OOP, SOLID, and Design Pattern

NOTE: Apprentice should follow the criteria and complete the number of questions mentioned.

1. Object Oriented Programming

Criteria: Complete **all** the questions

- Create a Python class to represent a University. The university should have attributes like name, location, and a list of departments. Implement **encapsulation** to protect the internal data of the University class. Create a Department class that **inherits** from the University class. The Department class should have attributes like department name, head of the department, and a list of courses offered. Implement **polymorphism** by defining a common method for both the University and Department classes to display their details.

- Build a Python class to represent a simple banking system. Create a class for a BankAccount, and another for Customer. The BankAccount class should have a constructor to initialize the account details (account number, balance, account type). The Customer class should have a **constructor** to set the customer's details (name, age, address) and create a BankAccount object for each customer. Implement a **destructor** for both classes to display a message when objects are destroyed.

2. SOLID Principles

Criteria: Complete all the questions. No need to store any data. Just play around with objects

- **[Single-Responsibility Principle (SRP)]** Implement a simple program to interact with the library catalog system. Create a Python class **Book** to represent a single book with attributes: Title, Author, ISBN, Genre, Availability (whether the book is available for borrowing or not). Create another Python class **LibraryCatalog** to manage the collection of books with following functionalities:
  - Add books by storing each book objects (Hint: Create an empty list in constructor and store book objects)
  - get book details and get all books from the list of objects

  Lets say, we need a book borrowing process (what books are borrowed and what books are available for borrowing). Implement logics to integrate this requirement in the above system. Design the classes with a clear focus on adhering to the Single Responsibility Principle(SRP) which represents that **"A module should be responsible to one, and only one, actor."**

- **[Open-Closed Principle (OCP)]** Download the python file from this link. Suppose we have a Product class that represents a generic product, and we want to calculate the total price of a list of products. Initially, the Product class only has a price attribute, and we can calculate the total price of products based on their prices.

  Now, let's say we want to add a discount feature, where some products might have a discount applied to their prices. To add this feature, we would need to modify the existing Product class and the calculate_total_price function, which violates the Open/Closed Principle. Redesign this program to follow the Open-Closed Principle (OCP) which represents "Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification."

- **[Liskov Substitution Principle (LSP)]** Download the python file from this link. In this file, there is an implementation of a banking system for account handling. There is a savings account and a checking account class. The checking account inherits the savings account as both have the same functionality and the checking account allows overdrafts (allow processing transactions even if there is not sufficient balance). Redesign this program to follow the Liskov Substitution Principle (LSP) principle which represents that **"objects should be replaceable by their subtypes without altering how the program works".**

- **[Interface Segregation Principle (ISP)]** Download the python file from this link. Suppose we have an interface called PaymentProcessor that defines methods for processing payments and refunds. Then we have a class called OnlinePaymentProcessor that implements the PaymentProcessor interface. However, some parts of our system only need to process payments and do not handle refunds. Redesign this program to follow the Interface Segregation Principle (ISP) principle which represents that **"Clients should not be forced to depend upon methods that they do not use. Interfaces belong to clients, not to hierarchies."** (Hint: Create two different classes in which one class use interfaces for process payment and another class can process and refund payment both)

- **[Dependency Inversion Principle (DIP)]** Download the python file from this link. Suppose we have a NotificationService class that is responsible for sending notifications. The NotificationService class directly depends on the EmailSender class to send emails.

  In this implementation, the NotificationService class directly depends on the EmailSender class, which violates the Dependency Inversion Principle. The high-level NotificationService should not depend on the low-level EmailSender, as it tightly couples the classes together.

  Redesign this program to follow the Dependency Inversion Principle (DIP) principle which represents that **"Abstractions should not depend upon details. Details should depend upon abstractions."**

3. Design Patterns

Criteria: Complete all the questions

- **[Factory Design Pattern]** Build a logging system using the Factory Design Pattern. Create a LoggerFactory class that generates different types of loggers (e.g., FileLogger, ConsoleLogger, DatabaseLogger). Implement methods in each logger to write logs to their respective destinations. Show how the Factory Design Pattern helps to decouple the logging system from the application and allows for flexible log handling.
- **[Builder Design Pattern]** Design a document generator using the Builder Design Pattern. Create a DocumentBuilder that creates documents of various types (e.g., PDF, HTML, Plain Text). Implement the builder methods to format the document content and structure according to the chosen type. Demonstrate how the Builder Design Pattern allows for the creation of different document formats without tightly coupling the document generation logic.
- **[Singleton Design Pattern]** Implement a configuration manager using the Singleton Design Pattern. The configuration manager should read configuration settings from a file and provide access to these settings throughout the application. Demonstrate how the Singleton Design Pattern ensures that there is only one instance of the configuration manager, preventing unnecessary multiple reads of the configuration file.