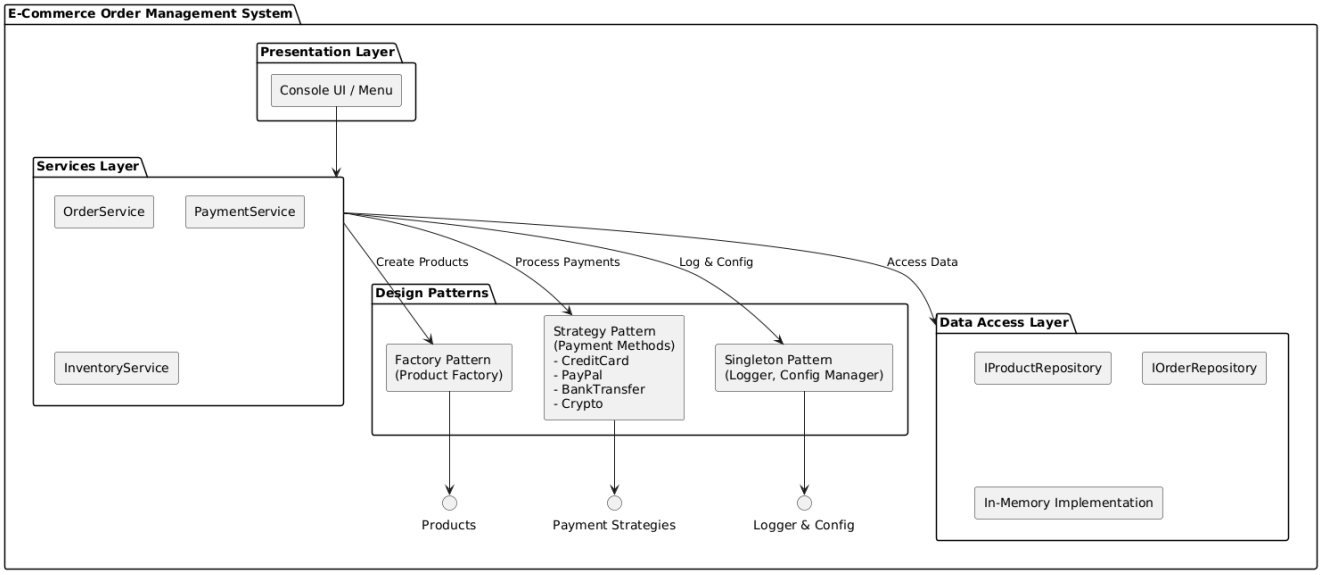
Assignment 3(6606)

Q1 : High Level Architecture Diagram



Q2 : Pattern Implementation

1. Factory Pattern

I used the Factory Pattern to create products based on their category. There is a ProductFactory class that makes the right product type like Electronics, Clothing, Books, or HomeGarden. Each product has its own details, like Electronics have warranty and Books have ISBN. If someone gives a wrong category, the factory handles it nicely without crashing.

2. Singleton Pattern

For things like Configuration and Logger, I used the Singleton Pattern. This makes sure only one instance of these classes exists throughout the program. They are thread-safe, so even if many parts of the program use them at the same time, they work without problems.

3. Strategy Pattern

The Strategy Pattern is used for payment methods. There is an interface for payment, and different classes implement it for Credit Card, PayPal, Bank Transfer, and Crypto payments. Each class has its own way of checking if the payment details are valid. The program can switch between these payment methods easily while running.

4. Repository Pattern

The Repository Pattern helps to manage data like products and orders. There are generic repository interfaces with common functions like add, update, delete, and get. Specific repositories for products and orders have extra functions, for example, to get products by category or orders by customer. The

data is stored in memory (no database), so it's simple and fast. I just implement but I didn't integrate this pattern in code.

3. SOLID Principles Application

Single Responsibility Principle:
Each class has a single responsibility, e.g., Product classes handle product data, Logger handles logging, Payment Strategies handle payment processing.

Open/Closed Principle:
New product categories or payment methods can be added without modifying existing classes, thanks to the Factory and Strategy patterns.

Liskov Substitution Principle:
Derived product classes and payment strategies can substitute their base interfaces without altering correctness.

Interface Segregation Principle:
Interfaces like IProductRepository and IOrderRepository are specific, avoiding bloated contracts.

Dependency Inversion Principle:
High-level modules (services) depend on abstractions (repositories, factories, strategies) rather than concrete implementations, facilitating testability and flexibility.

4. Future Enhancements

1. Database Integration instead of Seed(Mock) Data

2. Cancel Order System(Status Extended)

3. Payment Complete Integration (Stripe like in MERN)

4. Provide attractive UI instead of Console Based System

5. Add users(Like Admin & Users Separately a user can see his own details (Privacy))

6. Register & Sign Up Users

Etc

5. Problems & Solutions

Problem # 01 : Sometimes it was tricky to handle different product categories with unique properties. For example, Electronics have warranty but Books have ISBN. Use enum but I didn't user like before specially in Switch Cases.

Problem # 02 : Implementing the configuration manager and logger as thread-safe singletons was hard. I had to make sure that when many parts of the program use them at the same time, no data gets messed up. Specially how to pass in constructor. Privacy in ids

Problem # 03 : Each payment method needs different validation rules. Writing separate logic for credit cards, PayPal, bank transfer, and crypto was challenging. Also, switching between payment methods dynamically in the code required careful design.

Problem # 04 : Designing a generic repository that works for products and orders was complicated. I tried to simulate data storage carefully so that adding, updating, and deleting worked correctly.

Problem # 05 : Keeping the code organized so that business logic, data access, and user interface didn't mix was sometimes difficult, especially when handling orders and payments.

Problem # 06 : Making sure all inputs (like product details or payment info) are correctly validated to avoid crashes was a very very very time-consuming.

Problem # 07 : Writing code that strictly follows SOLID principles while still making it work and keeping it simple was challenging, especially in the beginning.

Problem # 08 : Testing all the different order scenarios, payment successes/failures, and inventory updates thoroughly was tough and required a lot of time.

& Many More !!!