**Task 3 Software Design UML Class Diagram & Design Pattern**

**A screenshot of a black screen

Description automatically generated**

**User:**

* **The “user” class represents the user/seller of the online business system. He or she oversees the online store and can interact with the system and as well as perform various actions such as sending out orders and restocking items. The association between "User" and "OnlineStore" signifies that the user is responsible for managing the online store.**

**OnlineStore:**

* **The “OnlineStore” represents the online store itself. It serves as a central entity that connects the user, customer, and products. The association between "Customer” and “OnlineStore” signifies that the customer buys and orders the products through the online store. Lastly, the association between the “OnlineStore” and “Product” shows that the online store manages and provides access to the selling products.**

**Product:**

* **The “Product" class represents the products selling in the online store. Customers can browse and purchase these products. The association between "Inventory" and "Product" indicates that the inventory manages the availability and quantity of products.**

**Order:**

* **The "Order" class represents an order placed by a customer. It connects the "Customer" and "Product" classes, indicating that an order involves a specific customer and one or more products.**

**Customer:**

* **The "Customer" class represents a customer who browses the online store and places orders. The association between "Order" and "Customer" signifies that a customer can have multiple orders.**

**Inventory:**

* **The "Inventory" class represents the inventory of products available in the online store. It is associated with the "Product" class, indicating that the inventory manages the availability and quantity of products.**

**Payment:**

* **The "Payment" class represents a payment made for an order. It is associated with the "Order" class, indicating that a payment is related to a specific order.**

**Design Pattern**

**Below are a few design patterns that can be used for the class diagram:**

1. **Observer Design Pattern:**
   * **The observer design pattern can be used to establish the one-to-many dependency between objects, allowing for automatic notification of changes in the state of a subject to multiple observers. In this case, the classes "Order," "Customer," and "Product" can act as subjects, while other classes such as "Payment" and "Inventory" can act as observers, being notified when changes occur.**
   * **The subject classes (e.g., "Order," "Customer," and "Product") would implement a subject interface, and the observer classes (e.g., "Payment" and "Inventory") would implement an observer interface. The subject classes would maintain a list of registered observers and notify them of any relevant changes.**
2. **Factory Method Design Pattern:**
   * **The Factory Method pattern can be used to provide an interface for creating objects while allowing subclasses to decide the exact class to instantiate. This pattern is useful when there is a need to create different types of objects with a common base class or interface. In the given class diagram, the "Product," "Order," and "Payment" classes could be created using the Factory Method pattern, allowing for flexibility and extensibility in object creation.**
   * **In this case, the creator classes (e.g., "Creator\_Product," "Creator\_Order," and "Creator\_Payment") would define factory methods to create instances of the corresponding concrete product classes (e.g., "ConcreteProduct\_Product," "ConcreteProduct\_Order," and "ConcreteProduct\_Payment"). Clients would then use the creator classes to create the desired objects.**
3. **Strategy Design Pattern:**
   * **The Strategy pattern can be used to encapsulate interchangeable behaviors and allow clients to choose a behavior dynamically. In the given class diagram, there may be different payment strategies or shipping strategies that can be applied to an order. The Strategy pattern allows the flexibility to choose a specific strategy at runtime.**
   * **A strategy interface can be defined (e.g., "PaymentStrategy") and implemented to concrete strategy classes (e.g., "CreditCardPaymentStrategy," "PayPalPaymentStrategy," etc.) that encapsulate specific payment behaviors. The "Order" class would have a reference to the selected payment strategy and can dynamically switch between different strategies.**

**Strategy Design Pattern:**

**The chosen pattern for the class diagram is strategy.**

**A screenshot of a computer screen

Description automatically generated**

* **The OnlineStore class represents the online store itself sending the details of the product that the customer wants to order.**
* **The Order class represents an order of a product placed by the customer.**
* **The Payment class is the strategy interface defining the contract for different payment strategies.**
* **The concrete strategy classes, including "Credit\_Card," "EWalletStrategy," “Debit\_Card”, “OnlineBanking”, and "CashOnDelivery," implement the "Payment" interface, encapsulating specific payment behaviors.**