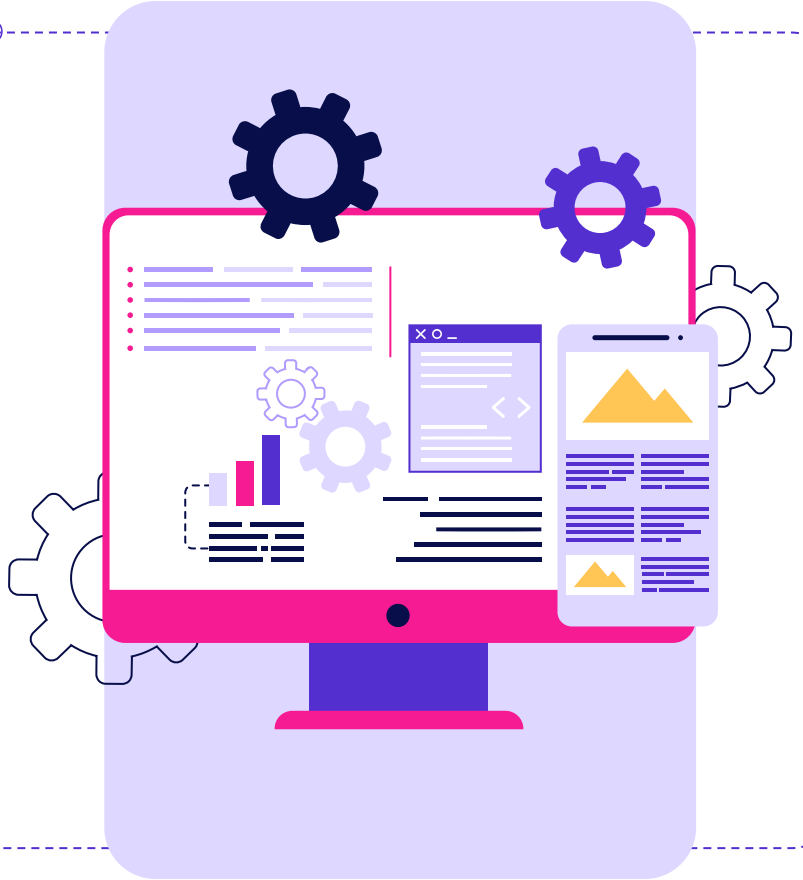
Project 2
Team 23



Team

Members

- Karan Bhatt
- Vedashree Ranade
- Madhusree Bera
- Yash Maheshwari
- Piyush Rana

Mentor

- Venu

Team Contribution

Name	Contribution
Madhusree Bera (2022202007)	• Code Analysis and Design pattern recognition • Feature 1 (Better user management) • Bonus Feature (Private and Public bookshelves) • Reports
Karan Bhatt (2022202003)	• Code Analysis and Design pattern recognition • Feature 2 (Common Library) • Reports
Vedashree Ranade (2022201073)	• Code Analysis and Design pattern recognition • Feature 2 (Common Library) • Reports
	• Code Analysis and Design pattern recognition • Feature 3 (Online Integration) • Reports
Piyush Rana (2022202012)	• Code Analysis and Design pattern recognition • Feature 3 (Online Integration) • Reports

Feature 1

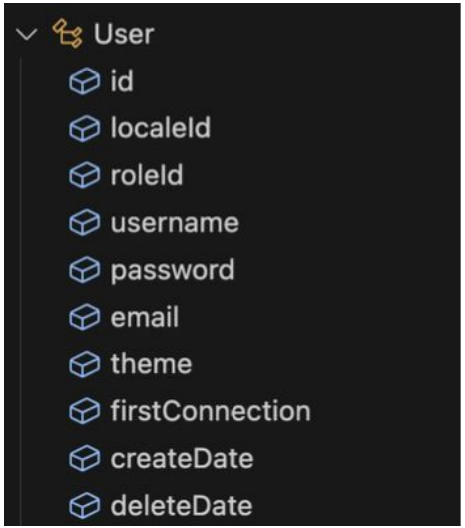
Better user management

Pattern Analysis

01 Builder Pattern - Creation of Users

Rationale:

- Creating a new user **does not require all the constructor parameters**
- Requires only username, email and password
- We can **avoid** instantiation of a **huge constructor**
- Allows addition of **optional fields (like Address)** without constructor clutter or multiple constructors



User Class

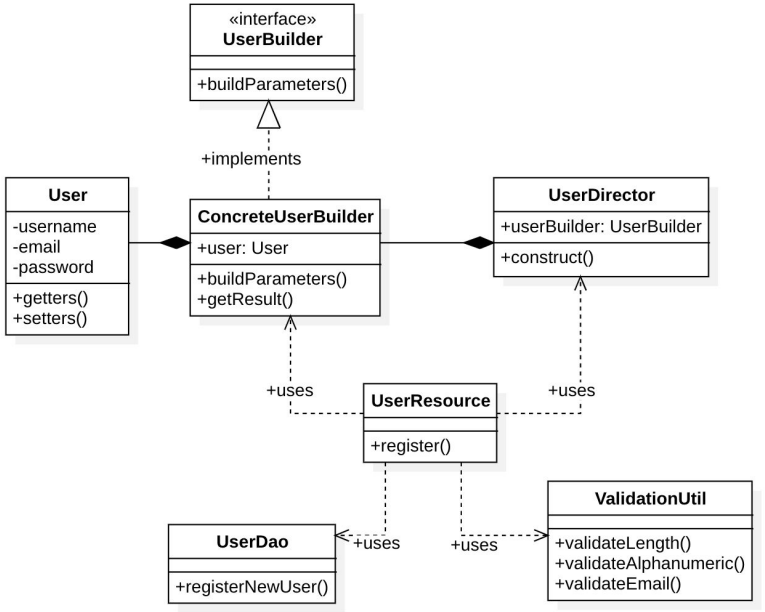
Pattern Analysis

02 Why Chain of Responsibility is not applicable

Rationale:

- In CoR, one single request requires processing by **multiple objects**
- Chain the receiving objects and pass the request along the chain **until one handles it**
- In our use case, if we create separate Class for every method for validating length, email and alphanumeric characters => **results in Imperative Abstraction smell**

Feature 1 Implementation



Class Diagram

Feature 1: Extensibility

- **new form parameters** can be easily introduced which may or may not be optional
- The **Director** will **orchestrate the construction process** when a new user with a separate set of attributes needs to be created.
    - For example: Address class can be easily added using Builder pattern
- Additional validation checks can be easily added by **adding methods** in ValidationUtil class

Feature 1: Modularity

- **Separating construction** of complex object from **representation**
  - parameters are being populated in the User object using the Builder class methods
  - Director class will construct the object using the object of Builder class
- Interaction with the **database** is done in UserDao class
- **API endpoint** for registration is present in UserResource class

User Class

Feature 1: Maintainability

- The construction logic for complex objects is **encapsulated** within the builder, **abstracting** it away from the calling method.
- provide a fluent interface or a **step-by-step construction process**, making code more **readable** and **maintainable**
    - Director class will construct the object using the object of Builder class
- Using Builder pattern **reduces Constructor overloading** (which is error prone and hard to maintain)

User Class

Feature 2

Common Library

Design Patterns Used

01 Singleton Pattern - Common Library

Need for a Single Instance: A single instance of the library ensures consistent data and access throughout the application.

Interaction with Modules: Both contribution and exploration modules interact with the library's data (books), necessitating a central point of access.

Centralized Control: The singleton pattern provides centralized control for library initialization and avoids code duplication for creating multiple instances.

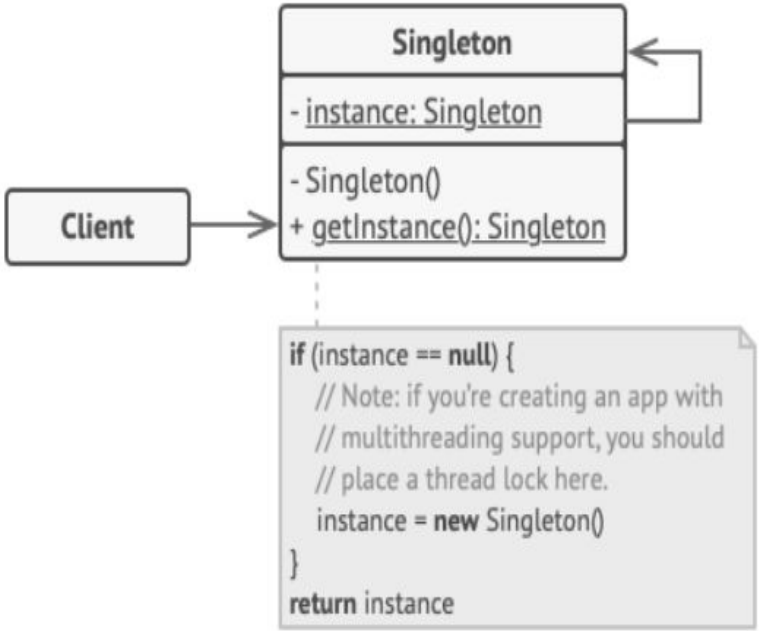
Benefits of Singleton Pattern: Using the singleton pattern for the common library offers a centralized, efficient, and maintainable approach to manage library resources.

Design Patterns Used

01 Singleton Pattern - Common Library

Relevant Code Files:

- LibraryResource.java
- Library.java



02 Builder Pattern - Creation of Books

Need for Builder Pattern: Facilitates step-by-step construction of Book objects, managing optional fields like multiple authors, genres, and rating.

Step-by-Step Construction: Enables clear, readable creation of Book objects, enhancing code maintainability.

Graceful Handling of Optional Fields: Allows addition of optional fields without constructor clutter or multiple constructors.

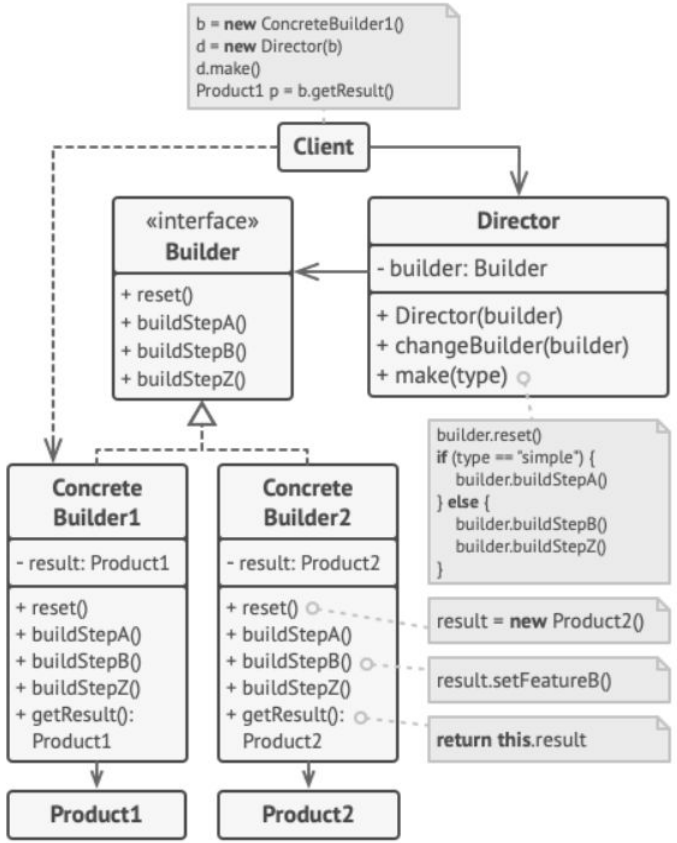
Code Cleanliness: Keeps code organized, especially for complex Book objects.

Benefits of Builder Pattern: Improves codebase extensibility, modularity, and maintainability, accommodating new attributes or changes easily.

02 Builder Pattern - Creation of Books

Relevant Code Files:

- Book.java
- BookBuilder.java
- BookDirector.java



03 Strategy Pattern - Book Ranking

Dynamic Ranking Feature: The strategy pattern is ideal for implementing the dynamic "BookRanking" feature in our library, allowing users to choose between ranking books by average rating or number of ratings.

Separation of Concerns: The strategy pattern enables the separation of ranking logic from the user interface.

Encapsulated Logic: We can define separate strategy classes for each ranking criterion (average rating, number of ratings), encapsulating the specific ranking logic within each class.

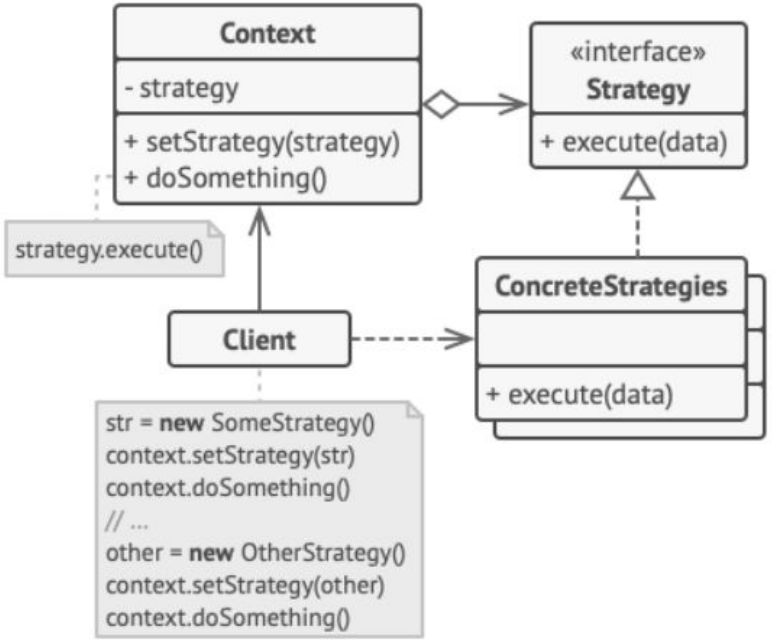
Flexible User Experience: When a user selects a criterion, the library can dynamically switch between these strategies to determine the top 10 books, providing a flexible and user-driven ranking experience.

Benefits of Strategy Pattern: Improves codebase extensibility, modularity, and maintainability, accommodating new changes easily.

03 Strategy Pattern - Book Ranking

Relevant Code Files:

- SortingStrategy.java
- AverageRatingSortingStrategy.java
- NumberOfRatingsSortingStrategy.java
- Top10List.java
- LibraryResource.java



04 Criteria / Filter Pattern - Filtering Books

User-Driven Filtering: The criteria pattern is a powerful approach for implementing user-driven book filtering in our library, allowing users to filter books based on authors, genres, and ratings.

Encapsulated Filtering Conditions: The criteria pattern enables the encapsulation of each filtering condition into separate criteria objects.

Combining Criteria: When a user selects filters, the library can combine these criteria objects to determine the books that match all specified conditions.

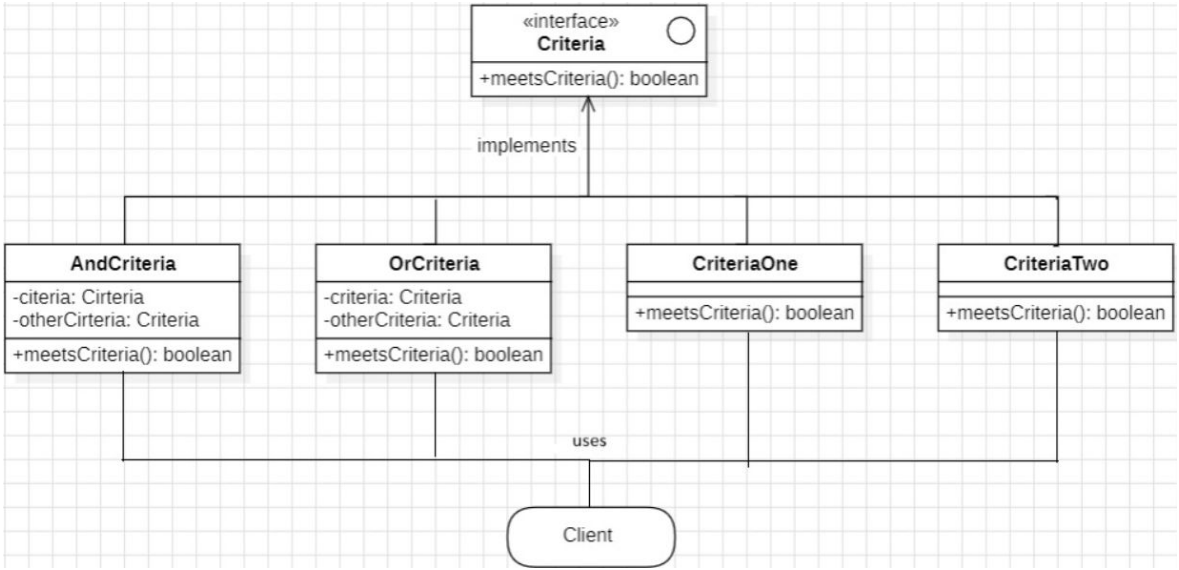
Flexible Filtering Experience: This approach offers a flexible and user-friendly filtering experience, enhancing the overall usability of the library.

Benefits of Criteria Pattern: Improves codebase extensibility, modularity, and maintainability, accommodating new changes easily.

04 Criteria / Filter Pattern - Filtering Books

Relevant Code Files:

- FilterCriteria.java
- AuthorFilterCriteria.java
- GenreFilterCriteria.java
- RatingFilterCriteria.java
- LibraryResource.java



Feature 3

Online Integration

Why we chose JavaScript?

Design pattern : Design pattern is a general repeatable solution to a commonly occurring problem in software design .

Design patterns are not tied to any specific programming language.

Key Benefits:

- **Generalizability:** Acknowledging that design patterns are not language-bound but offer solutions applicable across various programming environments.
- **Enhanced Understanding:** By implementing in a different language, we fostered a deeper understanding of the underlying principles and mechanics of each design pattern.
- **Adaptability:** Equipping ourselves with the ability to utilize design patterns effectively in any project or language, enhancing versatility as software developers.

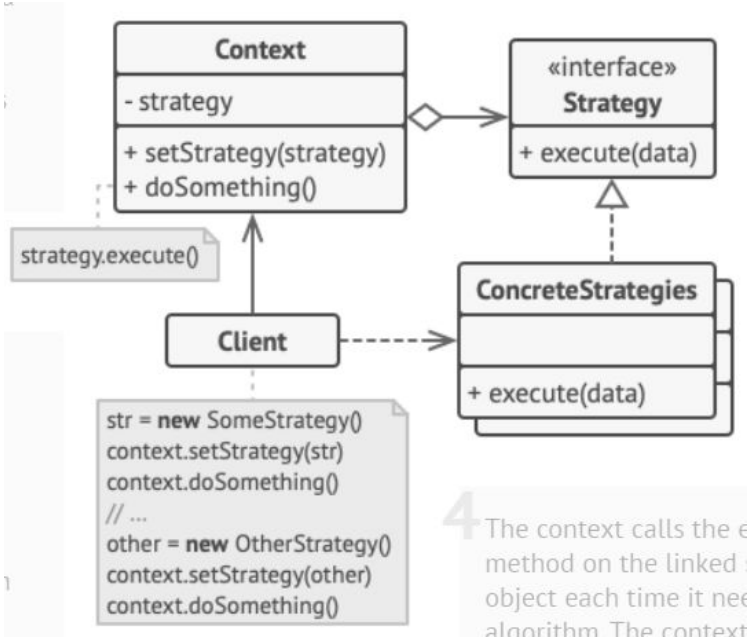
Design Patterns Used

01 Strategy Pattern - Selection of Content type selection

Rationale:
Understanding the Strategy Design Pattern

Definition:
The Strategy design pattern is a behavioral pattern that allows defining a family of algorithms, encapsulating each one, and making them interchangeable. It enables the user to choose from multiple algorithms or strategies dynamically.

Explanation:



Design Patterns Used

01 Strategy Pattern - Selection of Content type selection

Application in ItunesController:

In the ItunesController, the Strategy pattern is employed to handle different types of search strategies, such as searching for podcasts or audiobooks. Each search strategy is encapsulated within its own class, allowing for independent modification or extension without affecting the main controller logic.

Design Patterns Used

01 Strategy Pattern - Selection of Content type selection

Components of the Strategy Design Pattern

Strategy:

Defines a common interface for all supported types. In our case, the Strategy class declares the search method, which serves as the common interface for podcast and audiobook search strategies.

Concrete Strategies:

Concrete implementations of the Strategy interface. In the ItunesController, PodcastStrategy and AudiobookStrategy are concrete strategies that encapsulate the algorithms for searching podcasts and audiobooks, respectively.

Context:

The context refers to the class that uses the Strategy pattern. In our example, the ItunesController acts as the context, which delegates the search operation to the selected strategy based on user input.

Design Patterns Used

01 Strategy Pattern - Selection of Content type selection

Benefits and Advantages of the Strategy Design Pattern

Flexibility:

The Strategy pattern enables easy swapping of algorithms at runtime, allowing the system to adapt to changing requirements or user preferences without altering the core logic.

Modularity:

By encapsulating each algorithm within its own class, the Strategy pattern promotes code organization and modularity, making it easier to maintain and extend the system.

Promotes Reusability:

Strategies can be reused in different contexts or applications, promoting code reuse and minimizing duplication of code.

Enhances Readability:

The use of the Strategy pattern improves code readability by clearly separating the algorithmic logic from the main application logic, making the codebase easier to understand and maintain.

02 Adapter Pattern - Representing the results as table

Rationale:

Understanding the Adapter Design Pattern

Definition:

The Adapter design pattern allows objects with incompatible interfaces to work together by providing a wrapper or adapter around one of the objects. It acts as a bridge between the interface of a client and a target object, enabling them to collaborate seamlessly.

Purpose:

In the context of the ItunesController, the Adapter pattern is used to adapt the raw data obtained from the iTunes API into a format that is suitable for the application. It facilitates the integration of external data sources by transforming their interface into one that the application expects.

02 Adapter Pattern - Representing the results as table

Rationale:

Components of the Adapter Design Pattern

Adapter:

In the ItunesController, the DataAdapter class serves as the adapter. It encapsulates the logic for adapting the raw data obtained from the iTunes API into a format compatible with the application's requirements.

Adaptee:

The Adaptee refers to the existing interface that needs to be adapted. In this case, the raw data obtained from the iTunes API serves as the Adaptee, which requires transformation to align with the application's interface.

Client:

The client in this scenario is the ItunesController itself, which utilizes the adapter to convert the data obtained from the iTunes API into a format suitable for rendering search results.

02 Adapter Pattern - Representing the results as table

Benefits of the Adapter Design Pattern in ItunesController:

Compatibility:

Enables integration of the ItunesController with the iTunes API, accommodating differences in interface structures. This ensures seamless interaction between the application and external data sources.

Reusability:

Adapters, such as the DataAdapter class, can be reused across different features or projects within the ItunesController. They provide a standardized mechanism for parsing and adapting data from external APIs.

Encapsulation:

The DataAdapter encapsulates the logic for parsing and adapting raw API responses into a format compatible with the application's requirements. This keeps the client code in the ItunesController clean and focused on business logic.

Flexibility:

Allows for easy replacement or update of adapters without impacting the ItunesController's core functionality. As long as the adapter maintains the expected interface, changes to the parsing or adaptation process can be made seamlessly.

Feature Overview - Saving Favorites

Feature Description:

The "Saving Favorites" feature allows users to mark any audiobook or podcast as a favorite, which is then saved in the database for that user. Users can access their favorite items in a separate section within the application.

Objective:

Enhance user experience by providing a convenient way for users to bookmark their preferred audiobooks and podcasts for easy access later.

Implementation Plan:

User Interface:

Add a "Favorite" tab in the navigation bar and add a new column for adding the favourite in the table.

Database Schema:

Design database tables to store user favorite items, associating them with the user's unique identifier.

Feature Overview - Saving Favorites

Command Pattern Application to Saving Favorites:

The Command pattern encapsulates the user's request to add or remove a favorite item as an object. This abstraction allows the application to parameterize these operations, queue requests, and potentially support undoable actions.

Command Objects:

Each user action to add or remove a favorite item is represented by a command object, such as "Add to Favorites" and "Remove from Favorites."

Receiver:

The receiver is the service managing user favorites, responsible for adding or removing items from the user's favorites list.

Invoker:

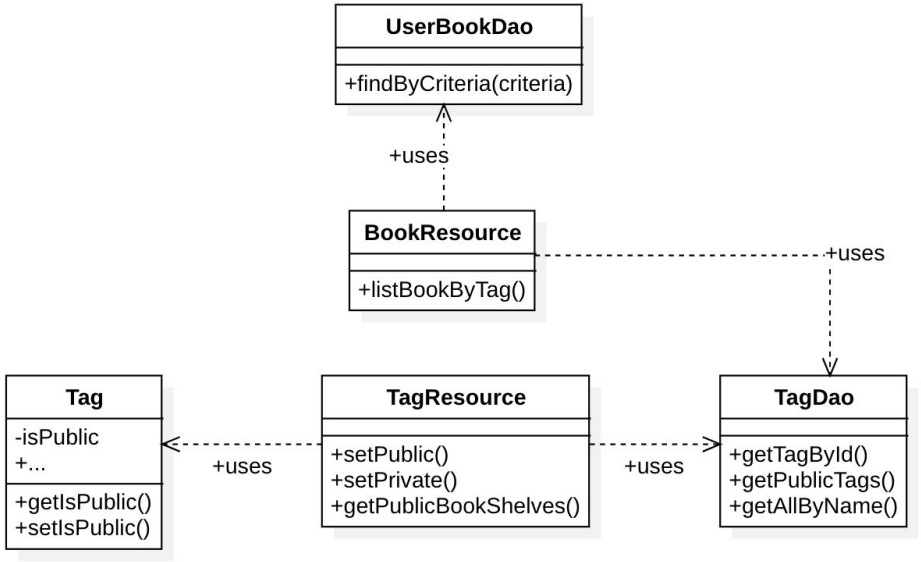
UI components in our case buttons to add to favorite serve as invokers, triggering the execution of command objects when users interact.

Client:

The application acts as the client, orchestrating command object execution based on user actions. It coordinates between UI components, command objects, and the receiver to fulfill user requests for managing favorites.

Bonus Task

Bonus Task



Class Diagram

Bonus Task: Procedure

- **Step 1:** Define a new attribute for Tag class named isPublic
- **Step 2:** Checkbox for setting public/private
- **Step 3:** setPublic() and setPrivate() functions called of TagResource class when checkbox is checked or unchecked
- **Step 4:** Create function to get books of selected public tag: listBookByTag()
- **Step 5:** Create frontend for showing list of public bookshelves

Challenges faced

01 Understanding the codebase

02 Learning and integrating with Angular

03 Identifying the best fit design patterns for features

04 Testing and Debugging

Report Link

Report: project2_23

Bonus Task Report: project2_bonus_23

THANKS!

Questions are welcome!