**Lab Final Question on Software Design Pattern**

1. In a large e-commerce application, different payment gateways need to be supported (e.g., credit cards, PayPal, cryptocurrency) with the ability to easily add new payment methods in the future. Implement the **Strategy pattern** to encapsulate the various payment methods as separate strategies. The payment strategy can be dynamically selected at runtime based on user preferences or configuration.
2. A social media platform needs to handle user notifications efficiently, delivering them in real-time while minimizing system resources. Implement the **Observer pattern** where users subscribe to specific events or updates, they are interested in. When an event occurs, the subscribers are notified and receive the relevant notifications in real-time.
3. Developing a weather application that provides weather data from multiple sources (e.g., APIs from different weather providers) while ensuring that the application is decoupled from the specific weather services. Use the **Adapter pattern** to create adapters for each weather service API, converting their data into a common format that the application can consume.
4. Building a complex workflow system that involves a series of tasks with dependencies and the ability to easily modify and extend the workflow. Use the **Composite pattern** to represent each task as a component and compose them into a tree-like structure that represents the entire workflow.
5. A car manufacturing company needs to manage the production of different car models with varying features and options. Company needs to separate the construction of complex car objects from their representation, and to build different configurations of cars using the same construction process. Use the **Builder pattern** to simulate the system.
6. An online booking system for hotels needs to handle room availability and reservations efficiently. Use **Flyweight pattern** to represent shared room information (e.g., room type, price) as lightweight flyweight objects, reducing memory usage and improving performance when handling a large number of room instances.
7. Developing a caching system that can efficiently store and retrieve frequently accessed data to improve application performance. Implement the **Proxy pattern** where the proxy acts as a cache for the underlying data source. The proxy intercepts requests and checks if the data is already cached before fetching it.
8. Implement a **Singleton pattern** to create a Logger class that ensures only one instance of the class is created.
9. Implement the **Factory Method design pattern** to create objects of different subclasses based on a common interface.
10. Implement a **Command pattern** to encapsulate a request as an object, thereby allowing clients to parameterize clients with queues, request or log requests, and support undoable operations.
11. In a corporate leave approval system, employees submit leave requests that must be approved based on the number of days requested. A Manager can approve up to 5 days, a Director can approve up to 10 days, and anything beyond that requires CEO approval. Instead of hardcoding who approves what, implement the **Chain of Responsibility pattern**, where each role checks if it can handle the request and, if not, passes it to the next level in the chain. This approach may make the approval process flexible, scalable, and easy to maintain.
12. Implement the **Interpreter Design Pattern** for a rule-based access control system to evaluate user permissions expressed as logical expressions like "Admin OR (Editor AND NOT Guest)". Each role or condition (e.g., Admin, Editor, Guest) is represented as a terminal expression, while logical operators like AND, OR, and NOT are non-terminal expressions. When a user attempts to perform an action, the system interprets the permission rule based on the user's roles, allowing dynamic and flexible access control without hardcoding complex permission logic.
13. Implement the **Mediator Design Pattern** for a chat room application to manage communication between multiple users without them directly referencing each other. Instead of each user (or colleague) sending messages directly to others, they communicate through a central ChatRoom (Mediator), which handles message routing, broadcasting, and user management. This decouples the participants, simplifies interactions, and makes the system easier to maintain and extend as new users or features are added.