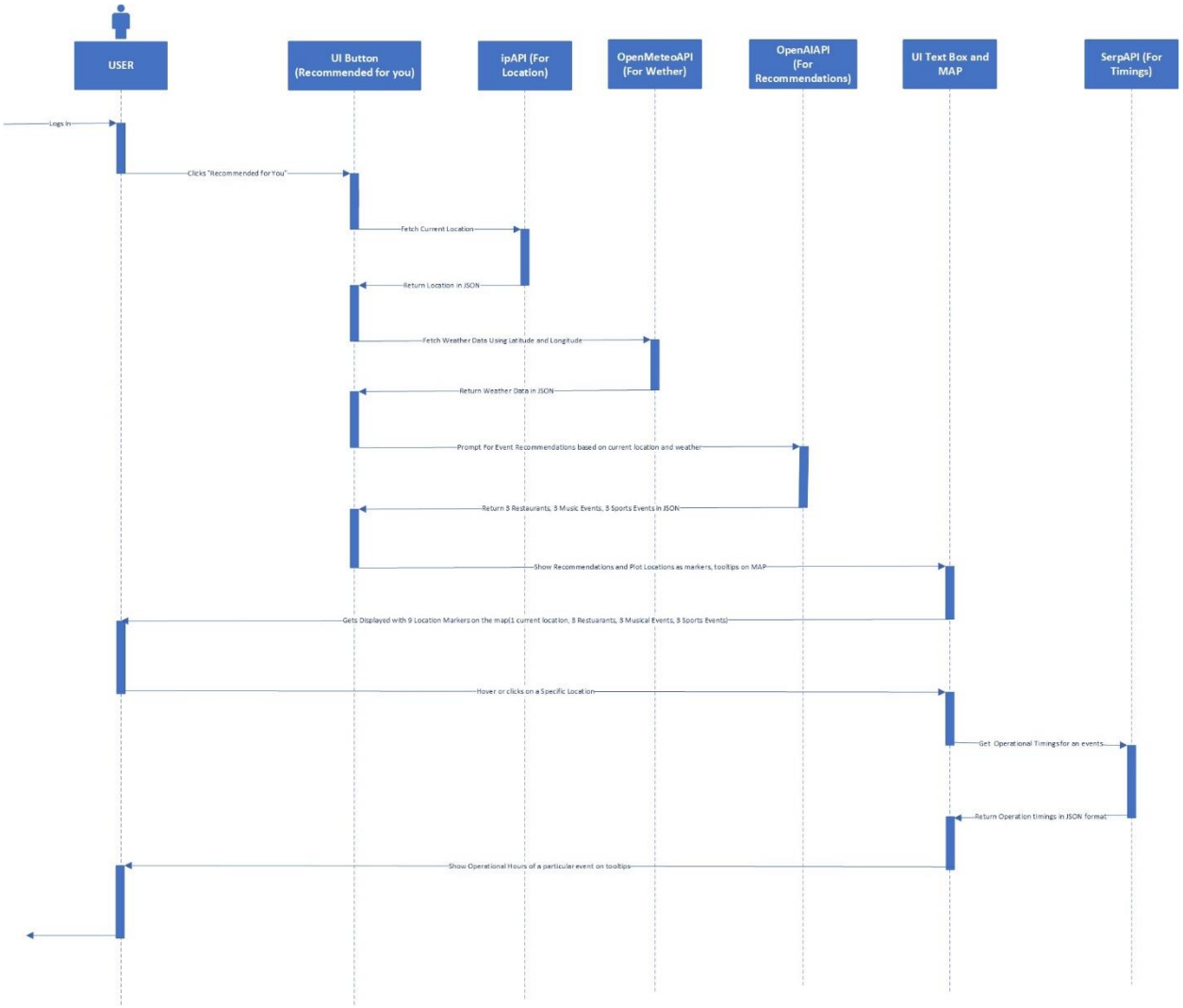
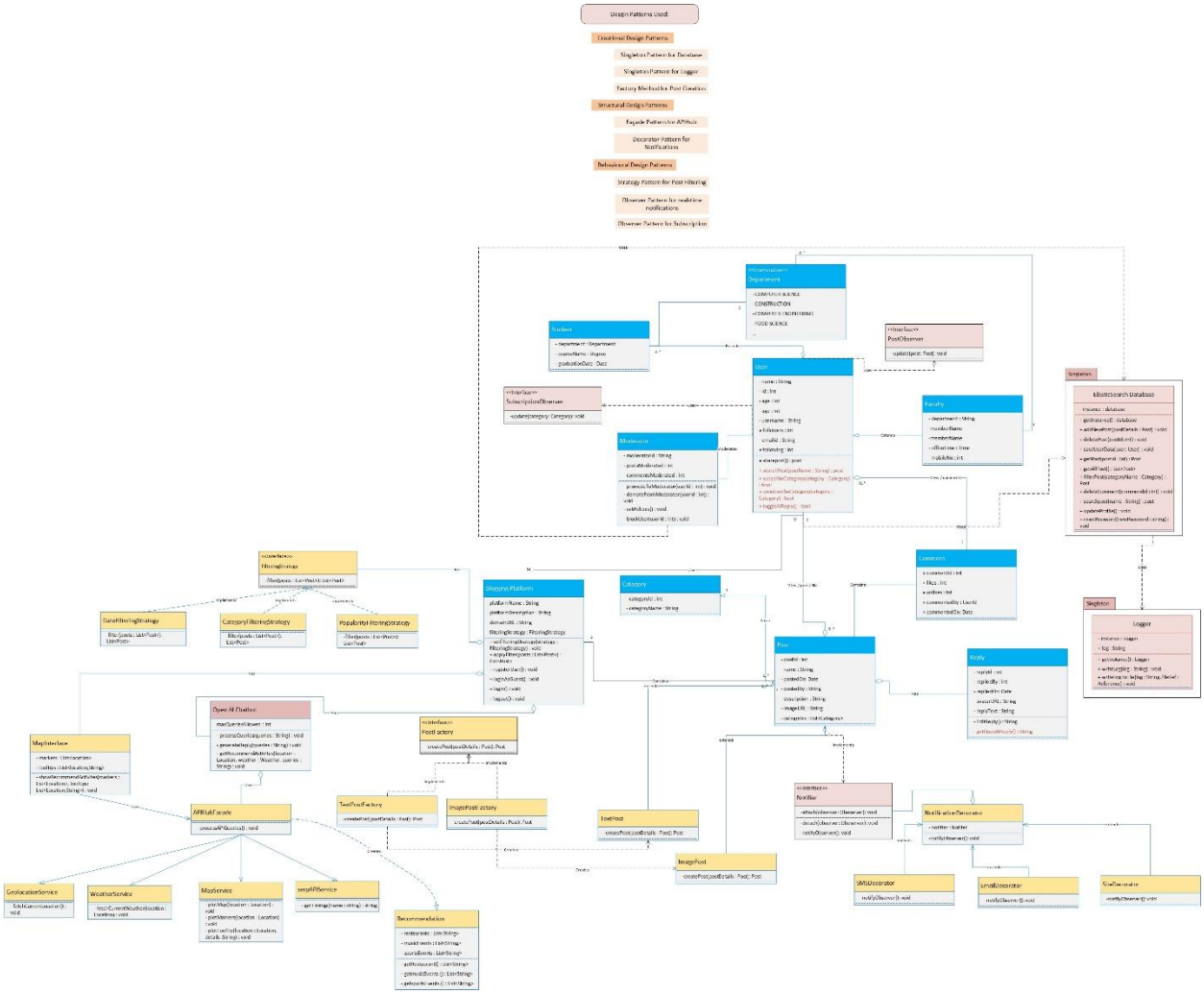
CSP 586 A20551502

Assignment No: 5

The UML interaction diagram for the feature Recommended for You:



2. The Refined Class diagram for this Platform Using Design Patterns:



Creational Design Patterns:

1. **Singleton Pattern:**

Explanation:

The Singleton pattern ensures that a class has only one instance and provides a global point of access to that instance. It is commonly used when exactly one object is needed to coordinate actions across the system, such as a database connection pool or a logger.

Usage in Class Diagram:

In class diagram, the Singleton pattern is applied to both the ElasticSearch Database and the Logger classes. Each class has a single static method getInstance() that returns the single instance of the class. This ensures that all parts of the system use the same instance of the database or logger.

Usage in Blogging Platform:

ElasticSearch Database Singleton:

In blogging platform, the ElasticSearch Database Singleton ensures that there is only one instance of the database throughout the application.

It provides methods like addNewPost, deletePost, saveUserData, etc., allowing different parts of the platform to interact with the database consistently.

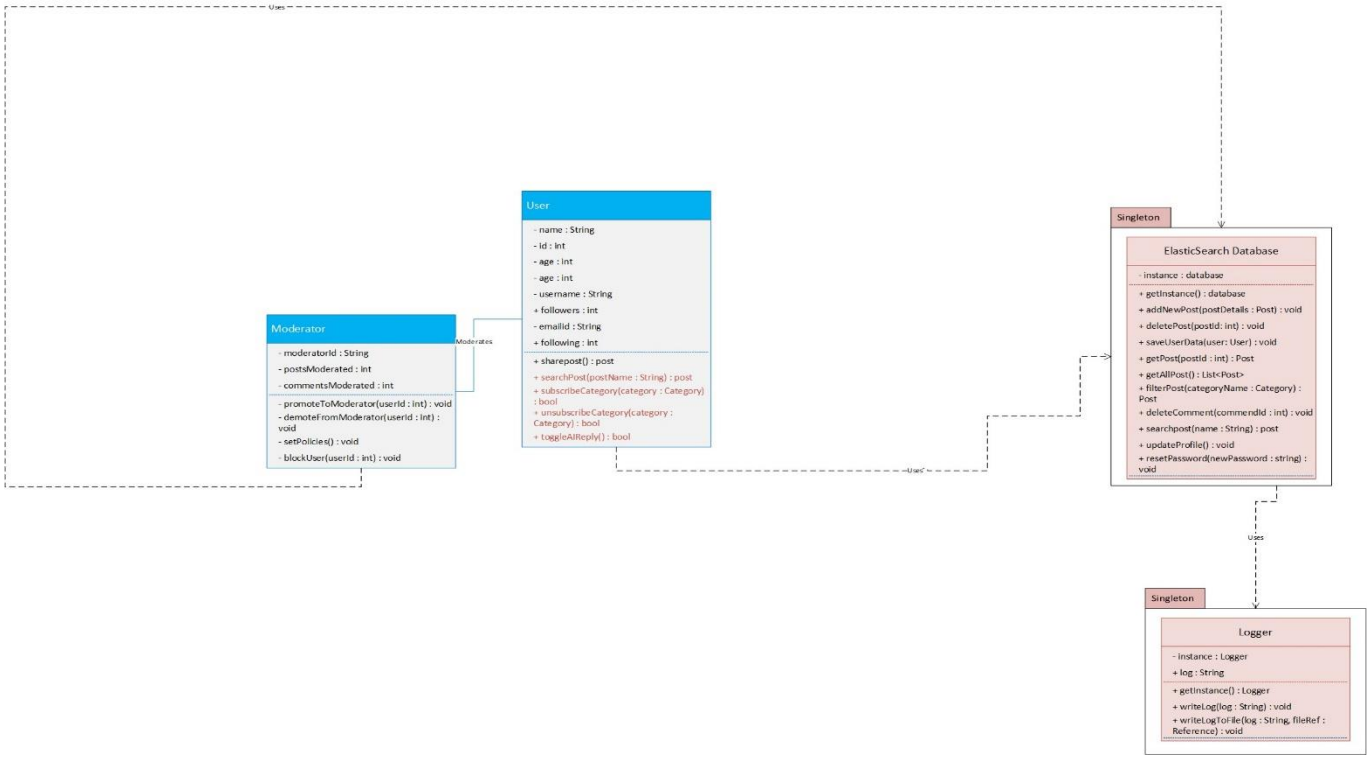
For example, when a new post is created, the addNewPost method of the Singleton database instance is called to store the post data.

Logger Singleton:

Similarly, the Logger Singleton ensures that there is only one instance of the logger, maintaining centralized logging functionality.

It provides methods like writeLog and writeLogToFile to handle logging operations.

For instance, when an error occurs during the execution of a feature in the platform, the writeLog method of the Singleton logger instance can be used to log the error message.



2. Factory Method Pattern:

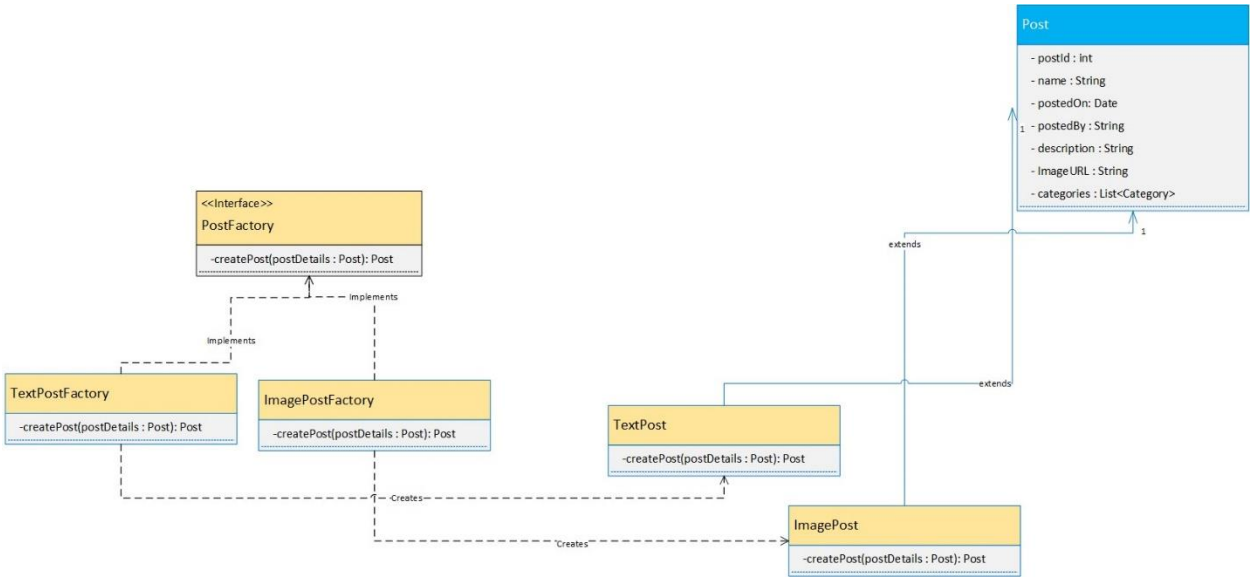
Explanation:

The Factory pattern is a creational design pattern that provides an interface for creating objects in a superclass but allows subclasses to alter the type of objects that will be created. It promotes loose coupling by abstracting the object creation process and delegating it to subclasses, thus allowing the client code to use the factory method to create objects without needing to specify their exact class.

Usage in Class Diagram:

In class diagram, the Factory pattern is applied to the creation of different types of posts (text posts and image posts). You have an interface PostFactory that defines a method createPost(postDetails: Post) for creating posts.

This interface is implemented by two concrete factory classes: TextPostFactory and ImagePostFactory. Each factory class is responsible for creating a specific type of post: text posts and image posts, respectively.



Usage in Blogging Platform:

PostFactory Interface:

The PostFactory interface defines a method createPost(postDetails: Post) for creating posts. This allows the creation of posts to be abstracted from the client code, making it easier to add new types of posts in the future without modifying existing code.

TextPostFactory:

The TextPostFactory class implements the PostFactory interface and is responsible for creating text posts.

When a text post needs to be created, the TextPostFactory class is invoked, and it returns an instance of TextPost.

ImagePostFactory:

Similarly, the ImagePostFactory class implements the PostFactory interface and is responsible for creating image posts.

When an image post needs to be created, the ImagePostFactory class is invoked, and it returns an instance of ImagePost.

TextPost and ImagePost:

Both TextPost and ImagePost classes extend the Post class and represent different types of posts supported by the platform.

Each factory class creates an instance of the corresponding post type (TextPost or ImagePost) based on the provided post details.

Structural Design Patterns:

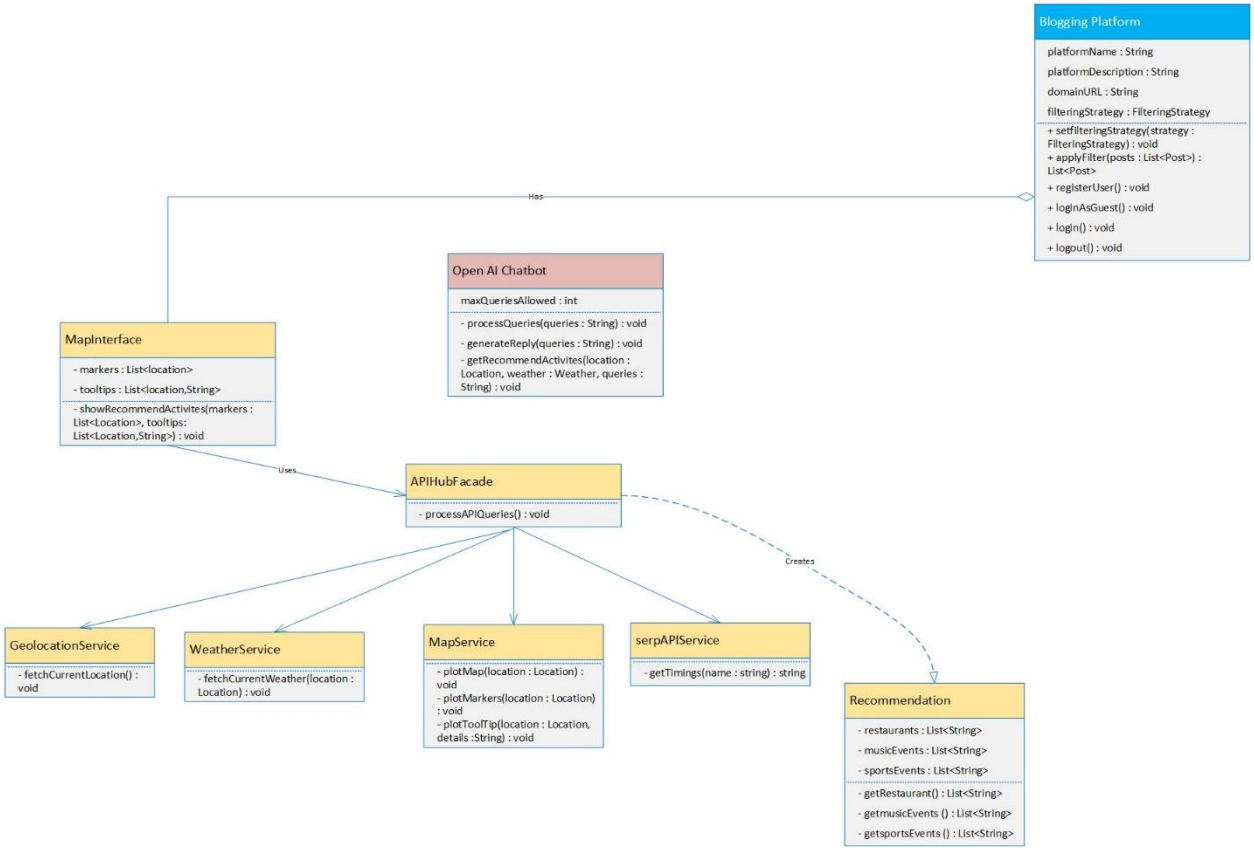
1. **Facade Pattern:**

Explanation:

The Facade pattern is a structural design pattern that provides a simplified interface to a complex subsystem. It hides the complexities of the subsystem and provides a single interface for the client to interact with.

Usage in Class Diagram:

In class diagram, the Facade pattern is implemented by the APIHubFacade class, which acts as a facade for interacting with various services related to API queries. The APIHubFacade class encapsulates the interactions with services like GeolocationService, WeatherService, MapService, and serpAPIService, providing a single method processAPIQueries() to coordinate the interactions with these services.



Usage in Blogging Platform:

The APIHubFacade class provides a simplified interface for the Open AI ChatBot to interact with various services for processing API queries.

It encapsulates the complexities of interacting with different services such as geolocation, weather, map plotting, and SERP (Search Engine Results Page) API, providing a unified method processAPIQueries() to trigger the processing of API queries.

Various services such as GeolocationService, WeatherService, MapService, and serpAPIService perform specific tasks related to API queries.

These services are used internally by the APIHubFacade class to fetch current location, weather information, plot maps, retrieve event timings, etc.

The Recommendation class provides methods for fetching recommendations such as restaurants, music events, and sports events based on the processed API queries.

2. Decorator Pattern:

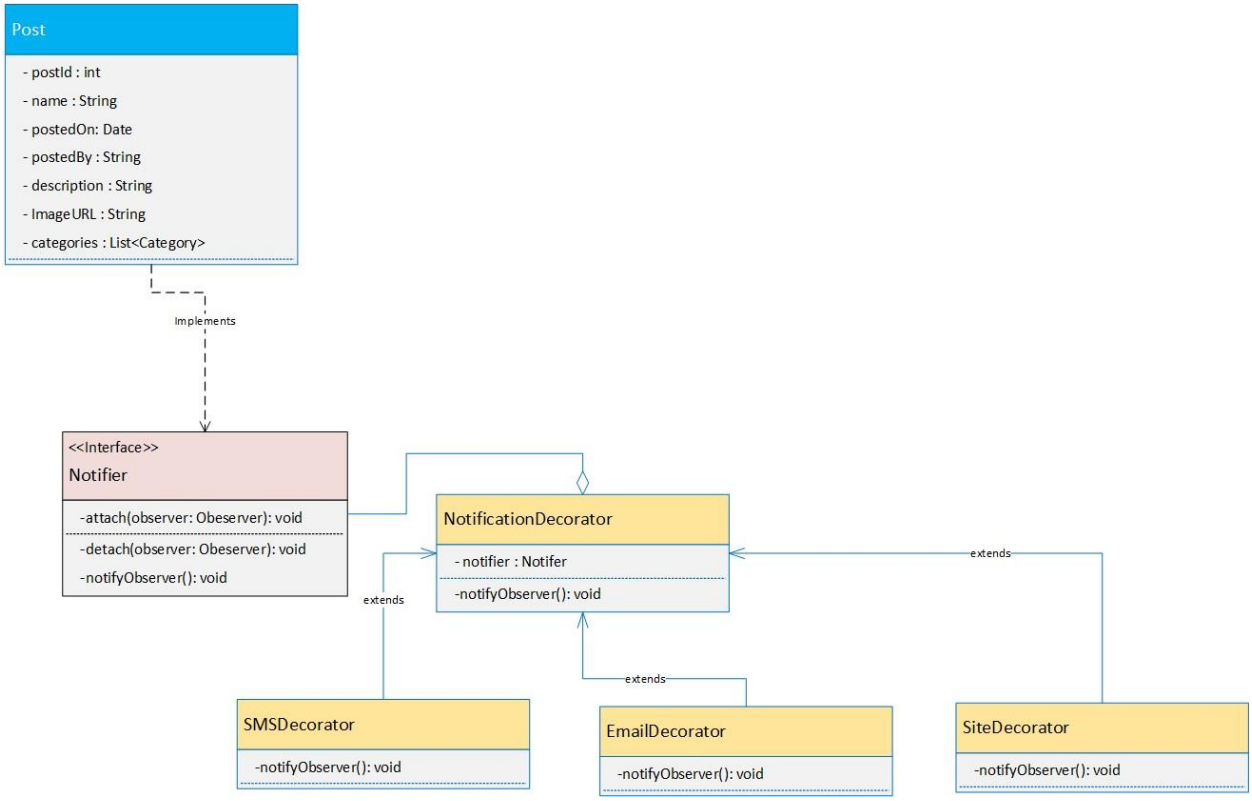
Explanation:

The Decorator pattern is a structural design pattern that allows behavior to be added to individual objects, either statically or dynamically, without affecting the behavior of other objects from the same class. It is useful for adding or modifying functionality of objects at runtime.

Usage in Class Diagram:

In class diagram, the Decorator pattern is applied to the notification services using the Notifier interface and several decorator classes (NotificationDecorator, SMSDecorator, EmailDecorator, SiteDecorator). The Notifier interface defines the basic operations for attaching, detaching, and notifying observers.

The NotificationDecorator class serves as the base decorator class, while the specific decorators (SMSDecorator, EmailDecorator, SiteDecorator) add specific notification methods.



Usage in Blogging Platform:

The Decorator pattern is used for notification services in the blogging platform. The Notifier interface defines basic operations, while decorator classes (NotificationDecorator, SMSDecorator, EmailDecorator, SiteDecorator) add specific functionality for sending notifications via different channels (SMS, email, website). This allows for flexible and dynamic notification management without modifying the core Notifier interface.

Behavioral Design Patterns:

1. **Strategy Pattern:**

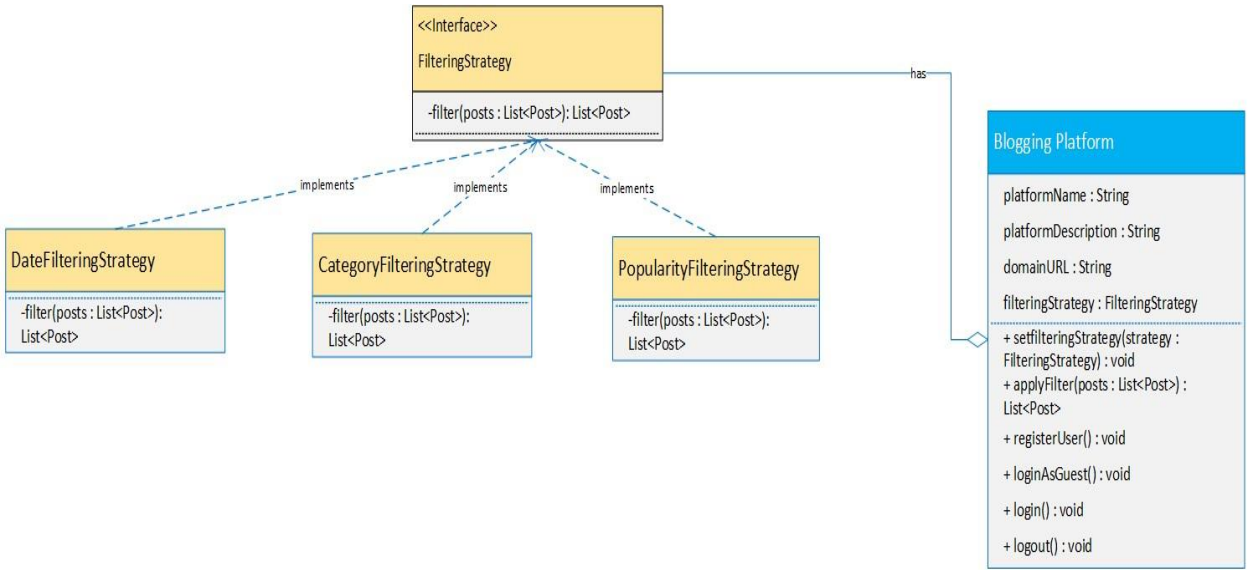
Explanation:

The Strategy pattern is a behavioral design pattern that allows a family of algorithms to be defined, encapsulated, and made interchangeable at runtime. It enables the client to choose from a variety of algorithms or strategies to accomplish a specific task.

Usage in Class Diagram:

In class diagram, the Strategy pattern is applied to the filtering functionality in the BloggingPlatform class. The BloggingPlatform class has a method setFilteringStrategy() to set the filtering strategy dynamically and a method applyFilter() to apply the chosen strategy to a list of posts.

The FilteringStrategy interface defines the contract for all filtering strategies, and concrete strategies (DateFilteringStrategy, CategoryFilteringStrategy, PopularityFilteringStrategy) implement this interface and provide specific filtering algorithms.



Usage in Blogging Platform:

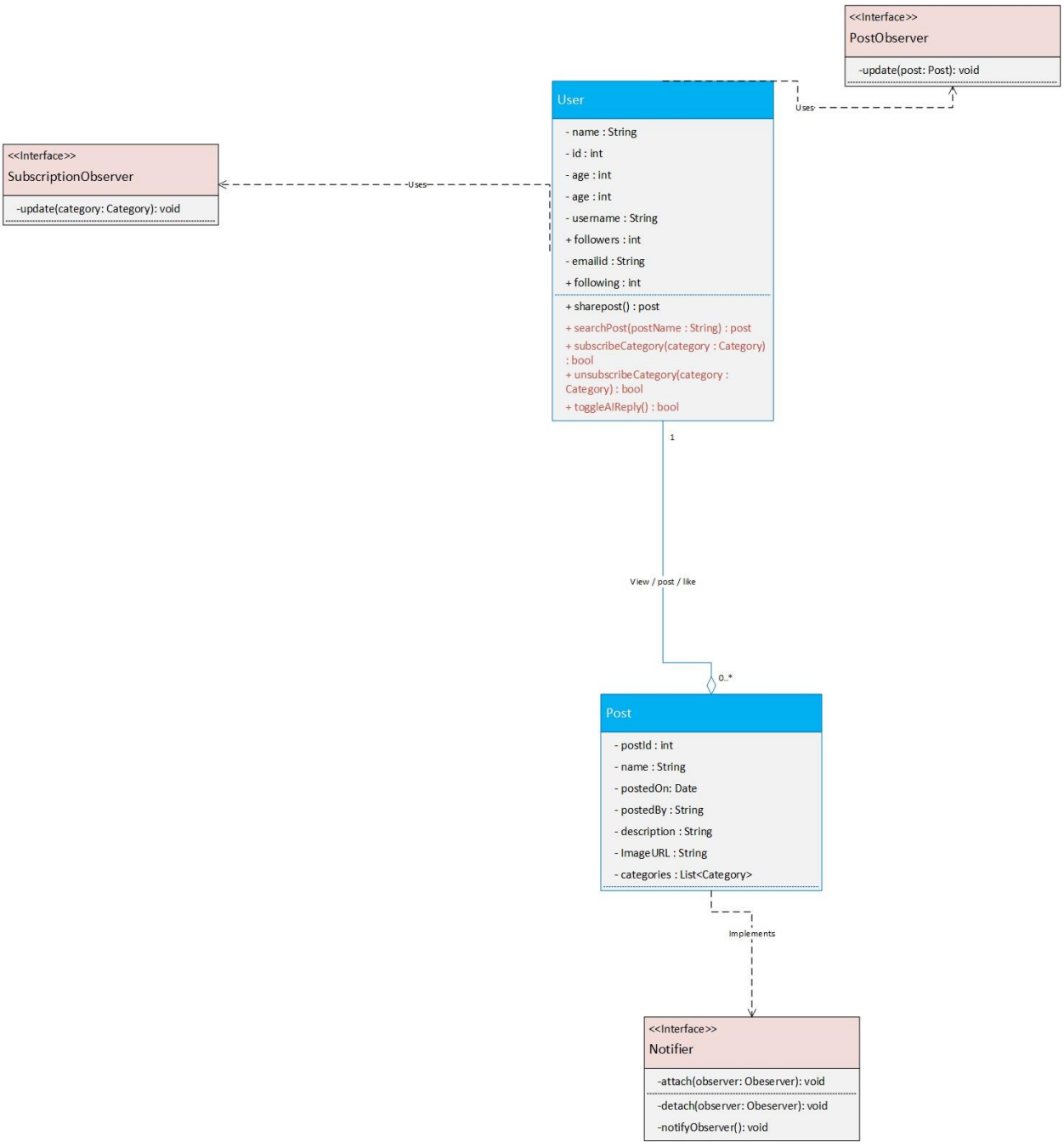
The Strategy pattern is used in the BloggingPlatform to provide flexibility in filtering posts. The BloggingPlatform class allows the client to choose a filtering strategy dynamically.

Concrete strategy classes (DateFilteringStrategy, CategoryFilteringStrategy, PopularityFilteringStrategy) implement specific filtering algorithms, which can be swapped at runtime to apply different filtering criteria to the list of posts.

2. Observer Pattern:

Explanation:

The Observer pattern is a behavioral design pattern where an object, known as the subject, maintains a list of its dependents, called observers, and notifies them of any state changes, usually by calling one of their



methods. This pattern promotes loose coupling between objects, as the subject does not need to know the specific details of its observers.

Usage in Class Diagram:

In class diagram, the Observer pattern is used to implement the notification system for new postings in the blogging platform. The Subject interface represents the subject (or publisher) that maintains a list of observers and notifies them of any changes.

The Observer interface defines the contract for all observers, and concrete observer classes (User, Moderator, etc.) implement this interface and subscribe to receive notifications.

Usage in Blogging Platform:

The Observer pattern is used in the blogging platform to notify observers (users, moderators, etc.) about new postings. The PostPublisher acts as the subject, maintaining a list of observers and notifying them when new posts are published. Observers receive notifications and update their state accordingly, promoting loose coupling between the subject and observers.