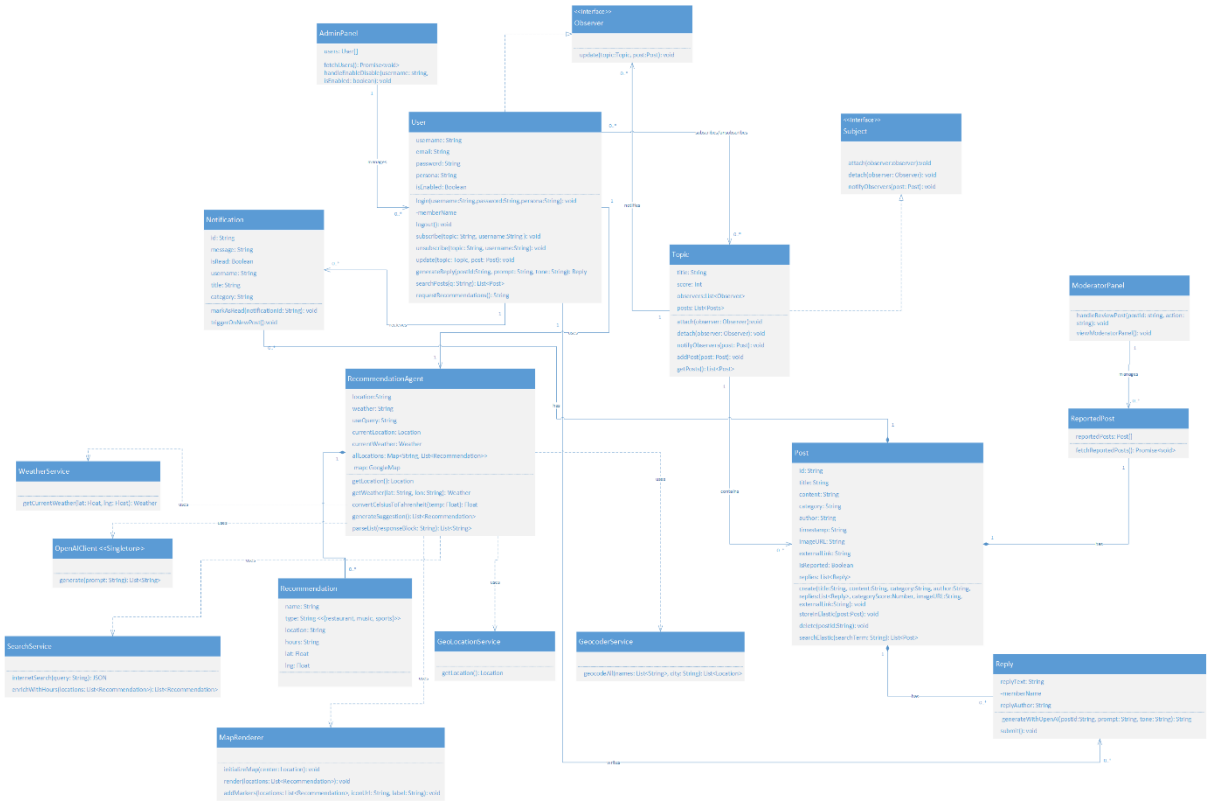
Assignment_5

Refined Class Diagram:



5. Document and Annotate 2 Creational Design Patterns that are used in your UML Design Class Diagram.

1. **Singleton Pattern**
   The Singleton pattern ensures that a class has only one instance throughout the application and provides a global access point to this instance. In the diagram, the class named OpenAIClient explicitly uses the Singleton pattern (marked clearly by <<Singleton>>).
   The reason behind using the Singleton pattern here is that interacting with the OpenAI API typically involves managing an API key, connection settings, session information. Creating multiple instances might cause inconsistencies, redundant API calls, or difficulties in resource management. Thus, by ensuring a single instance, system guarantees controlled and consistent interactions with the external OpenAI service. We made it singleton inorder to control API calls centrally and efficiently, to avoid unnecessary or duplicate API instances, to centralize management of API configurations and keys.

   Annotation:
   Creational Pattern: Singleton
   Ensures exactly one instance of OpenAIClient exists, offering centralized management of external API calls and resources.

2. **Factory Method Pattern**

   The Factory Method pattern provides an interface (a method) for creating objects within a class, letting subclasses or internal methods alter the type of objects created without exposing the creation logic to client code.

   In your design diagram, this pattern is used within the Post class through a method called create(). This method centralizes all logic related to creating and initializing a Post object, such as setting the post's id, title, content, timestamp, category, author, etc. By encapsulating this logic within a single factory method, the complexity involved in creating a properly configured Post object is abstracted away from other parts of your application, thus simplifying object creation and ensuring consistency. We made use of this in order to simplifies object creation, particularly when creating Post objects requires specific initialization logic, Abstracts and centralizes instantiation logic, making maintenance easier, Allows potential extension or modification of creation logic without affecting client code.

   Annotation:

   Creational Pattern: Factory Method

   Encapsulates complex Post creation logic, simplifying and centralizing object initialization to ensure consistent object states.

6. Documentation and Annotation of 2 Structural Design Patterns

1. **Facade Pattern**

   The Facade pattern provides a simple, high-level interface to a complex subsystem involving multiple interacting components. This hides the complexity and reduces the dependencies between the subsystem components and the client code that uses them.

   In our diagram, the RecommendationAgent acts as a Facade. It coordinates complex interactions among multiple services, such as WeatherService, OpenAIClient, GeolocationService, and SearchService. Rather than forcing client code (such as UI or higher-level business logic) to individually manage interactions with all these services separately, the RecommendationAgent simplifies the interaction by aggregating these complex tasks into straightforward, simplified methods. We made use of this in order to, simplifies usage by hiding complex interactions between multiple underlying services, Reduces the client's direct dependency on the detailed implementation of each subsystem, Enhances maintainability by centralizing interaction logic in one place.

   Annotation:

   Structural Pattern: Façade

   Provides a simplified, unified interface to complex recommendation logic involving multiple subsystem services, thus hiding complexity from clients.

2. **Composite Pattern**

   The Composite pattern is used to represent objects in hierarchical structures, allowing the client to treat individual objects and groups of objects uniformly. In our diagram, the Post and Reply classes form a composite structure. A Post contains multiple replies (List<Reply>). This hierarchical structure allows treating both the Post object and the collection of its Reply objects as a single entity, simplifying operations such as rendering or data management.

   This way, whether you have a single reply or many replies, your system can treat these uniformly, significantly reducing complexity and enhancing flexibility. We made use of this in order to manage hierarchical relationships between posts and their replies easily, Enables uniform processing and treatment of individual or grouped items (e.g., rendering posts with replies), Simplifies extension or modification of hierarchical content structures.

Annotation:
Structural Pattern: Composite
Enables handling Posts and Replies uniformly as hierarchical compositions, simplifying management of structured content.

7. Documentation and Annotation of 2 Behavioral Design Patterns

1. **Command Pattern**
   The Command pattern encapsulates requests as separate objects, enabling clients to issue parameterized requests without coupling directly to specific implementations or operations. Though your UML diagram explicitly shows the command concept via methods rather than separate command objects, the underlying idea is preserved by encapsulating different moderation actions into methods.
   In your specific implementation, the handleReviewPost(postId: string, action: string) method within the ModeratorPanel effectively encapsulates moderation commands such as approving, rejecting, or deleting posts. Each call to this method represents an independent moderation action, with parameters specifying the exact operation to perform. Although these method calls don't create separate command objects explicitly, each moderation action (approve, reject, delete) is treated logically as an independent "command-like" object that can be parameterized, managed, logged, or potentially extended into a more formal command-object pattern later. We made use of this in order to moderation actions become loosely coupled and easy to extend or modify, Future enhancements, such as maintaining a history of actions (logging) or enabling undo/redo functionality, can be seamlessly added, Commands can potentially be queued, scheduled, or handled asynchronously, further emphasizing the flexibility gained by encapsulating these actions. We made use of this in order to encapsulates moderation actions rather than performing direct manipulations, Offers flexible moderation, allowing parameterization and easier addition of new moderation actions, Provides foundational flexibility, supporting potential extensions (e.g., logging or undo functionality).

   Annotation:
   Behavioral Pattern: Command
   Encapsulates moderation actions (approve, reject, delete) as parameterized commands,providing flexibility to manage, log, or extend moderation capabilities without tightly coupling the operations.

2. **Observer Pattern**
   The Observer pattern establishes a one-to-many relationship where a Subject maintains a list of subscribers (Observers) and automatically notifies them whenever its state changes. In our UML diagram, this pattern clearly involves:
   - Subject interface: defines methods to attach, detach, and notify observers.
   - Observer interface: provides an update mechanism for observers.
   - Concrete Subject: The Topic class explicitly maintains observers (such as User, AdminPanel) and triggers notification upon changes (e.g., a new post being created).
   - Concrete Observers: User and AdminPanel classes implement observer logic, updating themselves automatically when notified.

   In a practical implementation, this Observer pattern can leverage event-driven programming techniques such as events/listeners. Specifically, when a new post is created in the Topic class, it triggers an event through the notifyObservers(post: Post) method, effectively "broadcasting" this event to all subscribed observer objects. Observers then respond to this event automatically, executing appropriate actions such as generating user notifications or updating admin dashboards.This approach decouples observers and subjects, allowing them to vary independently, Supports easy addition/removal of observers without affecting the subject, Promotes event-driven programming practices, leveraging events/listeners to facilitate communication between components. We made use

of this in order to enables real-time notifications and updates in a decoupled manner, Utilizes events/listeners for efficient, scalable handling of state changes, Facilitates responsiveness and flexibility within the application.
Annotation:
Behavioral Pattern: Observer
Uses an event/listener mechanism: the concrete Subject (Topic) broadcasts state changes to observers (User, AdminPanel) via events, allowing automated, decoupled, and flexible updates.