Class: ENSF 607

Assignment: Lab #4

By: Yajur Vashisht

Due Date: October 27th, 2023

<u>Exercise One:</u>

SOLID Principles are:
- Single Responsibility
- Open/Close
- Liskov Substitution
- Interface Segregation
- Dependency Inversion

For our project we have went with an E-commerce Product Management software in Java. We will be providing class diagrams, code, and appropriate use case for all the exercises in relation to our Product Management software theme.

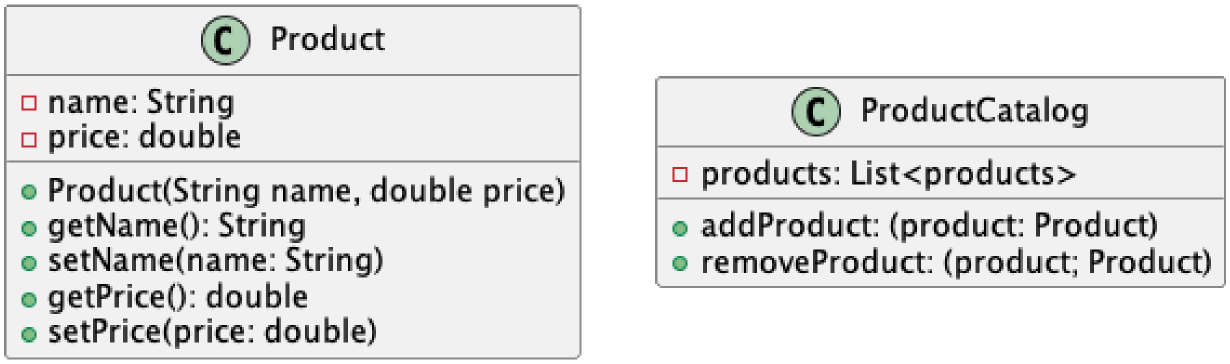

Figure 1: Single Responsibility Principle Class Diagram

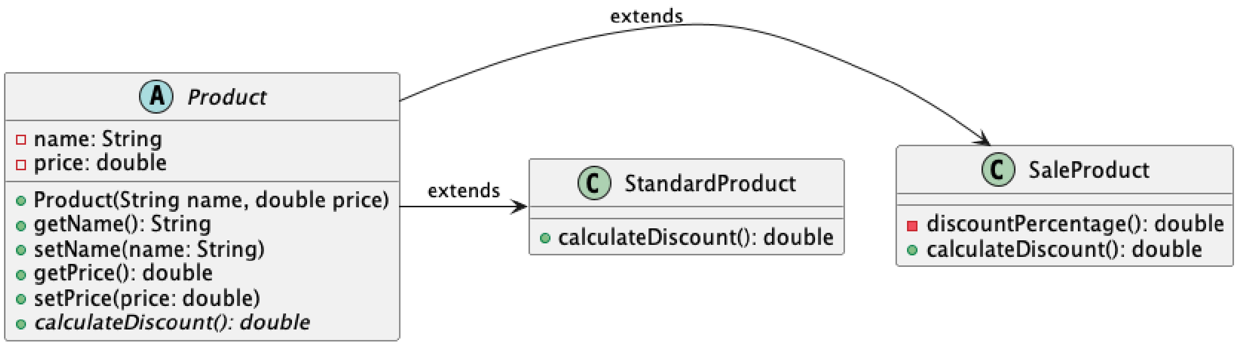

Figure 2: Open/Close Principle Class Diagram

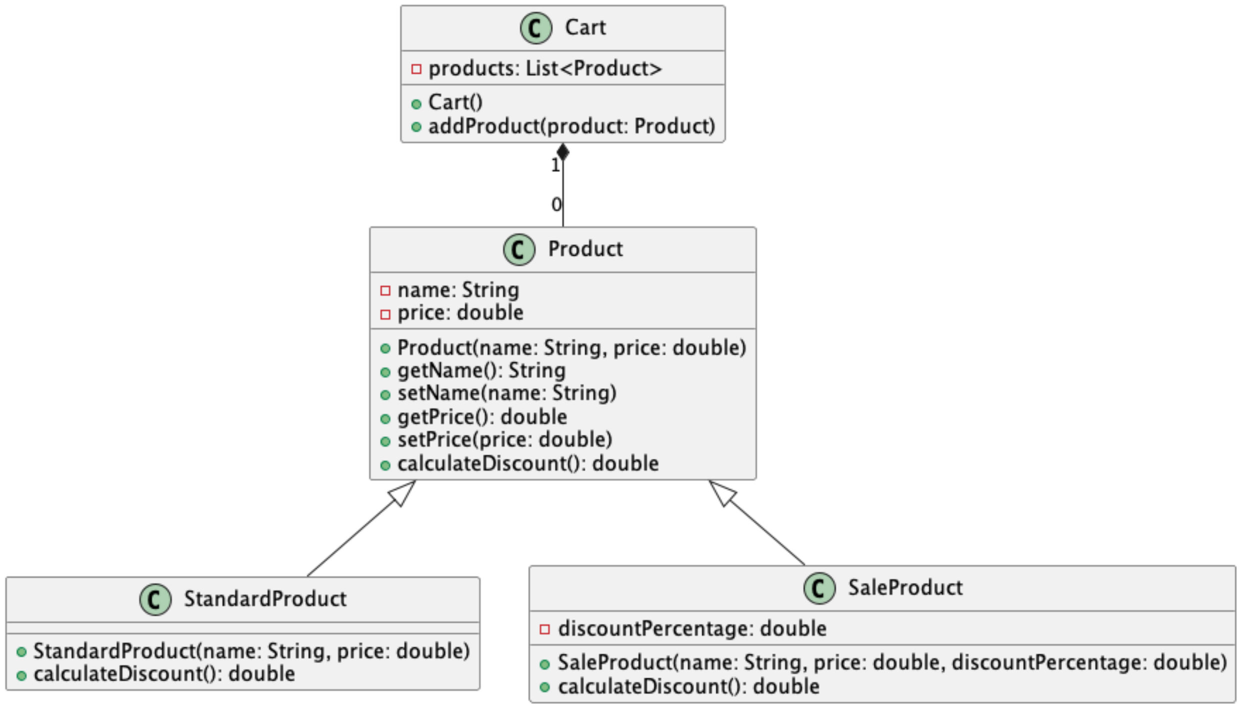
Cart
products: List<Product>
Cart()
addProduct(product: Product)
1
0
Product
name: String
price: double
Product(name: String, price: double)
getName(): String
setName(name: String)
getPrice(): double
setPrice(price: double)
calculateDiscount(): double
StandardProduct
StandardProduct(name: String, price: double)
calculateDiscount(): double
SaleProduct
discountPercentage: double
SaleProduct(name: String, price: double, discountPercentage: double)
calculateDiscount(): double

Figure 3: Liskov Substitution Principle Class Diagram

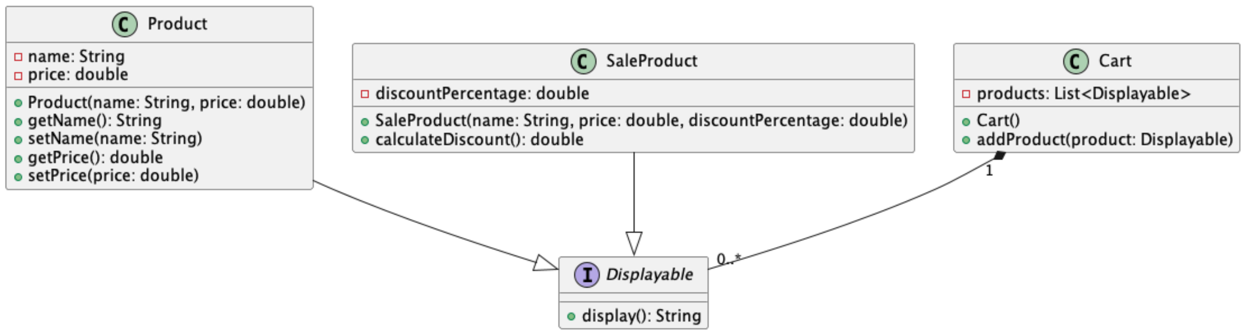
Product
name: String
price: double
Product(name: String, price: double)
getName(): String
setName(name: String)
getPrice(): double
setPrice(price: double)
SaleProduct
discountPercentage: double
SaleProduct(name: String, price: double, discountPercentage: double)
calculateDiscount(): double
Cart
products: List<Displayable>
Cart()
addProduct(product: Displayable)
1
0..*
Displayable
display(): String

Figure 4: Interface Segregation Principle Class Diagram

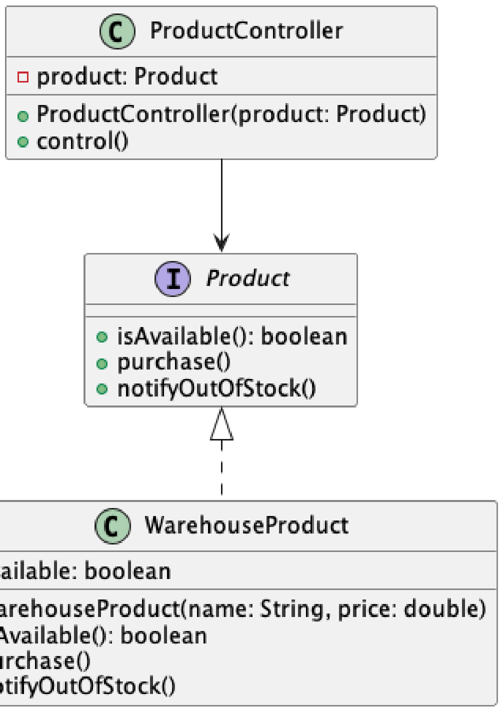

Figure 5: Dependency Inversion Principle Class Diagram

Exercise Two:

*Please see the code in the folder. Each principle has been divided into its own package with separate classes to demonstrate the SOLID principle in question.

Exercise Three:

Single Responsibility:

The single responsibility principle is demonstrated in the given example by the separation of roles between the Product and ProductCatalog classes. The Product class handles the product information, like name and price (without any additional responsibilities). The ProductCatalog class is responsible for managing a collection of products, which allows adding a product or removing one. This separation ensures that each class has a singular and distinct responsibility, aligning with the SRP.

Open/Close:

The Open/Closed Principle (OCP) is shown in Figure 2, where the existing code remains unchanged when introducing new functionality or classes. The base Product class serves as a foundation which can be extended to create new types of products, such as the StandardProduct and SaleProduct classes. These subclasses introduce new behaviors/features without modifications to the Product class, displaying the OCP by allowing for extensibility while maintaining code stability.

Liskov Substitution:

The Liskov Principle is demonstrated through the inheritance hierarchy, as shown in Figure 3. This hierarchy shows that subtypes, like SaleProduct, can seamlessly substitute their base types, such as Product, without causing issues or violating program correctness. This is in line with the LSP's rule of substitutability, where derived classes can be used interchangeably with their base classes, so there is a strong class hierarchy.

Interface Segregation:

To display the Interface Segregation Principle (ISP), Figure 4 introduces a Displayable interface, which specifies a single method, display(). This approach avoids creating "fat" interfaces that force implementing classes to provide unnecessary functionality. Each class only implements the methods it explicitly needs. An example is both Product and SaleProduct implement display(), but they only offer relevant functionality for their contexts. By adhering to ISP, the codebase remains streamlined, and classes are not loaded with unnecessary methods which promotes better code maintainability and adaptability.

Dependency Inversion:

The Dependency Inversion Principle (DIP) is shown in Figure 5, where high-level modules like the ProductController depend on abstractions, specifically the Product interface. This abstraction allows the ProductController to interact with any class implementing the Product interface, such as WarehouseProduct or SaleProduct. By relying on interfaces and abstractions, instead of direct dependencies, the DIP creates loose coupling between high-level and low-level modules, offering flexibility, ease of maintenance, and support for future changes in the codebase.