**Creational Design Patterns:**

Factory method:

Annotations:

* interface UserFactory: Represents an interface for creating users.
* class UserFactoryImpl: Implements UserFactory and provides methods to create different types of users.
* class User: Represents a basic user with methods to create, reply to, view posts, and get recommendations.
* class Moderator: Extends User and adds a method to delete posts.
* class Administrator: Extends User and adds a method to enable/disable login for other users.

Singleton pattern:

Annotations:

* class RecommendationService: Represents a service for providing recommendations.
* -instance: RecommendationService: Private field for holding the single instance of RecommendationService.
* -userLocation: String: Field to store the user's location.
* -weatherConditions: String: Field to store weather conditions.
* -getRecommendations(): List<Recommendation>: Method to get recommendations.
* +getInstance(): RecommendationService: Static method to get the single instance of RecommendationService.
* -RecommendationService(): Private constructor to prevent direct instantiation of RecommendationService.
* RecommendationService ..> OpenAIService: Indicates a relationship where RecommendationService uses OpenAIService.
* RecommendationService ..> WeatherService: Indicates a relationship where RecommendationService uses WeatherService.
* RecommendationService ..> LocationService: Indicates a relationship where RecommendationService uses LocationService.
* RecommendationService ..> SearchService: Indicates a relationship where RecommendationService uses SearchService.
* RecommendationService ..> SerpAPIService: Indicates a relationship where RecommendationService uses SerpAPIService.

**Structural Design patterns:**

composite pattern

Annotations:

* class Topic: Represents a topic with properties like name, description, keywords, and a list of posts. It has methods to add and remove posts.
* class Post: Represents a post with properties like ID, title, content, timestamp, comments, and author.
* interface Component: Represents the Component interface, defining the operation method.
* class Composite: Implements the Component interface and represents a composite component that can have children components. It has methods to add, remove, and perform operations on children components.
* class Leaf: Implements the Component interface and represents a leaf component that doesn't have children. It has a method to perform operations.
* Topic o-- Post : contains: Indicates that a Topic contains multiple Post instances.
* Composite o-- Component : children: Indicates that a Composite component can have children components.
* Component <|-- Composite: Indicates that Composite is a subtype of Component.
* Component <|-- Leaf: Indicates that Leaf is a subtype of Component.

Decorator pattern:

Annotations:

* interface User: Represents the base functionality that users can have, such as creating posts, replying to posts, viewing posts, and getting recommendations.
* class BaseUser: Implements the User interface and represents a basic user with properties like username, password, and email. It also implements the methods defined in the User interface.
* abstract class UserDecorator: An abstract class that extends User and serves as the base class for user decorators. It contains a reference to the decorated user (-user: User) and implements the methods defined in the User interface.
* class ModeratorDecorator: Extends UserDecorator and adds the functionality to delete posts (+deletePost(post: Post): void).
* class AdministratorDecorator: Extends UserDecorator and adds the functionality to enable/disable login for other users (+enableDisableLogin(user: User): void).
* BaseUser ..|> User: Indicates that BaseUser implements User.
* UserDecorator ..|> User: Indicates that UserDecorator extends User.
* ModeratorDecorator --|> UserDecorator: Indicates that ModeratorDecorator extends UserDecorator.
* AdministratorDecorator --|> UserDecorator: Indicates that AdministratorDecorator extends UserDecorator.

**Behavioral Design Patterns:**

Strategy pattern:

Annotations:

* interface RecommendationStrategy: Represents a strategy for generating recommendations based on user location and weather conditions.
* +generateRecommendations(userLocation: String, weatherConditions: String): List<Recommendation>: Method to generate recommendations based on user location and weather conditions.
* class OpenAIRecommendationStrategy: Implements RecommendationStrategy and uses various services (openAIService, weatherService, locationService, searchService, serpAPIService) to generate recommendations.
* -openAIService: OpenAIService: Private field representing the OpenAI service used by the strategy.
* -weatherService: WeatherService: Private field representing the weather service used by the strategy.
* -locationService: LocationService: Private field representing the location service used by the strategy.
* -searchService: SearchService: Private field representing the search service used by the strategy.
* -serpAPIService: SerpAPIService: Private field representing the SERP API service used by the strategy.
* +generateRecommendations(userLocation: String, weatherConditions: String): List<Recommendation>: Method to generate recommendations based on user location and weather conditions.
* class RecommendationService: Represents a service that uses a recommendation strategy to provide recommendations.
* -recommendationStrategy: RecommendationStrategy: Private field representing the current recommendation strategy used by the service.
* +setRecommendationStrategy(strategy: RecommendationStrategy): void: Method to set the recommendation strategy.
* +getRecommendations(userLocation: String, weatherConditions: String): List<Recommendation>: Method to get recommendations based on user location and weather conditions using the current strategy.
* RecommendationService ..> RecommendationStrategy : uses: Indicates that RecommendationService uses RecommendationStrategy for generating recommendations.
* OpenAIRecommendationStrategy ..> OpenAIService : uses: Indicates that OpenAIRecommendationStrategy uses OpenAIService.
* OpenAIRecommendationStrategy ..> WeatherService : uses: Indicates that OpenAIRecommendationStrategy uses WeatherService.
* OpenAIRecommendationStrategy ..> LocationService : uses: Indicates that OpenAIRecommendationStrategy uses LocationService.
* OpenAIRecommendationStrategy ..> SearchService : uses: Indicates that OpenAIRecommendationStrategy uses SearchService.
* OpenAIRecommendationStrategy ..> SerpAPIService : uses: Indicates that OpenAIRecommendationStrategy uses SerpAPIService.

Observer Pattern:

Annotations:

* interface Observer: Represents the observer in the Observer design pattern, with a method update(subject: Subject) to update its state when notified by a subject.
* interface Subject: Represents the subject in the Observer design pattern, with methods attach(observer: Observer), detach(observer: Observer), and notify() to manage observers and notify them of changes.
* class User: Implements Observer and represents a user who can subscribe to topics, posts, and the recommendation service.
* -username: String: Private field representing the user's username.
* +update(subject: Subject): void: Method to update the user's state when notified by a subject.
* class Topic: Implements Subject and represents a topic that users can subscribe to.
* -posts: List<Post>: Private field representing the posts related to the topic.
* +attach(observer: Observer): void: Method to attach an observer (user) to the topic.
* +detach(observer: Observer): void: Method to detach an observer (user) from the topic.
* +notify(): void: Method to notify all subscribed users about changes in the topic.
* class Post: Implements Subject and represents a post that users can subscribe to.
* -comments: List<Comment>: Private field representing the comments on the post.
* +attach(observer: Observer): void: Method to attach an observer (user) to the post.
* +detach(observer: Observer): void: Method to detach an observer (user) from the post.
* +notify(): void: Method to notify all subscribed users about changes in the post.
* class RecommendationService: Implements Subject and represents a service that users can subscribe to for receiving recommendations.
* +attach(observer: Observer): void: Method to attach an observer (user) to the recommendation service.
* +detach(observer: Observer): void: Method to detach an observer (user) from the recommendation service.
* +notify(): void: Method to notify all subscribed users about changes in the recommendations.