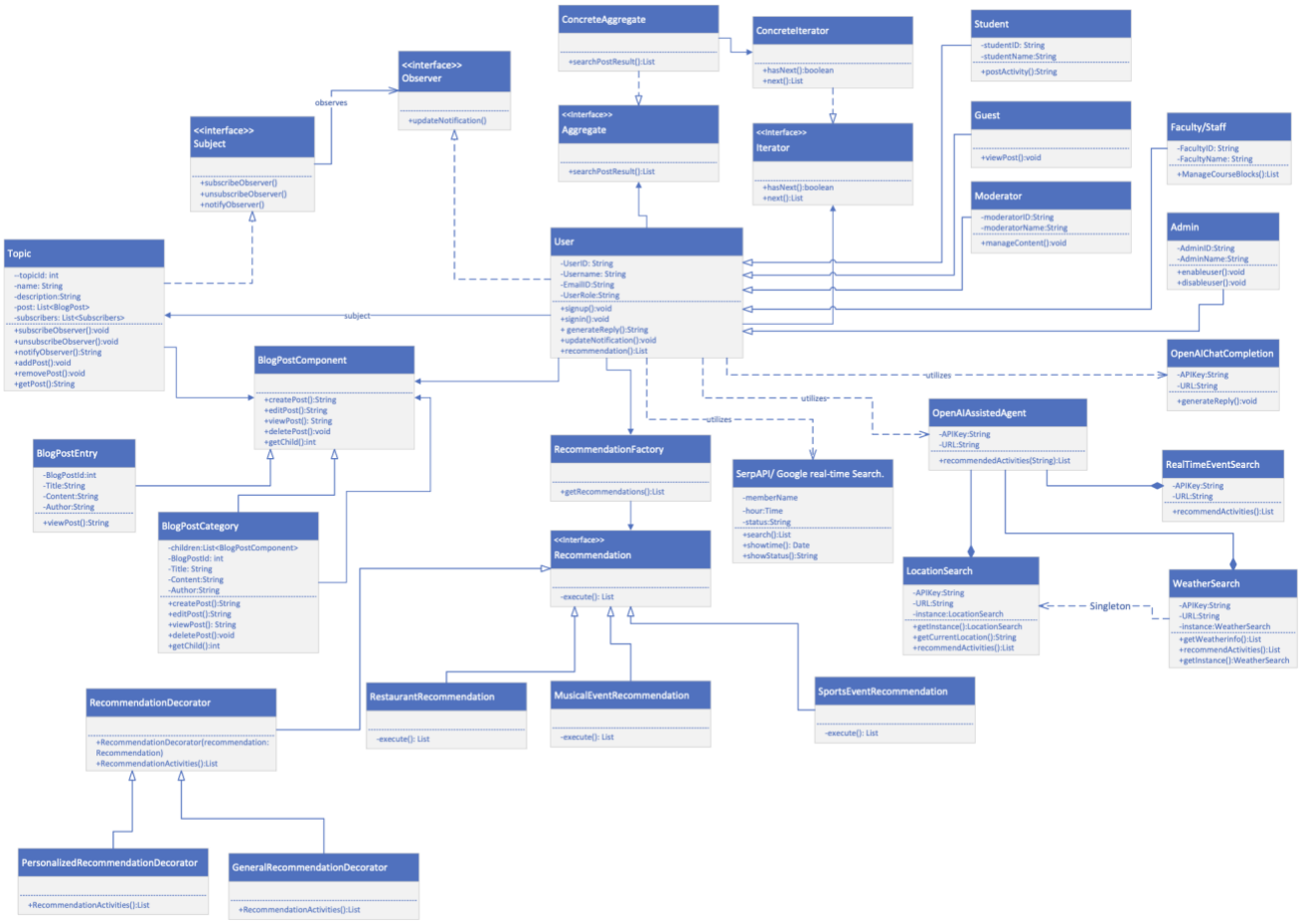
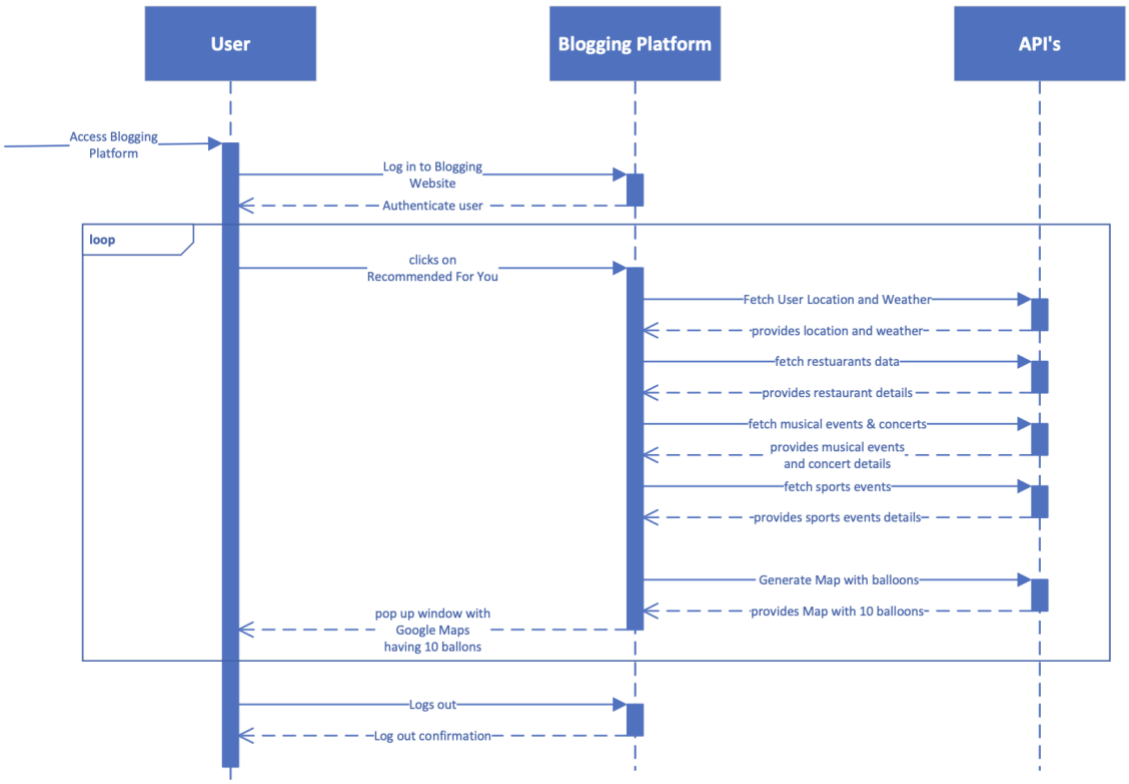
Assignment – 5

Vaishnavi Prasanna Shetty
A20519894
vprasannashetty@hawk.iit.edu

- **Class Diagram**

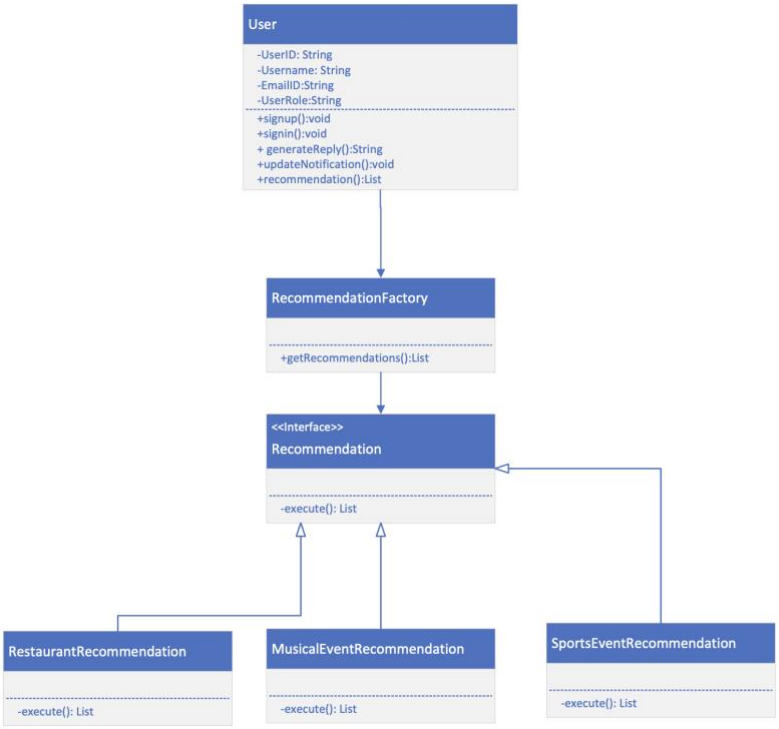


- **<u>UML Interaction Diagram for the feature ‘Recommended for you’</u>**



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

I. Factory Method



The diagram provided represents a system where different types of recommendations are generated for a user. We have used the Factory Method design pattern for this, which falls under the category of Creational Design Patterns. In the Factory Method pattern defines the interface for the creation of objects but subclasses decide on which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

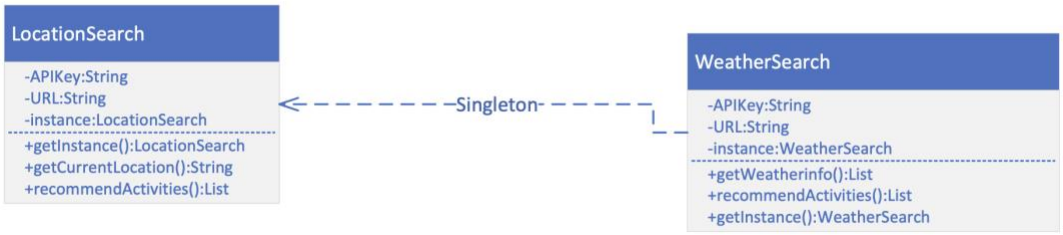
In the context of the diagram, the RecommendationFactory class would act as the creator, and it has a method getRecommendations() which returns a list of recommendations. It doesn't specify what type of recommendations it will create; that detail is abstracted away.

The Recommendation interface serves as the 'Product' in the Factory Method pattern. It declares the execute() method, which is expected to be implemented by any concrete class that implements this interface.

Concrete classes like RestaurantRecommendation, MusicalEventRecommendation, and SportsEventRecommendation implement the Recommendation interface, specifically the execute() method, to provide the specifics of how those recommendations are processed and generated. Each of these classes represents a different 'Product' that the factory method can create.

User is the client in the context of the Factory Method pattern, which requires a product namely recommendation.

II. Singleton Design Pattern



The Singleton Pattern ensures a class has only one instance and provides a global point of access to it. In this case a class allows only a single instance of itself to be created and provides a way to access that instance. In the context of the provided UML diagram, both LocationSearch and WeatherSearch are represented as Singletons.

Purpose: The Singleton pattern is used to ensure that only one instance of LocationSearch and WeatherSearch can exist at any one time.

Instance Variable: Both classes have a private static variable (-instance) that holds the sole instance of the class. This variable is managed internally and is not directly accessible from outside the class.

GetInstance Method: Both classes provide a +getInstance() public method, which returns the unique instance of the class. If the instance does not already exist, the method creates it. This is the global access point to the instance.

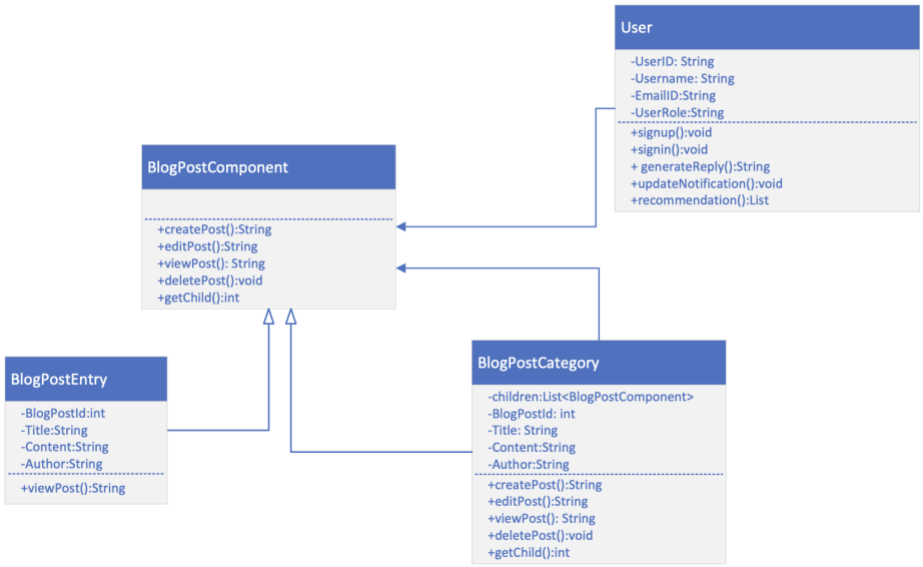
Private Constructor: Though not shown in UML, it is understood that the constructor for a Singleton pattern should be private to prevent other classes from creating new instances directly.

Usage: LocationSearch.getInstance() is used to obtain the current location and recommend activities based on that location. WeatherSearch.getInstance() is used to get weather information and recommend activities based on the weather. It would need to use LocationSearch's instance for location-dependent weather data.

Collaboration: The pattern suggests collaboration between LocationSearch and WeatherSearch. WeatherSearch uses LocationSearch.getInstance().getCurrentLocation() to tailor the weather information it provides.

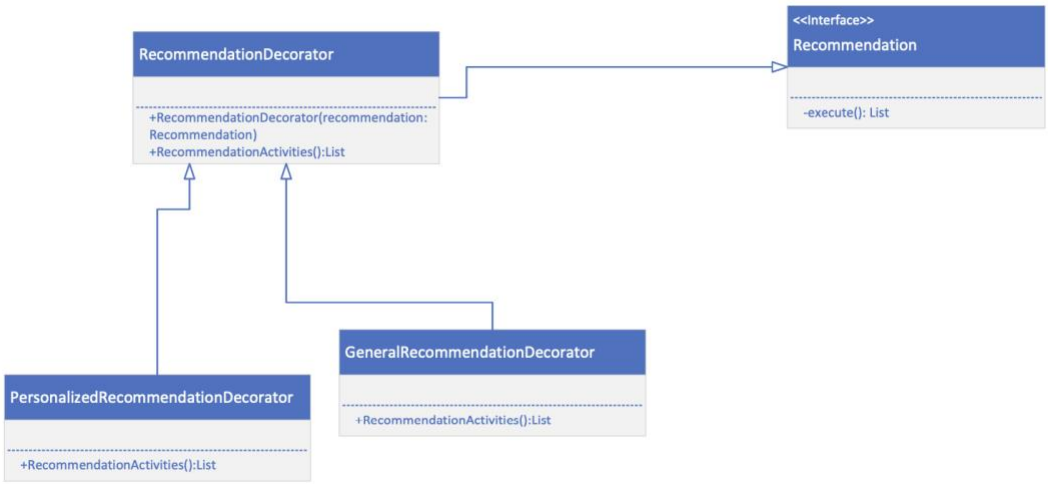
6.Document and Annotate 2 Structural Design Patterns that are used in your UML Design Class Diagram.

i. Composite Design Pattern



The Composite Design Pattern allows us to treat individual objects and compositions of objects uniformly by organizing them into tree structures that represent part-whole hierarchies.
The User (Client) will utilize the methods of the BlogPostComponent to create, view, edit, or delete blog posts or categories without needing to know whether they are interacting with a BlogPostEntry or a BlogPostCategory.
Component defines an interface for all the objects in the composition for both composite and leaf nodes. In our case, BlogPostComponent would defines common operations for both simple and composite objects in the hierarchy. This would be implementing the default behaviors for createPost(), editPost(), deletePost(), and getChild().
Leaf (BlogPostEntry): These are the basic elements of the structure that do not have children. They represent the actual content, such as individual blog posts, in the composite structure.
Composite (BlogPostCategory): A component that has children. It can hold other leaves or composites, enabling the creation of complex structures. The children attribute signifies that this class can contain multiple BlogPostComponent elements.

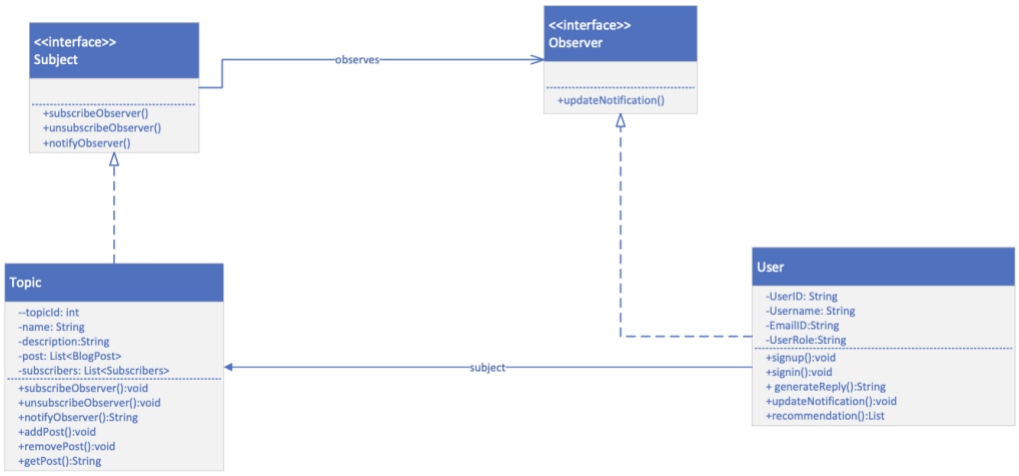
ii. Decorator Design Pattern



In the decorator design pattern we have,
Component Interface (Recommendation):This interface defines the structure for types of recommendations. It includes operations that all concrete implementations and decorators will share, such as obtaining a list of recommendations.
Concrete Component: This would be a class that implements the Recommendation interface to provide the core functionality.
Abstract Decorator (RecommendationDecorator): This abstract class implements the Recommendation interface and holds a reference to a Recommendation object. It's designed to delegate the recommendation request to the object it decorates, which allows for additional behaviors to be added.
The recommendationDecoratorRecommendation attribute points to the wrapped component.
The recommendationActivities List is a collection of activities that the decorator uses to add new functionality or information to the existing Recommendation. Concrete Decorators (GeneralRecommendationDecorator and PersonalizedRecommendationDecorator):
These classes extend RecommendationDecorator and represent specific "decorations" to original Recommendation objects, which overrides methods from the RecommendationDecorator to provide additional behavior.
The recommendationActivities List in each concrete decorator represents the unique additions or modifications they introduce to the recommendation process.
In the context of the decorator pattern, the class diagram specifies how the RecommendationDecorator serves as a foundation for a specialized decorator (GeneralRecommendationDecorator and PersonalizedRecommendationDecorator). Each concrete decorator class has the potential to extend the behavior of Recommendation objects in a dynamic and flexible way without requiring changes to the existing code of the Recommendation interface or concrete components that may exist.

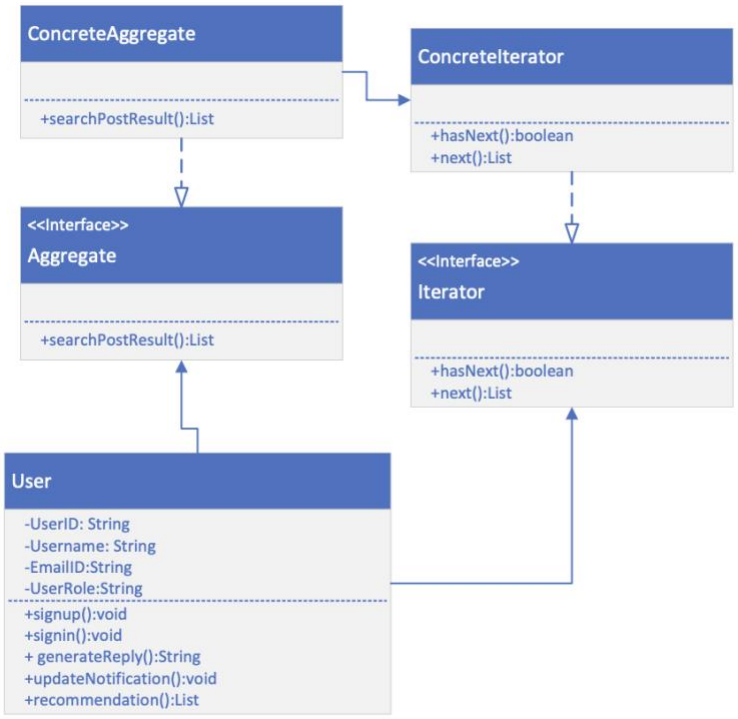
7. <u>Document and Annotate 2 Behavioral Design Patterns that are used in your UML Design Class Diagram.</u>

a. <u>Observer Design Pattern</u>



The Observer Pattern defines a one-to-many dependency between objects so that when one object changes state, all of its dependents are notified and updated automatically.
In this observer design pattern we have,
Subject Interface: Methods for attaching (subscribeObserver) and detaching (unsubscribeObserver) observers and Notifies (notifyObservers) all observers of state changes.
Topic Class: A concrete implementation of the Subject which manages a list of posts and observers. It can add (addPost) and remove (removePost) posts, as well as retrieve a post (getPost).
Observer Interface: Contains the updateNotification() method to receive updates from the subject.
User Class: Implements the Observer interface.
Relationships: The dashed line from Topic to Observer indicates that Topic will notify User objects of changes, but without being directly coupled to them. The solid line from Topic to Subject indicates that Topic is a type of Subject. This pattern is advantageous for designing flexible systems where subjects and observers can vary independently.

b. Iterator Design Pattern



Iterator Design Pattern allows the traversal of the elements of an aggregate without exposing the underlying implementation.
Aggregate (<<Interface>> Aggregate): This interface represents the list. It includes the method searchPostResult(), which returns an iterator for traversing the collection.
ConcreteAggregate is a concrete implementation of the Aggregate interface. It implements the searchPostResult() method and provides the means to create a ConcreteIterator for its collection.
Iterator (<<Interface>> Iterator): This interface provides the standard for iteration, which includes two essential methods namely hasNext() which Checks if there are more elements to iterate over. next() Moves the iterator to the next element and returns it.
ConcreteIterator (ConcreteIterator): It is the implementation of the Iterator interface. This class is responsible for the iteration logic and maintaining the current position within the collection.
User acts as the client in this pattern. It utilizes the iterator to search through posts. When the User class invokes the searchPostResult() method, it receives an iterator from the ConcreteAggregate. This allows the User to traverse through search results using hasNext() and next() methods without knowing the underlying data structure of the post collection.