Design Patterns for Encompass

Table of Contents:

1. Introduction:

This document outlines the design patterns selected for our app and provides a rationale for each selection. Design patterns offer proven solutions to common design problems and promote modularity, extensibility, and maintainability in software development.

2. Design Pattern 1: Strategy Design Pattern

2.1 Explanation:
The Strategy Design Pattern allows for the selection of different algorithms or strategies at runtime. In the context of our app, this pattern is used to choose the relevant AI algorithm for recommending media, groups, and posts to users.

2.2 Benefits:
- Enables dynamic algorithm selection based on runtime conditions or user preferences.
- Promotes code reusability by encapsulating specific algorithms into separate strategy classes.
- Facilitates easy addition or modification of algorithms without impacting the client code.

2.3 Use Case:
In app, the Strategy Design Pattern is employed to recommend media, groups, and posts to users. Based on various factors such as user preferences, behaviour, or community interactions, the appropriate AI algorithm is dynamically selected at runtime to provide personalised recommendations for each section of the homepage.

3. Design Pattern 2: Observer Design Pattern

3.1 Explanation:
The Observer Design Pattern establishes a one-to-many relationship between objects, where multiple observers are notified automatically of changes in a subject. For our app, this pattern is used to notify users who have clicked the notification button about new events and messages in a specific community.

3.2 Benefits:
- Supports real-time notifications to keep users informed of updates and events.
- Facilitates loose coupling between the subject (community) and its observers (users), promoting modularity.
- Simplifies the addition or removal of observers without affecting the subject or other observers.

3.3 Use Case:
In our app, the Observer Design Pattern is applied to notify users who have subscribed to a specific community. When new events or messages occur within that community, the subject (community) notifies all registered observers (users who clicked the notification button), ensuring they receive real-time updates.

4. Design Pattern 3: Proxy Design Pattern

4.1 Explanation:
The Proxy Design Pattern provides a surrogate or placeholder for another object, controlling access to it. In our app, the Proxy Pattern is utilised to store and update the states of the Feed, otherProfile, and Community pages when the user performs actions such as refreshing the Feed, clicking on a profile, or accessing a community.

4.2 Benefits:
- Enhances performance by avoiding unnecessary resource loading or heavy operations until needed.
- Provides a level of indirection for accessing objects, enabling additional functionality such as caching or lazy loading.
- Simplifies the management of object states and updates.

4.3 Use Case:
In our app, the Proxy Design Pattern is employed to store and manage the states of different pages. For example, when a user refreshes the Feed, the Proxy for the Feed page

retrieves and updates the content, minimising unnecessary server calls. Similarly, the Proxies for otherProfile and Community pages handle their respective states, providing efficient access and updating.

5. Design Pattern 4: Factory Design Pattern

5.1 Explanation:
The Factory Design Pattern provides an interface for creating objects, allowing subclasses to determine the class to instantiate. In our app, the Factory Pattern is utilised for user creation, as we have different types of users (e.g., admin, moderator, regular user), and we want to create them without modifying the client code.

5.2 Benefits:
- Promotes loose coupling between the client code and the specific user classes, enhancing flexibility.
- Supports the addition of new user types without modifying existing code.
- Centralizes user creation logic, making it easier to manage and maintain.

5.3 Use Case:
In our app, the Factory Design Pattern is applied to create different types of users. When a new user is registered, the factory method determines the appropriate user class based on the user type selected. This allows us to accommodate various user roles without the need to modify the client code, providing scalability and maintainability.

6. Conclusion:
By incorporating these design patterns into our app, we can enhance its functionality, modularity, and maintainability. The Strategy Pattern enables suitable recommendations for context of recommendations, the Observer Pattern ensures real-time notifications, the Proxy Pattern manages object states efficiently, and the Factory Pattern facilitates flexible user creation for the different tiers of users. These patterns collectively contribute to a robust and extensible app.