Design Rationale Document 3307 – Muskan Bhatia, Lakshya Prasad

Pattern Justification

1. **Object-Oriented Programming (OOP) Principles**

Inheritance: Implemented by having a User class act as the base class for both Buyer and Seller. Common attributes (e.g., username, password, email) and methods (login(), logout()) are centralized in User, while role-specific functions—addProduct() for Seller and placeOrder() for Buyer—extend its behavior. This simplifies role-based logic and promotes code reuse across modules.

Polymorphism: Methods such as displayDashboard() and processOrder() are overridden in subclasses so each role performs actions suited to its privileges while preserving a common interface. This design allows adding new roles (e.g., Admin) without changing existing code.

Encapsulation: All data members are kept private with controlled access via getters/setters. This protects sensitive information (like user credentials and payment data) and maintains clear boundaries between modules.

Singleton Pattern (Database Manager)

Why: A single, shared database instance was needed to manage products, users, and orders consistently throughout the system. Using multiple DB objects could cause race conditions and data inconsistency; the Singleton pattern prevents that.

Implementation:

```
class DB { public: static DB& instance() { static DB inst; return inst; } std::vector<Product> products; private: DB() = default; DB(const DB&) = delete; DB& operator=(const DB&) = delete; };
```

Result: Guarantees only one connection to the mock in-memory database. Provides a globally accessible data layer for the UI and backend logic.

Repository / Data Access Pattern

Why: To separate database operations from business logic, ensuring cleaner and more maintainable code.

Implementation: Logical “repositories” like UserRepository and ProductRepository interact with DB::instance() for CRUD operations. The UI or service layers never directly modify the database—this improves modularity and testability.

Separation of Concerns (Architecture)

Controller – Service – Repository Structure: Controllers handle user interactions via Qt signals/slots (e.g., login button, add to cart). Services contain core business logic for validation, checkout, and stock updates. Repositories perform data management through the Singleton DB. This modular layering simplifies debugging and scaling: changes in one layer don’t affect others.

User Interface Design (Responsive Prototype)

A Qt Widget-based UI was used to simulate a real marketplace flow: Login → Catalog → Cart → Checkout.

The MainWindow acts as the application entry point and navigates to these pages. Pages are designed with Qt Layouts for automatic resizing and consistent alignment. Buttons disable during processing (e.g., payment) to prevent duplicate actions. This low-fidelity UI demonstrates user flow without complex styling or backend integration.

Challenges and Solutions

1. **Qt Signal/Slot Integration**

Challenge: UI buttons initially failed to trigger backend actions.
Solution: Introduced controller classes to connect signals to logic slots and verified connections in debug mode.

2. **Database Consistency and State Management**

Challenge: Simultaneous updates to cart and product stock caused inconsistencies.
Solution: Used the Singleton DB to manage shared state and atomic updates.

3. **Order Checkout Simulation**

Challenge: Payment and stock updates had to appear atomic without a real API.
Solution: Added a mock PaymentService that returns “APPROVED” and uses transaction-style updates in DB.

4. **Limited Time for UI Design**

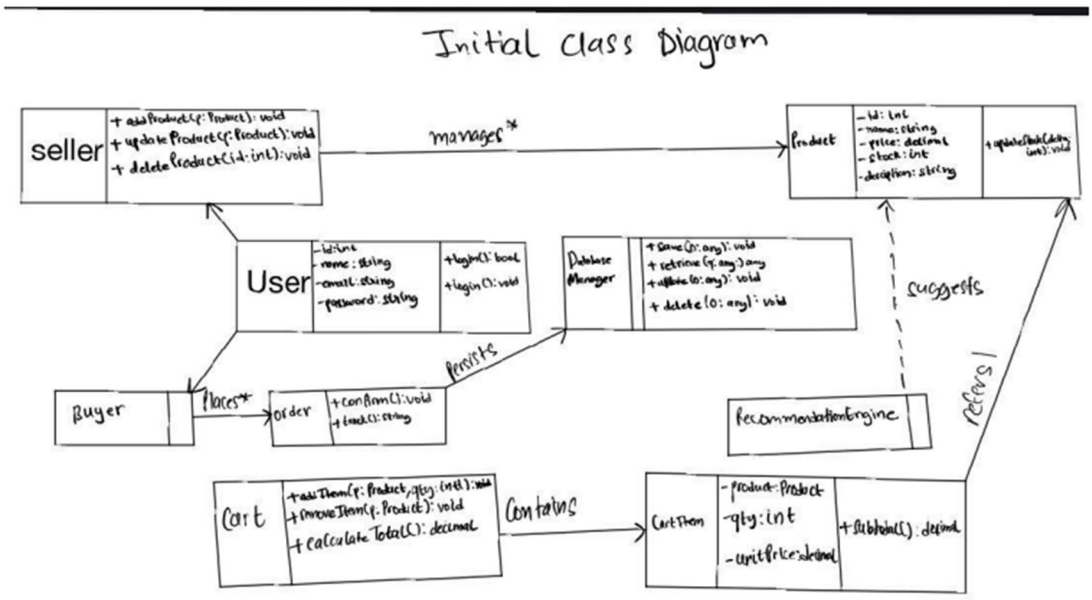
Challenge: Balancing C++ logic and Qt layout creation within the deadline.
Solution: Used basic layouts and a minimal color scheme to prioritize functionality over appearance.

5. **Integration of Design Pattern**

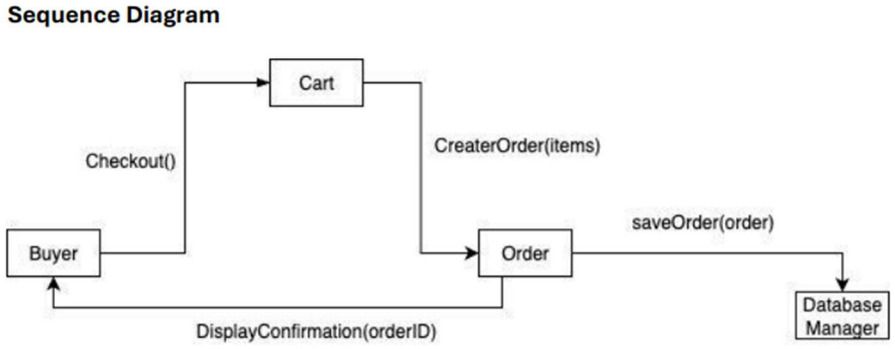
Challenge: Understanding proper use of Singleton vs Factory patterns.

Solution: Implemented Singleton for A2 and planned Factory/Observer for A3 (High-Fidelity Prototype).

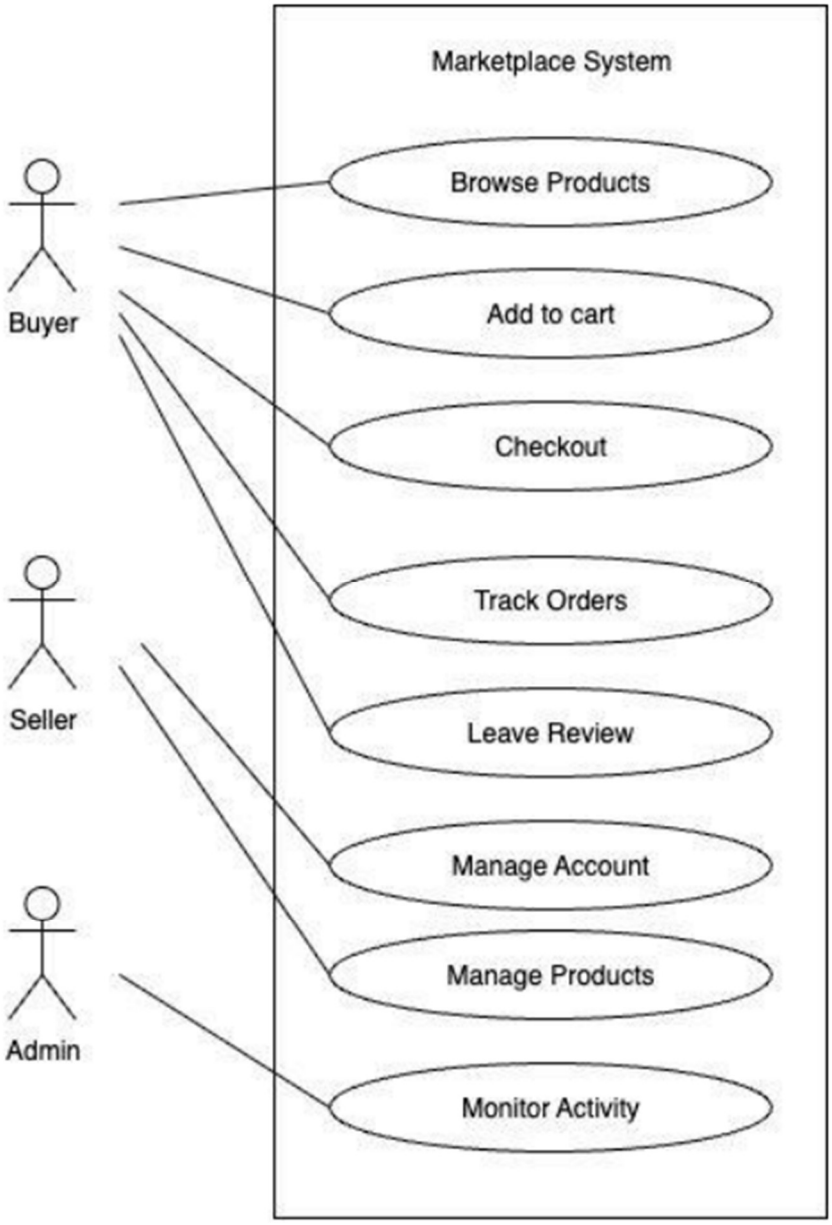
INITIAL CASE DIAGRAM:



SEQUENCE DIAGRAM:



USECASE DIAGRAM:



End of Document