OOP Project Report

Group Members:

Bilal Usmani CT-24071

Maaz Khan CT-24076

Owais Kamran CT-24085

Syed Faham CT-24069

Introduction to the Problem statement

The objective of this project is to design a system that handles order placement, payment processing using different payment methods, and order tracking.

This system should:

- Allow users to browse products.
- Place orders for desired products.
- Choose and process payments through different methods (e.g., cash or card).
- Enable order tracking by users and store administrators.

Design Patterns

1. Strategy Pattern – For Payment Processing

Pattern Category: Behavioral
Where Applied:

- PaymentMethod is an interface with a pay() method.
- CardPayment and CashPayment are concrete implementations of PaymentMethod.
- Order class uses a PaymentMethod* pointer, allowing it to interact with any payment method type polymorphically.

Why this Design Pattern is used:

- The system must support multiple payment methods (like card and cash), each with its own way of processing a payment.

- The Strategy pattern allows these different algorithms (payment processes) to be encapsulated in their own classes.
- The Order class only holds a reference to the base class PaymentMethod, not to specific types like CashPayment or CardPayment.

Benefits:

- **Flexibility:** At runtime, the system can choose which payment method to use.
- **Extensibility:** New payment types like CryptoPayment can be added without touching existing code in Order.
- **Clean separation:** Business logic (ordering) is separated from the algorithm (how payment is processed).

2. Singleton Pattern – For Store Class

Pattern Category: Creational
Where Applied:

- The Store class is responsible for managing all products and orders in the system. Since there is only one store, it is implemented as a Singleton.

Why this Design Pattern is used:

- In this system, there is only one Store, which holds the complete list of products and orders.
- If multiple instances of Store were created, it would lead to data inconsistency, such as duplicate inventories or disconnected orders.
- The Singleton pattern ensures there is only one instance of Store shared globally.

Benefits:

- Global accessibility: Any User or Admin can access the same Store instance.
- Resource optimization: Avoids unnecessary duplication of resources.

3. Factory Method Pattern – For Creating Payment Method Instances

Pattern Category: Creational

Where Applied:

- PaymentMethod is an abstract class that defines a common interface with the pay() method.
- CardPayment and CashPayment are subclasses that implement the specific behavior of pay().
- The Order class holds a pointer to PaymentMethod (paymentMethod: PaymentMethod*), which allows for assigning any type of payment dynamically.

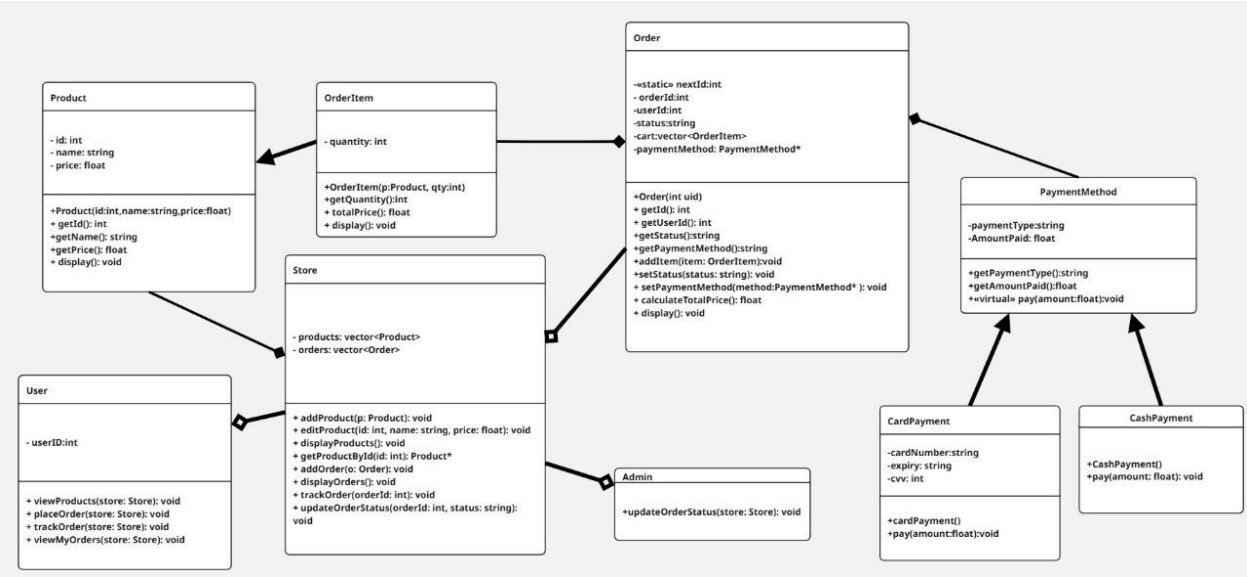
Why this Design Pattern is used:

- A factory method can be implemented to create specific payment method objects (e.g., CardPayment, CashPayment) based on input like user selection.
- This encapsulates the creation logic and avoids coupling the Order class directly to specific payment classes.
- Supports the Open/Closed Principle – new payment types (e.g., UPIPayment, CryptoPayment) can be added without modifying existing code.

Benefits:

- **Scalability**
  New payment methods can be added without modifying the core system logic — just by creating new subclasses and updating the factory.
- **Code Reusability and Maintainability**
  Centralized object creation logic makes the codebase cleaner and more manageable.
- **Improved Flexibility**
  Different parts of the system can request objects of type PaymentMethod without knowing the exact class.
- **Easier Testing**
  Test stubs or mocks of PaymentMethod can be easily injected for testing purposes, since the system works on an abstraction.

UML Diagram



Conclusion

This project models a real-world order management system that handles order placement, payment processing using multiple payment types, and order tracking. The design leverages object-oriented design principles and patterns to ensure scalability, flexibility, and code maintainability.

The following design patterns were identified and utilized:

- **Strategy Pattern:** Used for managing different payment behaviors (CardPayment, CashPayment) under a common interface.
- **Singleton Pattern:** Can be used to ensure that the Store object exists only once system-wide.

- **Factory Method Pattern:** To centralize and abstract the creation of payment method objects without exposing instantiation logic to client classes like Order.

These patterns promote:

- **Extensibility** (e.g., easily adding new payment methods),
- **Low coupling** (e.g., Order doesn't need to know the details of how payments are processed),
- **Reusability** (e.g., same logic reused for different payment types),
- **Maintainability** (e.g., centralized object creation through factories).