Advanced OOP Exercises in C#

Exercise 1: Flight Booking System Using Inheritance and Polymorphism

Create a flight booking system with the following:

1. Class `Flight` with:
   - `FlightNumber` (flight number).
   - `Destination` (destination).
   - `Price` (base ticket price).
   - Method `CalculateFinalPrice()` -> returns the ticket price (overridden in subclasses).

2. Two subclasses inheriting `Flight`:
   - `EconomyFlight` -> applies a 10% discount to the base price.
   - `BusinessFlight` -> adds 30% extra to the base price.

3. Create a list of flights and print each flight's details with the final price.

Exercise 2: Implementing Strategy Pattern for a Payment System

Design a flexible payment system where users can choose their payment method:

1. Create an interface `IPaymentStrategy` with `ProcessPayment(decimal amount)`.
2. Implement three classes using this interface:
   - `CreditCardPayment` -> simulates credit card payment.
   - `PayPalPayment` -> simulates PayPal payment.
   - `BitcoinPayment` -> simulates cryptocurrency payment.
3. Create a `ShoppingCart` class with:
   - `TotalAmount` property.
   - `SetPaymentMethod(IPaymentStrategy strategy)` method.
   - `Checkout()` method to process payment using the selected strategy.

Test the system by creating a shopping cart and switching payment methods dynamically.

Exercise 3: Notification System Using Observer Pattern

Build a notification system where users subscribe to receive updates on order status.

1. Create an interface `IObserver` with `Update(string message)`.
2. Create an interface `ISubject` with:
   - `Attach(IObserver observer)` -> adds a subscriber.
   - `Detach(IObserver observer)` -> removes a subscriber.
   - `Notify(string message)` -> sends notifications to subscribers.
3. Create a `Order` class that implements `ISubject` and maintains a list of `IObserver` (subscribers).
4. Create a `Customer` class that implements `IObserver`, where the customer receives order notifications.
5. Test the system by adding customers as subscribers, updating an order, and observing the notifications.

Exercise 4: Dynamic Feature Addition Using Decorator Pattern

Build a beverage system where drinks can be customized using decorators.

1. Create an interface `IBeverage` with methods `GetCost()` and `GetDescription()`.
2. Create a `Coffee` class that implements `IBeverage` and returns `"Basic Coffee"` with its price.
3. Create two decorator classes (`Decorator`) to add features dynamically:
   - `MilkDecorator` -> adds `"Milk"` to the description and increases the price.
   - `SugarDecorator` -> adds `"Sugar"` to the description and increases the price.
4. Test the system by creating a coffee and adding multiple customizations.