Assignment2-CPS5301

Information

- Name: Cheng Bao
- No.: 1335784

Flaws of The Design

According to the description, codes and class diagram of this system, below are the flaws of this design:

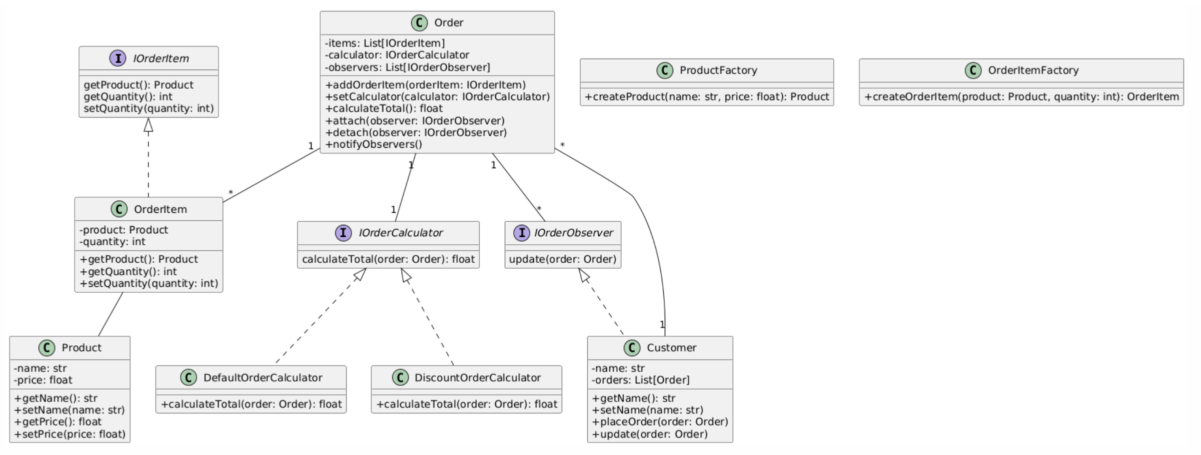
1. **Lack of Encapsulation**
    - Although in Python, there is no strong restriction about the public/private attributes, but mostly we will assign a private attributes of the class with "__" prefix(like `self.__name` instead of `self.name`). So I assume that the attributes in these classes are all public, which violates the principle of encapsulation.
    - Attributes should be private to prevent direct access and modification from outside the class. And if the attributes of the class are private, there should be getter/setter function created for attributes to be queried or settled.
2. **Tight Coupling**
    - The `Customer` class is tightly coupled with the `Order` class by directly use order class as a list. This will make it difficult to modify the `Order` class without affecting the `Customer` class.
    - Same situation happened between the `Order` class and the `Product` class.
3. **Lack of Flexibility and Scalability**
    - The design does not provide a way to scale or extend the functionality of orders or products. In another way of saying, not using any abstract class or interface in this design.

- The `Order` class does not provide any alternatives for different pricing strategies. It just provide a simple function of summing the product prices, which might not cover all scenarios (like discounts or taxes).
- Because of this design, it will cost significant changes when adding some new product types.

4. **Violation of Single Responsibility Principle (SOLID Principle)**
   - The `Order` class violates the Single Responsibility Principle by handling both the collection of products and the calculation of the total price.
5. **Violation of Dependency Inversion Principle (SOLID Principle)**
   - The `Customer` and `Order` classes depend on concrete classes rather than abstractions, which makes the system rigid and hard to maintain.

New Class Diagram

In terms of fixing the flaws mentioned above and using the **Factory Pattern**, **Observer Pattern,** and **Strategy Pattern,** the new class diagram of this system is shown as follows:



In this class diagram:

- All private attributes now have getter and setter functions.
- A new `OrderItem` class added, implemented from the `IOrderItem` interface, includes an instance of `Product` class with quantity, to manage the order items.
- The `Order` class was modified, by adding the instance of the `IOrderCalculator` interface and `IOrderObserver` interface, to seperate the responsibility of price

calculation and notification. The `Customer` class was modified simultaneously to fit in.

- New `ProductFactory` and `OrderItemFactory` classes were added, to achieve the Factory design pattern.

Design Pattern & Design Principle Explanation

Used Design Pattern Explanation

- **Factory Pattern**
    - `ProductFactory` and `OrderItemFactory` are responsible for creating `Product` and `OrderItem` objects, separating the creation and the usage.
- **Strategy Pattern**
    - The `IOrderCalculator` interface and its implementations `DefaultOrderCalculator` and `DiscountOrderCalculator` allow for different order calculation strategies to be chosen at runtime.
- **Observer Pattern**
    - The `IOrderObserver` interface and the `Customer` class, which implements this interface, allow the `Order` class to notify observers when its state changes.

Design Principle Explanation

- **Single Responsibility Principle**
    - Each class has a single responsibility. For example, `OrderItem` is only responsible for managing order items, and `OrderCalculator` is only responsible for calculating the total order price.
- **Open/Closed Principle**
    - The class diagram is open for extension but closed for modification, thanks to the use of interfaces and abstract classes. For instance, new calculator strategies can be added without modifying the `Order` class.
- **Liskov Substitution Principle**

    - Subclasses can be substituted for their base classes. In this class diagram, all classes interact through interfaces, so any class implementing the interface can replace another where the interface is used.
- **Interface Segregation Principle**
    - Each interface only contains the methods that the instance needs.
- **Dependency Inversion Principle**
    - High-level modules (like the `Order` class) depend on abstractions (like `IOrderCalculator` and `IOrderObserver`), not on concrete classes.