**Assignments on SOLID Principles**

**1. Single Responsibility Principle (SRP) – *Library Book Management***

**Problem Statement:**  
The city library wants to build a simple system to keep track of books. Initially, the library’s IT team created a Book class that:

* Stores details about the book (title, author, ISBN).
* Saves the book’s information to a database.
* Prints a report about the book for librarians.

This design works but violates good software design practices. The Book class has too many responsibilities — storing data, saving to the database, and printing.

**Task:**  
Refactor the system so that:

1. **One class** is responsible for holding the book data.
2. **One class** is responsible for saving books to a database.
3. **One class** is responsible for printing the book’s details.

**Goal:**  
Make sure each class has only **one reason to change** in the future. For example, a change in the printing format should not require modifying the class that saves data.

**2. Open/Closed Principle (OCP) – *Shape Area Calculation***

**Problem Statement:**  
A graphics application calculates the area of different shapes — currently circles and rectangles. The code uses if statements to check which shape is being processed, then calculates its area.

The problem is that **every time a new shape** (like a triangle or hexagon) is added, you have to **edit the existing area calculation code**, risking bugs and breaking existing functionality.

**Task:**

1. Create a IShape interface that declares a method to calculate the area.
2. Implement this interface for each shape (Circle, Rectangle, Triangle, etc.).
3. Update the area calculator to work with the IShape interface so it can handle any new shape without being modified.

**Goal:**  
Make the area calculator **open for extension** (you can add new shapes) but **closed for modification** (you never change the existing calculation code).

**3. Liskov Substitution Principle (LSP) – *Bird Example***

**Problem Statement:**  
A zoo management software has a Bird class with a Fly() method. All birds inherit from it, including sparrows, parrots, and ostriches.

However, ostriches cannot fly. When the Fly() method is called on an ostrich, it throws an exception. This leads to runtime errors and violates the principle that **objects of a superclass should be replaceable with objects of its subclasses without breaking the program**.

**Task:**

1. Redesign the class hierarchy so that flying birds and non-flying birds are handled separately.
2. Ensure that code expecting a flying bird will never accidentally get a non-flying bird.

**Goal:**  
Avoid forcing subclasses to implement methods they cannot logically perform, and maintain substitutability in the system.

**4. Interface Segregation Principle (ISP) – *Multi-Function Printer***

**Problem Statement:**  
A printing company has an IMachine interface that defines methods for Print(), Scan(), and Fax().

However:

* Simple printers only print.
* Office multi-function devices print and scan.
* Some specialized devices only fax.

When a simple printer implements IMachine, it is forced to also provide empty or error-throwing methods for scanning and faxing, which makes the code messy and confusing.

**Task:**

1. Break down the IMachine interface into smaller, more specific interfaces (e.g., IPrinter, IScanner, IFax).
2. Make sure each class implements only the interfaces it needs.

**Goal:**  
Ensure that no class is forced to depend on methods it does not use.

**5. Dependency Inversion Principle (DIP) – *Payment Processing***

**Problem Statement:**  
An online store has a PaymentProcessor class that directly depends on the PayPalService class. This works fine — until the business wants to support Stripe payments.

Currently, to switch payment providers, developers must modify the PaymentProcessor class, which tightly couples it to a specific payment service. This makes testing harder and reduces flexibility.

**Task:**

1. Create an IPaymentService interface that both PayPal and Stripe implementations will use.
2. Modify the PaymentProcessor so it depends only on the IPaymentService interface, not on concrete classes.
3. Use **dependency injection** to pass the desired payment service at runtime.

**Goal:**  
Make the high-level PaymentProcessor independent of low-level payment service details, so switching providers doesn’t require changing core business logic.