library, the main purpose is either reusability or separation of concern. If I hardcode connection string in my class library (.dll) project, it will be tightly coupled with my DLL project. Which we certainly don't want. We want whosoever using my DLL can set their own connection string (or any other properties) as per their needs. So, If you are using DLL within your web project you can consider yourself one of the clients of that DLL. And you can define connection string in your web. config file, instead of the DLL owner hardcode connection string in their project

**Dependency Inversion Principle (DIP)**

DIP states that high-level modules/classes should not depend on low-level modules/classes. Both should depend upon abstraction.

Also, abstraction should not depend upon details. Details should depend upon abstraction



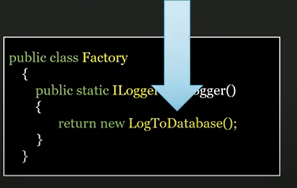


**The above examples solve problem No 2 (**In case of future updates **LogToDatabase)**

**All the problems solve. But there is also some problems**

**Problem 1: Control to select logger class is still with our Business Logic DLL.**

**So our client application do not have control to select which option they will choose. Just like connection string.**



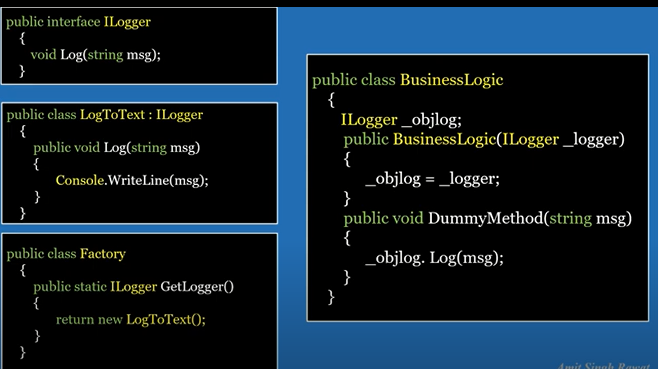
**Dependency Injection:**

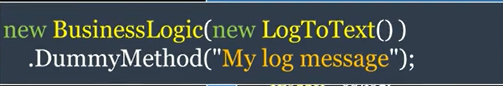
DI is a design pattern used to implement IOC

It allows the creation of dependent objects outside of a class and provides those to a class in different ways.

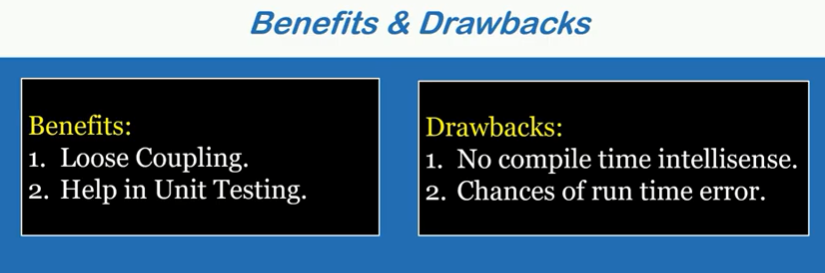
Using DI, we move the creation and binding of the dependent objects outside of the class that depends on them

Types of DI: Constructor, Property, Method Injection





**Call From Client Application**



**Help in Unit testing But How?**

