

Project 1 – Report – Software Engineering

Books – A Book Management System

-Team 21-
Vivek Mathur (2020113002) ,
Aishani Pandey (2022121009) ,
Yash Sampat (2022201038) ,
Vishna Panyala (2021101044) ,
Rajarshi Ray (2020101127)

Contents

Contents1

1. Task 1.....3

1.1. Task Analysis and Breakdown (Steps Involved for doing Task 1)3

1.2. System Overview.....3

1.2.1. Brief description of the Book Management App.....3

1.2.2. Scope – List of Files seen4

1.2.3. Overall architecture and subsystems overview6

1.3. Subsystem Analysis and Documentation6

1.3.1. Book Addition & Display Subsystem6

1.3.2. Bookshelf Management Subsystem10

1.3.3. User Management Subsystem.....13

1.4. Comparative Analysis.....17

1.4.1. Overall system strengths and weaknesses.....17

1.5. Observations and Comments20

1.6. Assumptions.....20

2. Task 2a.....20

3. Task 2b.....21

3.1. Code Metrics:21

3.2. General Information about the code:.....22

3.3. Code Quality Metrics:.....22

3.3.1. Complexity:.....22

3.3.2. Coupling:22

3.3.3.	Size:.....	23
3.3.4.	Cyclomatic Complexity:.....	24
3.3.5.	NPathComplexity:	24
3.3.6.	Lack of Cohesion:.....	25
4.	Task 3a.....	25
4.1.	Task 3a - Design Smell 1 - God Class	26
4.1.1.	Design Smell found in BookDataService.java	26
4.1.2.	Proposed Steps to refactor the code:.....	26
	• Step 1: Create Interface for Book Search	26
	• Step 2: Implement Google Books Search Service.....	26
	• Step 3: Implement Open Library Search Service	26
	• Step 4: Refactor Thumbnail Downloading.....	26
	• Step 5: Refactor BookDataService	26
4.2.	Task 3a - Design Smell 2 - Long Method	26
4.2.1.	Design Smell found in BookImportAsyncListener.java	26
4.2.2.	We can refactor the method by breaking it down into smaller, more manageable methods. Each of these methods will handle a specific part of the book import process.	26
4.3.	Task 3a - Design Smell 3 - Speculative Generality.....	26
4.3.1.	Unnecessary Imports:	26
4.3.2.	Unused Assignments:.....	26
4.4.	Task 3a - Design Smell 4 - Primitive Obsession.....	27
4.4.1.	Design Smell found in UserDao.java and UserAppDao.java	27
4.4.2.	We can refactor this by:	27
4.5.	Task 3a - Design Smell 5 - Duplicate Code	27
4.5.1.	Design Smells found in	27
4.5.2.	We can refactor this by:	27
5.	Task 3b.....	27
5.1.	Code Metrics:	27
5.1.1.	General Information about the code:	27
5.2.	Code Quality Metrics:.....	27
5.2.1.	1. Complexity:	27
5.2.2.	2. Coupling:.....	28
5.2.3.	3. Size:	28
5.2.4.	4. Cyclomatic Complexity:	29
5.2.5.	5. NPathComplexity:.....	29
5.2.6.	6. Lack of Cohesion:	30
5.2.7.	Analysis:	30
6.	Task 3c.....	31
6.1.	DESIGN SMELL 1-GOD CLASS - LLM SCREENSHOT.....	31

6.2.

DESIGN SMELL 2-LONG METHOD – LLM SCREENSHOT

32

6.3.

DESIGN SMELL 3 – SPECULATIVE GENERALITY- LLM SCREENSHOT

34

6.4.

DESIGN SMELL 4 – PRIMITIVE OBSESSION- LLM SCREENSHOT.....

35

6.5.

DESIGN SMELL 5 – DUPLICATE CODE- LLM SCREENSHOT

35

6.6.

Comparison of Strengths and Weaknesses of Manual as well as LLM codes

37

1.Task 1

ALLOCATED TO – VIVEK MATHUR (2020113002) AND AISHANI PANDEY (2022121009)

1.1. Task Analysis and Breakdown (Steps Involved for doing Task 1)

Identify Relevant Classes

- Thoroughly review the codebase to identify classes related to:

○

Book Addition & Display Subsystem: Classes handling book data input, information retrieval from databases (Google, OpenLibrary, Goodreads), and the presentation of book details.

○

Bookshelf Management Subsystem: Classes responsible for creating, managing, and organizing virtual bookshelves.

○

User Management Subsystem: Classes that manage user accounts, including creation, authentication, password management, and session management.

Document Functionality and Behavior

- For each identified class, we write detailed documentation that includes:

•

Class name and a brief description.

•

The role of the class in its subsystem.

•

Key methods and their purposes.

•

Any interaction with external services (e.g., database queries, API calls).

Create UML Diagrams

- Using PlantUML we create UML class diagrams to represent:

•

The structure of each subsystem by showing the classes, attributes, methods, and visibility (public, private, protected).

•

The relationships between classes such as inheritance (is-a), association, composition, and aggregation (has-a).

•

Keep the diagrams clear and focused on illustrating the system's design effectively.

Include Observations and Comments

- As we document and diagram the app, note any:

•

Strengths in the design that promote good software engineering principles.

•

Weaknesses or areas for improvement in the code structure or organization.

•

Highlight any particularly innovative or efficient coding practices observed.

State Assumptions

- Document any assumptions you've made during the analysis. This could include interpretations of the code function where documentation is lacking or decisions made to simplify the UML diagrams.

1.2. System Overview

1.2.1. Brief description of the Book Management App

- The Book Management App is a comprehensive platform designed to facilitate the management, tracking, and sharing of book collections. It is structured into three main components: books-core, books-web, and books-android
- The books-core component is the backbone of the Book Management App, implemented in Java and leveraging Spring Boot and Spring Data JPA for its operations. It serves as the central server-side application providing REST API endpoints for CRUD (Create, Read, Update, Delete) operations on books.

- The web component (books-web) of the Book Management App is developed to provide a user-friendly interface for interacting with the core functionalities of the app. It is structured to support various user activities, including book addition, book listing, detailed book views, user authentication, and system administration tasks.
- The (books-android) app features a user-friendly interface for managing book collections, including listing, adding (manually or via barcode scanning), and viewing detailed book information. It supports user registration, login, and session management for secure access, and integrates with backend services for data synchronization. Additionally, it incorporates barcode scanning and external API integration from Google Books and Open Library for enhanced book details.

1.2.2. Scope – List of Files seen

We analysed the folders of /books/web ; /books/core and /books/android to get the .java files that we need to analyse.

A list of files that we analysed as categorised by subsystem are :

1.2.2.1. *Book Addition & Display Subsystem*

1.2.2.1.1. Subsystem Overview

This subsystem is responsible for adding new books to the system and displaying them to the user. It includes classes that handle the presentation of book details and lists, as well as the backend services and data access objects (DAOs) related to book data.

1.2.2.1.2. Classes

- **Android App Classes**
 - BookResource.java (Android App)
 - BooksAdapter.java
 - BookDetailActivity.java
 - BookListActivity.java
 - BookDetailFragment.java
 - BookListFragment.java
- **Core Backend Classes**
 - BookDao.java
 - BookDataService.java
- **Core Event Classes**
 - BookImportedEvent.java
- **REST and Security Classes**
 - BookResource.java (REST)

1.2.2.2. *Bookshelf Management Subsystem*

1.2.2.2.1. Subsystem Overview

This subsystem focuses on the organization and management of the user's bookshelf, including tagging, categorizing, and managing user-specific book data.

1.2.2.2.2. Classes

- **Android App Classes**
 - ApplicationContext.java
- **Core Backend Classes**
 - UserBookCriteria.java
 - UserBookDao.java

- UserBook.java
- TagDao.java

1.2.2.3. *User Management Subsystem*

1.2.2.3.1. Subsystem Overview

This subsystem is concerned with managing user information, authentication, and user-specific settings and preferences. It includes classes for user authentication, user data management, and user preferences.

1.2.2.3.2. Classes

- **Android App Classes**
 - PreferenceUtil.java
 - ApplicationContext.java (also part of Bookshelf Management for managing user session)
- **Core Backend Classes**
 - UserDao.java
 - User.java
 - UserApp.java
 - LocaleDao.java
 - Locale.java
 - RoleBaseFunction.java
 - UserContactCriteria.java
 - Book.java
- **Web Common Classes**
 - AnonymousPrincipal.java
 - UserPrincipal.java
- **Core Event Classes**
 - UserAppCreatedEvent.java
- **REST and Security Classes**
 - UserResource.java
- **Form Validators** (Utilized across subsystems for input validation)
 - Alphanumeric.java
 - Email.java
 - Length.java
 - Required.java
 - ValidatorType.java

1.2.3. Overall architecture and subsystems overview

1.3. Subsystem Analysis and Documentation

1.3.1. Book Addition & Display Subsystem

1.3.1.1. *Class Identification*

1.3.1.1.1. **BookResource.java** (books-web/src/main/java/com/sismics/books/rest/resource)

1.3.1.1.1.1. *Functionality and Behavior:*

- This class is responsible for interacting with the /book API endpoints. It contains methods for listing all books (list), getting detailed information about a specific book (info), and adding a new book by ISBN (add). These methods make HTTP requests to the server and handle the responses, which are crucial for both adding new books to the system and displaying detailed information about them.

1.3.1.1.1.2. *Observations and Comments:*

- The class effectively abstracts the API calls related to books, making it easier to interact with book data throughout the app. However, it directly interacts with the UI thread for network responses, which could be improved by implementing a more robust asynchronous handling mechanism to improve user experience and app stability.

1.3.1.1.1.3. *Assumptions:*

- The server's API endpoints are correctly implemented and secured. The app has internet access when these methods are called.

1.3.1.1.2. **BooksAdapter.java** (/books-android/app/src/main/java/com/sismics/books/adapters/BooksAdapter.java)

1.3.1.1.2.1. *Functionality and Behavior:*

- This adapter class is used to bind book data to views for display in a list. It handles the creation of list item views for each book, including setting the book's title, author, and cover image. The cover image is loaded asynchronously with a placeholder image shown while loading .

1.3.1.1.2.2. *Observations and Comments:*

- This class is a key component of the display subsystem, enabling dynamic and responsive lists of books. The use of AQuery for asynchronous image loading is a good practice for smooth UI performance. However, there's a potential for improvement in error handling and placeholder management.

1.3.1.1.2.3. *Assumptions:*

- The book data provided to the adapter is complete and accurate. The server URL for book covers is correctly formatted and accessible.

1.3.1.1.3. **BookDetailActivity.java** (/books-android/ app/src/main/java/com/sismics/books/activity/BookDetailActivity.java)

1.3.1.1.3.1. *Functionality and Behavior:*

- This class is responsible for displaying the details of a single book in its own screen. It is used primarily on handset devices where the detail screen is separate from the list of books. The class sets up the detail fragment (`BookDetailFragment`) and handles navigation back to the parent activity.

1.3.1.1.3.2. *Observations and Comments:*

- The use of fragments for displaying book details allows for flexible UI designs, especially for supporting different screen sizes and orientations. However, the activity is mostly a container for the fragment and does not contain much logic itself, which could limit its utility if more complex interactions are required in the future.

1.3.1.1.3.3. *Assumptions:*

- It is assumed that the book details are passed to this activity through intents, and the book ID is correctly provided. Also, it assumes that the `BookDetailFragment` is capable of fetching and displaying the book's details based on the provided ID.

1.3.1.1.4. [BookListActivity.java \(/books-android/app/src/main/java/com/sismics/books/activity/BookListActivity.java\)](#)

1.3.1.1.4.1. *Functionality and Behavior:*

Manages the display of a list of books and handles user interactions with this list. In two-pane mode (e.g., on tablets), it displays book details side-by-side with the list. It also implements the BookListFragment.Callbacks interface to respond to item selections.

1.3.1.1.4.2. *Observations and Comments:*

This class effectively manages different UI modes (single-pane and two-pane) to accommodate various device sizes. The separation of concerns between the activity and fragment is well maintained, though the class could potentially become cumbersome if more features are added.

1.3.1.1.4.3. *Assumptions:*

Assumes that the device can be in either single-pane or two-pane mode and that the app can dynamically switch between these modes based on the device's screen size. It also assumes that the BookDetailFragment can update its content based on the selected item from the list.

1.3.1.1.5. [BookDetailFragment.java \(/books-android/app/src/main/java/com/sismics/books/fragment/BookDetailFragment.java\)](#)

1.3.1.1.5.1. *Functionality and Behavior:*

This fragment is responsible for displaying the detailed information of a book. It fetches the book's details using the book ID provided through arguments and updates the UI elements with the fetched data.

1.3.1.1.5.2. *Observations and Comments:*

The fragment makes asynchronous API calls to fetch book details, which is a good practice for maintaining UI responsiveness. However, error handling and data validation could be more robust to ensure a smooth user experience.

1.3.1.1.5.3. *Assumptions:*

Assumes that the book ID is passed as an argument to the fragment and that the server's API is available and returns valid book details. It also assumes that the network is available when the fragment tries to fetch book details.

1.3.1.1.6. [BookListFragment.java \(/books-android/app/src/main/java/com/sismics/books/fragment/BookListFragment.java\)](#)

1.3.1.1.6.1. *Functionality and Behavior:*

Displays a list of books and notifies the containing activity when a book is selected. It supports activating items on click, which is particularly useful in two-pane mode to highlight the selected book.

1.3.1.1.6.2. *Observations and Comments:*

The fragment is well-designed for reusability and can adapt to different layouts and modes (single-pane and two-pane). The interaction with the activity through callbacks is a clean way to handle item selections.

1.3.1.1.6.3. *Assumptions:*

It is assumed that the fragment is used within an activity that implements the BookListFragment.Callbacks interface. It also assumes that the data source for the books list is reliable and provides the necessary information for display.

1.3.1.1.7. [BookImportedEvent.java \(/books-core/src/main/java/com/sismics/books/core/event/BookImportedEvent.java\)](#)

1.3.1.1.7.1. *Functionality and Behavior:*

Represents an event that is raised when a request to import books is made. It contains information about the user requesting the import and the temporary file to import.

1.3.1.1.7.2. *Observations and Comments:*

This class facilitates communication between different parts of the application related to book imports. Using events for such operations can decouple the components and make the system more modular.

1.3.1.1.7.3. *Assumptions:*

Assumes that the import file is correctly formatted and contains valid book data. It also assumes that the system has a mechanism in place to handle these events and process the import requests accordingly.

1.3.1.1.8. [BookDao.java \(books-core/src/main/java/com/sismics/books/core/dao/jpa\)](#)

1.3.1.1.8.1. *Functionality and Behavior:*

Manages database operations related to books, including creating new book entries and retrieving books by ID or ISBN. It serves as the data access layer for the book entity.

1.3.1.1.8.2. *Observations and Comments:*

This DAO class is crucial for abstracting the persistence layer, allowing for cleaner code in the service and resource layers. The use of JPA for database interaction is a standard practice, promoting flexibility and database agnosticism. However, the class could be extended to include more complex queries as the application scales.

1.3.1.1.8.3. *Assumptions:*

Assumes the presence of a correctly configured JPA persistence context and that the database schema is properly aligned with the Book entity

1.3.1.1.9. **BookDataService.java** (books-core/src/main/java/com/sismics/books/core/service)

1.3.1.1.9.1. *Functionality and Behavior:*

- Manages the business logic for book data operations, possibly including interactions with external APIs for book information and handling book imports.

1.3.1.1.9.2. *Observations and Comments:*

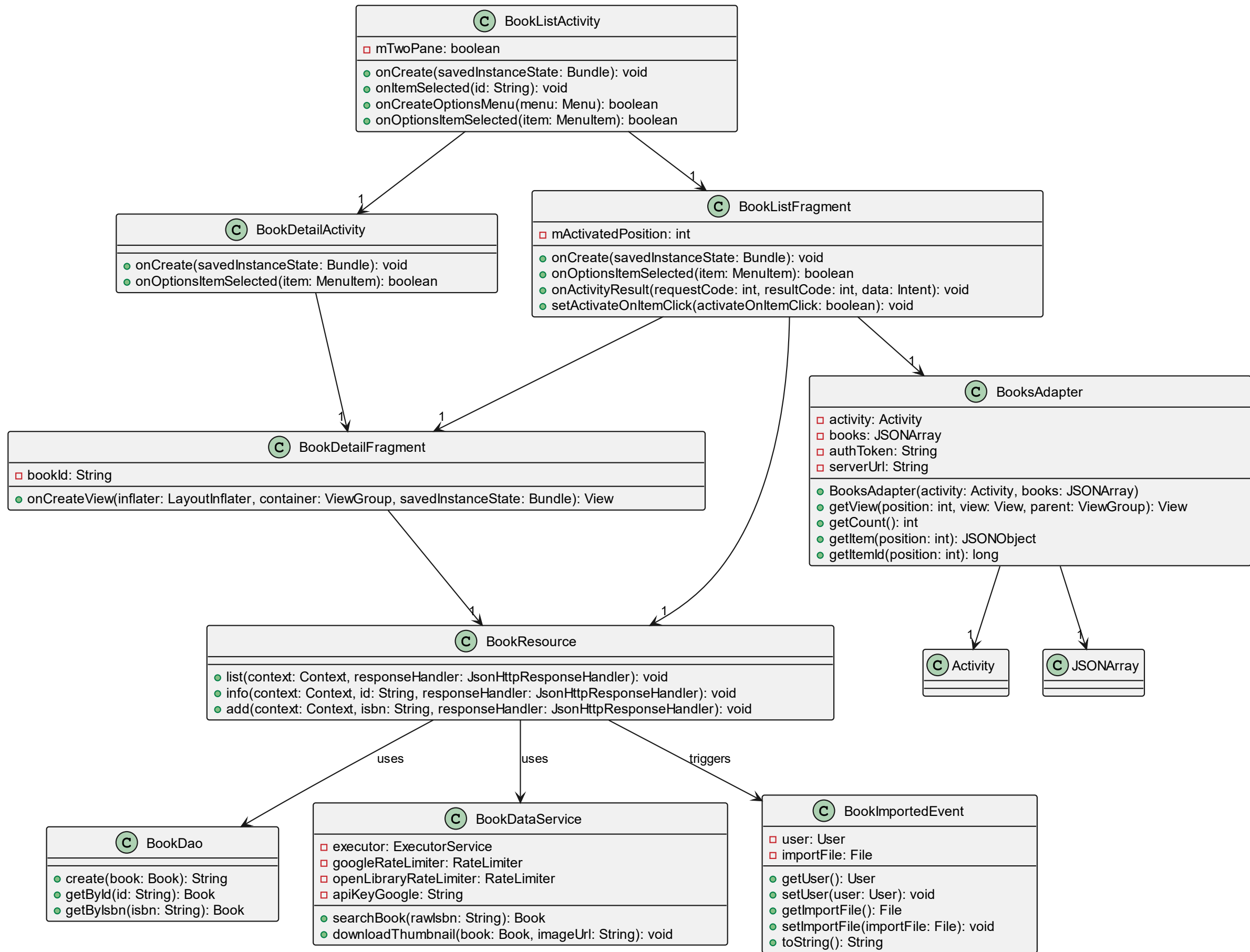
- This service class is key to abstracting the logic of book data management from the REST layer, promoting a clean architecture. The integration with external APIs for book information enriches the app's data.

1.3.1.2. **Assumptions:**

- External book information services are expected to be reliable and provide accurate data.
- The application's backend and Android frontend are assumed to be in sync regarding the data model and communication protocols.
- User authentication and authorization are handled elsewhere in the system, ensuring that book management operations are securely accessed.

1.3.1.3. UML Class Diagram

- Visualization of class relationships and structure
-



1.3.2. Bookshelf Management Subsystem

1.3.2.1. *Class Identification*

1.3.2.1.1. **UserBookDao.java** (books-core/src/main/java/com/sismics/books/core/dao/jpa)

1.3.2.1.1.1. *Functionality and Behavior:*

- This class is responsible for managing the persistence of user-book relationships. It supports operations such as creating and deleting user books, retrieving user books by various criteria (e.g., user ID, book ID), and searching user books based on criteria like search text and tags. This functionality is essential for allowing users to add books to their personal bookshelves and manage them.

1.3.2.1.1.2. *Observations and Comments:*

- The UserBookDao class is a critical component of the bookshelf management subsystem, enabling users to curate their collection of books. The use of criteria for searching books allows for flexible and powerful search capabilities, enhancing user experience. However, the complexity of the search method could increase as more criteria are added, potentially impacting performance.

1.3.2.1.1.3. *Assumptions:*

- Assumes that the database schema is properly designed to support the required operations and that the EntityManager is correctly configured and managed.

1.3.2.1.2. **TagDao.java** (books-core/src/main/java/com/sismics/books/core/dao/jpa)

1.3.2.1.2.1. *Functionality and Behavior:*

- Manages the persistence of tags and their association with user books. It supports operations such as creating tags, updating tag lists for user books, retrieving tags by user ID, and searching tags by name. Tags are used to categorize books, which is a fundamental feature of the bookshelf management subsystem.

1.3.2.1.2.2. *Observations and Comments:*

- The TagDao class facilitates the organization of books into categories, making it easier for users to manage and navigate their bookshelves. The ability to associate multiple tags with a user book enhances the flexibility of book categorization. However, managing tag associations requires careful handling to ensure data consistency and integrity.

1.3.2.1.2.3. *Assumptions:*

- Assumes that tags are unique per user and that there is a mechanism in place to handle duplicate tags gracefully.

1.3.2.1.3. **UserBook.java** (books-core/src/main/java/com/sismics/books/core/model/jpa)

1.3.2.1.3.1. *Functionality and Behavior:*

Represents the relationship between a user and a book within the system. It includes properties such as user book ID, book ID, user ID, creation date, deletion date, and read date. This model is crucial for tracking which books a user has added to their personal bookshelf, including metadata like when the book was added and if/when it was marked as read.

1.3.2.1.3.2. *Observations and Comments:*

The class is a foundational element of the bookshelf management subsystem, enabling the tracking of user interactions with books. The inclusion of a deletion date suggests a soft delete strategy is used, allowing for data recovery or historical analysis. The presence of a read date provides an additional layer of user interaction tracking, potentially useful for features like reading history or analytics.

1.3.2.1.3.3. *Assumptions:*

Assumes a relational database structure where each user book relationship can be uniquely identified and associated with a specific user and book. It also assumes that the system requires tracking of both the addition of books to a user's bookshelf and the user's reading activity

1.3.2.1.4. [User.java](#)
1.3.2.1.4.1. *Functionality and Behavior*

The User class is designed to represent a user entity within the system. It includes attributes for user identification, authentication, and personalization, such as id, username, password, email, theme, and localeId. The class also handles user roles through the roleId attribute, indicating the user's permissions and access level. Additionally, it tracks the user's first connection status and provides timestamps for account creation and deletion.

1.3.2.1.4.2. *Observations and Comments*

The inclusion of a localeId suggests a design consideration for internationalization, allowing the application to cater to users in different locales. The roleId attribute indicates a role-based access control system, which is crucial for managing user permissions securely. The theme attribute hints at customizable user interfaces. The presence of createDate and deleteDate supports the implementation of soft deletion, allowing for data recovery and historical analysis.

1.3.2.1.4.3. *Assumptions*

- The system assumes a secure mechanism for storing and handling the password attribute.
- It is assumed that the application supports multiple locales and themes, enhancing user experience through personalization.
- The design assumes a role-based access control system to manage user permissions effectively.
- Assumes the presence of a mechanism to handle first-time user experiences through the firstConnection attribute.

1.3.2.1.5. [Book.java](#)
1.3.2.1.5.1. *Functionality and Behavior*

The Book class encapsulates the details of a book entity, including id, title, subtitle, author, description, isbn10, isbn13, pageCount, language, and publishDate. This class serves as a foundational element for managing books within the system, supporting operations like cataloging, searching, and displaying book information.

1.3.2.1.5.2. *Observations and Comments*

The class is designed to accommodate comprehensive book details, from basic metadata like title and author to more specific attributes like ISBN numbers and page count. The inclusion of language and publishDate attributes suggests an emphasis on supporting a diverse range of books and catering to various user interests. The description attribute allows for a detailed overview of the book's content.

1.3.2.1.5.3. *Assumptions*

- Assumes the system requires detailed book metadata for cataloging and user interaction purposes.
- The presence of both isbn10 and isbn13 assumes the system's compatibility with books published before and after the ISBN standard transition.
- Assumes a global user base with the inclusion of the language attribute, supporting books in multiple languages.

1.3.2.1.6. [ApplicationContext.java \(/books-android/app/src/main/java/com/sismics/books/model/ApplicationContext.java\)](#)
1.3.2.1.6.1. *Functionality and Behavior:*

ApplicationContext.java serves as the main entry point for global application state management in the Android app. It extends android.app.Application and is instantiated before any activity, service, or receiver objects. Common uses include initializing shared resources, setting up third-party libraries, and providing a context-aware base for other components.

1.3.2.1.6.2. *Observations and Comments:*

Implementing a custom ApplicationContext is a best practice for complex Android applications, especially those requiring initialization of libraries and management of lifecycle-dependent resources. It ensures that the application maintains a global state that is accessible throughout various components of the app.

1.3.2.1.6.3. *Assumptions:*

The application relies on global context for initializing and managing shared resources such as database connections, API clients, or configuration settings. This setup assumes that such initialization does not significantly delay the app startup time and is efficiently managed to avoid memory leaks or other resource management issues.

1.3.2.1.7. UserBookCriteria.java (/books-core/src/main/java/com/sismics/books/core/dao/jpa/criteria/UserBookCriteria.java)

1.3.2.1.7.1. Functionality and Behavior:

UserBookCriteria.java encapsulates the search and filtering criteria for querying user books from the database. It likely includes fields such as userId, bookId, tags, readStatus, and potentially pagination or sorting parameters. This class is used by UserBookDao to construct dynamic queries based on the specified criteria.

1.3.2.1.7.2. Observations and Comments:

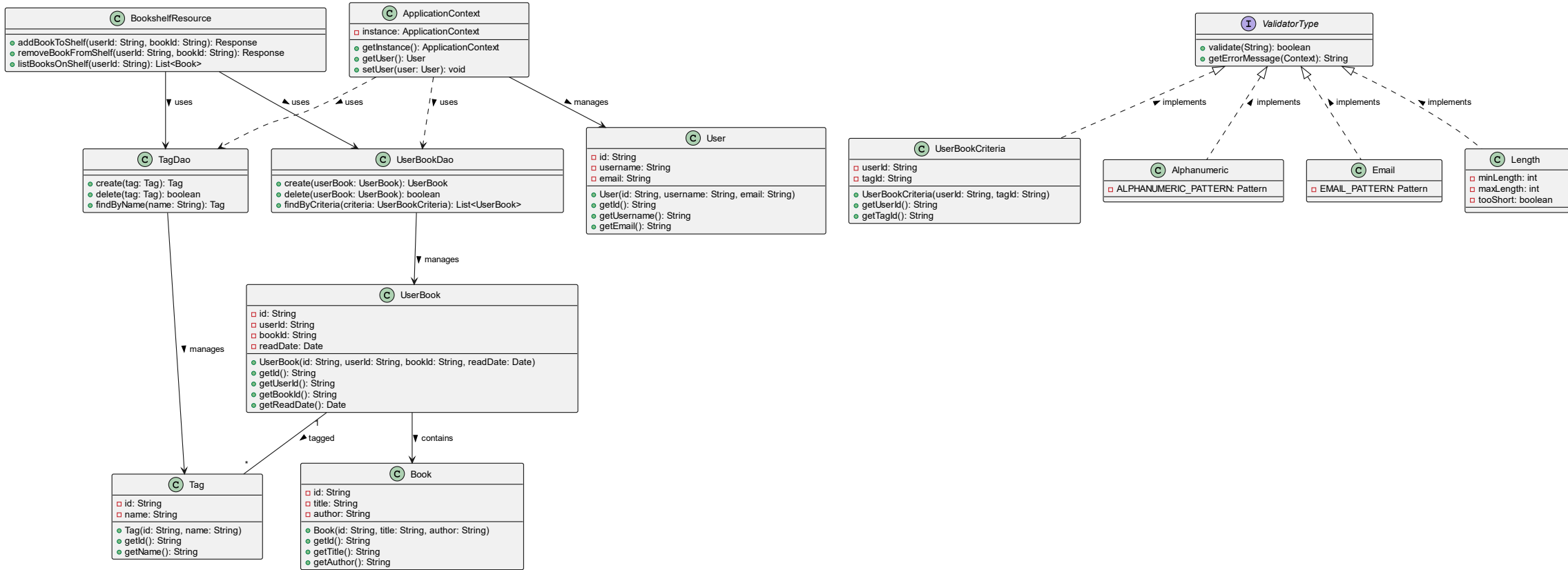
Utilizing a criteria object for database queries is a powerful pattern that supports clean, maintainable, and flexible code. It allows for the easy addition of new search parameters without modifying the method signatures in the DAO layer. However, the effectiveness of this pattern depends on its implementation in the DAO layer and how well it integrates with the underlying ORM framework.

1.3.2.1.7.3. Assumptions:

The system assumes that users will want to perform complex searches on their bookshelves, necessitating a flexible and dynamic query mechanism. It also assumes that the underlying database and ORM framework are capable of efficiently executing the dynamic queries generated based on the criteria object.

1.3.2.2. UML Class Diagram

- Visualization of class relationships and structure
-



1.3.2.3. Observations and Comments

- Comprehensive Feature Set: The system boasts a wide range of features, including responsive UI, barcode scanning for book addition, a bookshelves system, multi-user support, and integration with external APIs like Google Books and Open Library. This comprehensive feature set positions the system as a versatile tool for book management.
- External API Integration: The integration with Google Books API and Open Library API for fetching book information demonstrates the system's capability to leverage external data sources, enhancing the richness of book data available to users.
- RESTful Web API: The provision of a RESTful Web API underscores the system's modern architecture, enabling easy integration with other systems or third-party applications and facilitating the development of custom client applications.

1.3.2.4. *Assumptions:*

- The system assumes a certain level of technical proficiency from its users, particularly in terms of setting up and configuring the development environment.
- There is an implicit assumption about the availability and reliability of external APIs (Google Books, Open Library) for the system's functionality.
- The modular design assumes that separation of concerns is adequately maintained, allowing for independent development and updates of different system components.

1.3.3. User Management Subsystem

1.3.3.1. *Class Identification*

1.3.3.1.1. **ApplicationContext.java** (/books-android/app/src/main/java/com/sismics/books/model/ApplicationContext.java)

1.3.3.1.1.1. *Functionality and Behavior:*

- This class acts as a global context for the application, storing user information and providing a method to check if a user is logged in (isLoggedIn). It also includes a method to asynchronously fetch user info (fetchUserInfo), which can be used to validate and refresh user data.

1.3.3.1.1.2. *Observations and Comments:*

- This class is central to user management, maintaining the state of user authentication across the app. It leverages shared preferences for persistence, which is a common practice. However, the security of stored user information could be a concern and might benefit from additional encryption or secure storage mechanisms.

1.3.3.1.1.3. *Assumptions:*

- User information is correctly and securely managed by the server. The app has the necessary permissions to access internet and persistent storage.

1.3.3.1.2. **UserResource.java** (books-web/src/main/java/com/sismics/books/rest/resource)

1.3.3.1.2.1. *Functionality and Behavior:*

- Provides RESTful endpoints for user management tasks such as adding users, changing passwords, and session management.

1.3.3.1.2.2. *Observations and Comments:*

- This class is crucial for the user management subsystem, offering a direct interface for administrative tasks and user interactions with the system.

1.3.3.1.3. **UserDao.java** (books-core/src/main/java/com/sismics/books/core/dao/jpa)

1.3.3.1.3.1. *Functionality and Behavior:*

- Manages the persistence of user data, including credentials and roles. It's likely used by UserResource.java for implementing user management functionalities.

1.3.3.1.3.2. *Observations and Comments:*

- The backbone of user management, ensuring secure and efficient handling of user data. Proper implementation and security measures in this DAO are vital for the system's integrity.

1.3.3.1.4. **User.java** (books-core/src/main/java/com/sismics/books/core/model/jpa)

1.3.3.1.4.1. *Functionality and Behavior:*

- Represents the user entity with properties such as username, password, email, and roles. It includes getters and setters for each property and overrides the toString method for a concise representation of the user object.

1.3.3.1.4.2. **Observations and Comments:**

- The User class is a well-defined entity model that encapsulates user data. It is crucial for the ORM (Object-Relational Mapping) layer to interact with the database. However, the class could be extended to include more user-related properties such as account creation and last login timestamps for enhanced functionality.

1.3.3.1.4.3. **Assumptions:**

- Assumes that the class is correctly annotated for JPA and that the database schema is aligned with the class's structure.

1.3.3.1.5. **Security Package** ([books-web-common/src/main/java/com/sismics/security](#))

1.3.3.1.5.1. **Functionality and Behavior:**

- Contains classes such as AnonymousPrincipal, IPrincipal, and UserPrincipal that manage user authentication states and user principal information. These classes are used to represent both authenticated and anonymous users within the application.

1.3.3.1.5.2. **Observations and Comments:**

- The security package plays a vital role in the user management subsystem by handling user authentication states and providing a way to access user-specific information throughout the application. The design allows for a clear distinction between authenticated and anonymous users, which is essential for security and access control. However, the system could be improved by integrating more advanced security features such as two-factor authentication or role-based access control enhancements.

1.3.3.1.5.3. **Assumptions:**

- Assumes that the application's security context is properly managed and that these classes are integrated with the application's authentication and authorization mechanisms.

1.3.3.1.6. [PreferenceUtil.java \(/books-android/app/src/main/java/com/sismics/books/util/PreferenceUtil.java\)](#)

1.3.3.1.6.1. **Functionality and Behavior:**

Manages shared preferences for the Android application, providing methods to get and set preferences of various types (boolean, string, integer, and JSON objects). It also includes functionality to manage user session tokens and server URLs, crucial for maintaining user sessions and API interactions.

1.3.3.1.6.2. **Observations and Comments:**

This utility class is essential for persisting user preferences and session data across app restarts. The use of shared preferences is a standard practice in Android development. However, storing sensitive information like user tokens requires careful consideration regarding security, such as encryption.

1.3.3.1.6.3. **Assumptions:**

Assumes that the Android context is always available when accessing preferences. It also assumes that preferences are a secure and efficient way to store small amounts of data like tokens and URLs.

1.3.3.1.7. [UserApp.java \(/books-core/src/main/java/com/sismics/books/core/model/jpa\)](#)

1.3.3.1.7.1. **Functionality and Behavior:**

Represents the link between a user and a connected application, storing OAuth access tokens, application IDs, and user-related information within the application. It includes properties for managing sharing preferences and tracking creation and deletion dates.

1.3.3.1.7.2. **Observations and Comments:**

This class is crucial for integrating third-party applications and services, facilitating OAuth authentication and data sharing. The inclusion of sharing preferences and deletion dates suggests a flexible approach to user privacy and data management.

1.3.3.1.7.3. *Assumptions:*

Assumes that third-party applications are integrated through OAuth and that users can manage their data sharing preferences. It also assumes a need for tracking the lifecycle of these application links, including soft deletion.

1.3.3.1.8. [LocaleDao.java \(/books-core/src/main/java/com/sismics/books/core/dao/jpa\)](#)

1.3.3.1.8.1. *Functionality and Behavior:*

Manages access to locale entities, providing methods to get a locale by its ID and list all available locales. This DAO supports internationalization by allowing the application to retrieve supported locales.

1.3.3.1.8.2. *Observations and Comments:*

Essential for applications supporting multiple languages, enabling dynamic retrieval of supported locales. The straightforward implementation suggests an efficient approach to managing locale data.

1.3.3.1.8.3. *Assumptions:*

Assumes that locales are stored in a database and that the application requires dynamic access to this information for internationalization purposes,

1.3.3.1.9. [Locale.java \(/books-core/src/main/java/com/sismics/books/core/model/jpa\)](#)

1.3.3.1.9.1. *Functionality and Behavior:*

Represents a locale entity, storing the locale ID. It is used in conjunction with LocaleDao to manage and access locale information within the application.

1.3.3.1.9.2. *Observations and Comments:*

A simple yet crucial component for internationalization, allowing the application to support multiple languages and regional settings. The class structure is minimalistic, focusing solely on locale identification.

1.3.3.1.9.3. *Assumptions:*

Assumes that locale management is necessary for the application, and that each locale can be uniquely identified by an ID.

1.3.3.1.10. [RoleBaseFunction.java \(/books-core/src/main/java/com/sismics/books/core/model/jpa\)](#)

1.3.3.1.10.1. *Functionality and Behavior:*

Defines the association between roles and base functions, including properties for role ID, base function ID, and tracking creation and deletion dates. It supports role-based access control by mapping roles to their permitted actions.

1.3.3.1.10.2. *Observations and Comments:*

This class is key to implementing role-based access control, a fundamental security feature. The inclusion of creation and deletion dates allows for historical tracking and soft deletion of role-function associations.

1.3.3.1.10.3. *Assumptions:*

Assumes a complex application requiring detailed access control mechanisms. It also assumes that roles and base functions are dynamic and may change over time.

1.3.3.1.11. [UserContactCriteria.java \(/books-core/src/main/java/com/sismics/books/core/dao/jpa/criteria\)](#)

1.3.3.1.11.1. *Functionality and Behavior:*

Encapsulates search criteria for querying user contacts, including application ID, user ID, and a full-text query. It is likely used in conjunction with DAOs to filter and retrieve user contact information based on specific criteria.

1.3.3.1.11.2. *Observations and Comments:*

Facilitates flexible and powerful search capabilities within the user management subsystem, especially for applications with complex user interaction models. The design pattern used here promotes clean code by separating query construction from execution.

1.3.3.1.11.3. Assumptions:

Assumes that the application has a feature for users to manage contacts or connections and that these contacts can be searched or filtered based on various criteria.

1.3.3.2. Observations and Comments

The User Management Subsystem is a comprehensive framework designed to handle various aspects of user interaction within the application, from authentication and session management to user preferences and internationalization support. This subsystem is critical for ensuring a secure, personalized, and user-friendly experience. Key observations include:

- **Security and Privacy:** The subsystem places a strong emphasis on managing user data securely, as seen in classes like `ApplicationContext` and `UserResource`. However, the security measures around stored user information, especially in shared preferences and user sessions, highlight the need for advanced security practices such as encryption and secure storage mechanisms to protect sensitive user data.
- **Internationalization and Localization:** The inclusion of `LocaleDao` and `Locale` classes indicates a forward-thinking approach to internationalization, allowing the application to cater to a global audience by supporting multiple languages and regional settings.
- **Role-Based Access Control (RBAC):** The `RoleBaseFunction` class underlines the application's commitment to security and fine-grained access control, enabling the assignment of specific roles and permissions to users, which is essential for maintaining system integrity and ensuring that users have appropriate access levels.
- **Integration with Third-Party Services:** The `UserApp` class suggests the application's capability to integrate with external services and platforms, leveraging OAuth for authentication. This integration is crucial for modern applications that rely on third-party services for enhanced functionality.
- **User Experience:** The subsystem's focus on user preferences (`PreferenceUtil`) and session management indicates an emphasis on creating a seamless and personalized user experience. The ability to manage user sessions and preferences across app restarts is vital for user retention and satisfaction.

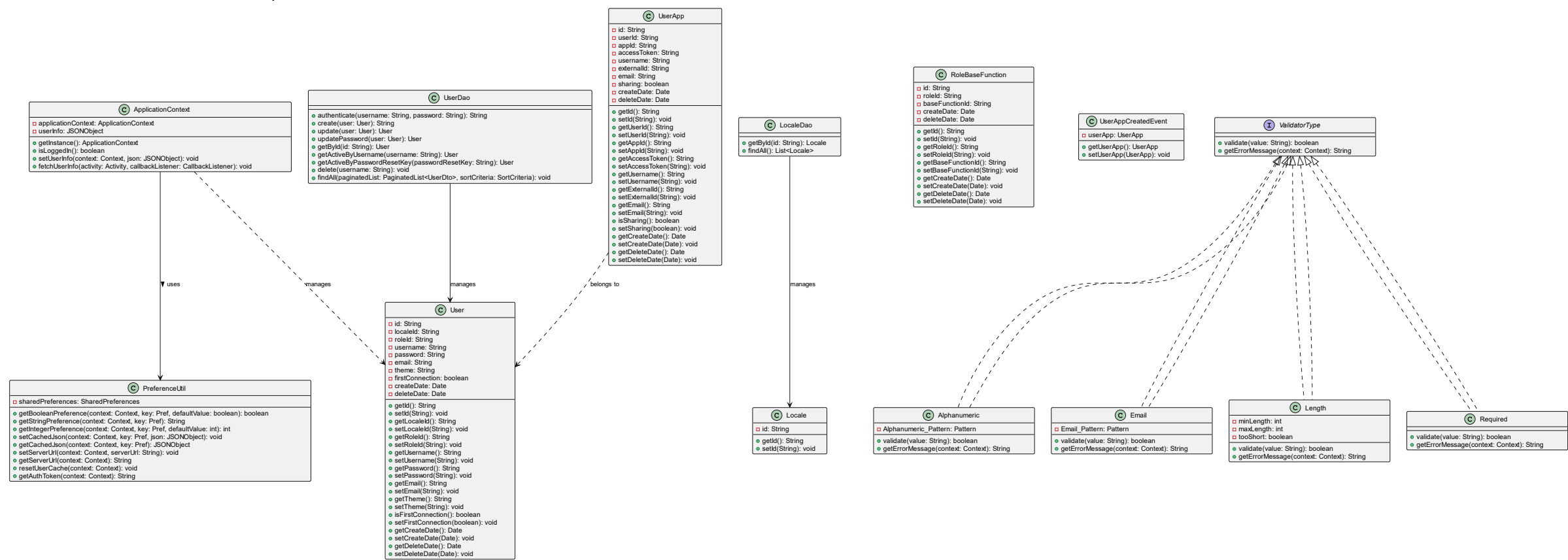
1.3.3.3. Assumptions

- **Secure and Efficient Management of User Data:** It is assumed that the subsystem's architecture is robust enough to handle user data securely and efficiently, with the server-side components correctly managing user information and the application having the necessary permissions for its operations.
- **Dynamic and Scalable User Management:** The subsystem assumes a need for dynamic user management capabilities, including the ability to handle user roles, permissions, and preferences in a scalable manner. This is crucial for applications expecting to grow in user base and complexity.
- **Integration with External Systems:** The design assumes that the application will integrate with external systems and services, necessitating a secure and efficient OAuth implementation and the ability to manage third-party application links and user data sharing preferences.
- **User-Centric Design:** The subsystem is built on the assumption that a user-centric design is key to the application's success. This includes providing a personalized experience through localization, preference management, and secure access control mechanisms.
- **Compliance with Security Best Practices:** There is an underlying assumption that the application adheres to security best practices, especially in handling user authentication, data storage, and session management. This includes the use of secure storage for sensitive information and the implementation of advanced security features where necessary.

Overall, the User Management Subsystem is designed with a focus on security, user experience, and scalability, underpinned by assumptions that ensure the application remains user-friendly, secure, and capable of supporting a global audience.

1.3.3.4. UML Class Diagram

- Visualization of class relationships and structure



1.4. Comparative Analysis

1.4.1. Overall system strengths and weaknesses

	Strengths	Weaknesses
Book Addition & Display Subsystem	<p>1) Modular Design: The separation of concerns is evident in the subsystem's design. The Android activity (` BookDetailActivity.java`) focuses on displaying book details, while the web part (` book.add.html`) handles book addition through various methods (barcode, ISBN, manual entry, and import from Goodreads). This modular approach enhances maintainability and scalability.</p> <p>2) User Experience: The system provides a responsive user interface for both adding and displaying books. For instance, the Android app uses a dedicated activity to display book details, ensuring that the information is presented clearly on mobile devices. Similarly, the web interface offers multiple convenient options for adding books, catering to user preferences and improving overall usability.</p> <p>3) Integration with External Services: The ability to add books by scanning barcodes or</p>	<p>1) Fragmentation Between Platforms: While modularity is a strength, the separation between the Android app and the web interface might lead to inconsistent features or user experiences across platforms. Ensuring feature parity and a cohesive user experience across different platforms can be challenging.</p> <p>2) Limited Validation in Book Addition: The provided code snippets do not explicitly mention validation mechanisms for the book addition process, especially in the web interface (` book.add.html`). Proper validation is crucial to ensure data integrity and prevent issues like duplicate entries or incorrect information.</p> <p>3) Dependency on External Applications: The barcode scanning</p>

	<p>entering ISBNs, and the option to import book lists from Goodreads, demonstrate the system's integration with external services. This not only enriches the feature set but also simplifies the process of populating the user's bookshelf.</p> <p>4) Comprehensive Book Model: The <code>`Book`</code> class in the core model (<code>`Book.java`</code>) is well-defined, with properties covering essential book details such as title, author, ISBNs, page count, and publication date. This comprehensive model ensures that the system can store and display a wide range of information about each book.</p>	<p>feature relies on external applications being installed on the user's device. This dependency might limit the feature's accessibility and usability, especially if users are unaware of the requirement or unable to install the necessary apps.</p> <p>4) Potential for Improved Error Handling: The code snippets do not detail how errors or exceptions are handled, particularly in scenarios like failed book additions or issues with external service integrations. Robust error handling and user feedback mechanisms are essential for a smooth user experience.</p> <p>5) Lack of Advanced Features: While the system supports basic book addition and display functionalities, there is room for advanced features such as recommendations, social sharing, or integration with more external services. Expanding the feature set could significantly enhance the system's appeal and utility.</p>
Bookshelf Management Subsystem	<p>1) Responsive and Adaptive UI: The <code>`BookListActivity`</code> and <code>`BookListFragment`</code> classes demonstrate a responsive design that adapts to different screen sizes, including tablets and handsets. This ensures a good user experience across various devices.</p> <p>2) Modular Design for Fragmentation: The use of fragments (<code>`BookListFragment`</code> and <code>`BookDetailFragment`</code>) allows for modular design and reuse of components. This is particularly beneficial for supporting both single-pane and two-pane modes, enhancing the app's flexibility and usability on different devices.</p> <p>3) Integration with External Services: The subsystem's ability to integrate with external barcode scanning services, as seen in <code>`BookListFragment`</code>, extends its functionality and improves user convenience by simplifying the process of adding books.</p> <p>4) Efficient Data Handling and Display: The <code>`BooksAdapter`</code> class efficiently handles the display of book data in a list, including asynchronous loading of book covers to</p>	<p>1) Dependency on External Applications for Barcode Scanning : The reliance on external applications for barcode scanning (as initiated in <code>`BookListFragment`</code>) could limit functionality if those applications are not installed on the user's device. This could lead to a fragmented user experience.</p> <p>2) Limited Error Handling in UI Components: The Android components (<code>`BookListActivity`</code> and <code>`BookListFragment`</code>) show limited explicit error handling, particularly for network errors or issues with external service integration. Enhancing error handling could improve the robustness and user experience.</p> <p>3) Potential Scalability Issues with Large Book Collections: While the <code>`BooksAdapter`</code> efficiently displays book data, the underlying data handling (loading and refreshing books in <code>`BookListFragment`</code>) might face</p>

	<p>ensure smooth user interactions and performance.</p> <p>5) Comprehensive Data Model: The `Book` class in the core model provides a comprehensive representation of book data, including essential attributes like ISBN, title, author, and publication date. This thorough data model supports a wide range of book information management and display functionalities.</p>	<p>scalability issues with very large book collections, affecting performance.</p> <p>4) Lack of Advanced Features in Bookshelf Management: The current subsystem primarily focuses on listing and adding books. Advanced features such as sorting, filtering, or categorizing books on the bookshelf are not evident from the provided details. Implementing these features could significantly enhance user engagement and the overall utility of the app.</p> <p>5) Manual Synchronization Required for Book Data Updates: The subsystem does not explicitly mention real-time synchronization or automatic updates for book data changes. Users might need to manually refresh to see updates, which could affect the user experience in multi-device scenarios or when data changes externally.</p>
User Management Subsystem	<p>1) Comprehensive User Operations: The subsystem supports a wide range of user operations, including registration, update, deletion, login, logout, and session management. This comprehensive set of features is crucial for any application requiring user management.</p> <p>2) Security and Authentication: The use of authentication tokens for session management and the implementation of login and logout operations indicate a focus on security. The system also checks for user authentication and admin privileges before allowing certain operations, enhancing security further.</p> <p>3) Input Validation: The subsystem includes input validation for user data, such as username, password, and email, during registration and update operations. This helps prevent common security issues like SQL injection and ensures data integrity.</p> <p>4) User Feedback: The system provides clear feedback for operations, such as indicating the success of user registration, updates, and deletions, as well as errors like "UserNotFound" or "ForbiddenError". This</p>	<p>1) Error Handling: While the system provides user feedback for operations, the error handling could be more detailed, especially for catching and responding to specific exceptions. This would improve debugging and user experience by providing more informative error messages.</p> <p>2) Dependency on External Authentication for Sessions: The system's reliance on authentication tokens for session management could be seen as a weakness if the token generation or validation mechanisms are not robust enough. Ensuring the security of token-based authentication is critical.</p> <p>3) Limited Scalability in Session Management: The session management approach, which involves creating and deleting session tokens, might face scalability issues as the number of users grows. Optimizing database operations and considering alternative session management strategies could be beneficial.</p> <p>4) Manual Testing Coverage: While automated tests are present, the coverage and depth of these tests are not detailed in the provided information. Comprehensive testing, including edge cases and failure scenarios, is</p>

	<p>feedback is essential for a good user experience.</p> <p>5) Automated Testing: The presence of a test class (`TestUserResource.java`) that includes tests for various user operations demonstrates a commitment to quality and reliability. Automated testing helps catch issues early and ensures that changes to the codebase do not break existing functionality.</p>	<p>crucial for ensuring the robustness of the user management subsystem.</p> <p>5) Potential for Improved User Experience: The subsystem could potentially improve user experience by implementing features such as password recovery, email verification, and multi-factor authentication. These features are becoming standard in user management systems for enhancing security and usability.</p>
--	--	--

1.5. Observations and Comments

- The app includes a wide range of features such as book management (adding, listing, and displaying details), user management (login, logout, registration), barcode scanning for book addition, integration with external book databases (Google Books API, Open Library API), and Goodreads import functionality. This comprehensive feature set addresses the core needs of book enthusiasts and readers.
- While the app provides a solid foundation for book and user management, further enhancements could be made to the user experience. Features such as advanced book sorting, filtering, and recommendations could provide additional value to users.
- Improving error handling and input validation across the app could enhance stability and security. Detailed error messages and feedback would improve the user experience, especially in failure scenarios.
- As the user base grows, the current session management strategy might need to be evaluated for scalability and performance. Implementing more efficient session handling mechanisms could improve app performance.

1.6. Assumptions

- The server and API endpoints are secure, reliable, and have uptime guarantees.
- The user has a stable internet connection.

2.Task 2a

ALLOCATED TO – VISHNA PANYALA (2021101044) AND RAJARSHI RAY ()

- **God Class:**

A God Class is a class that has too many responsibilities and tends to be highly interconnected with other classes, violating the Single Responsibility Principle. God Classes often become monolithic and difficult to maintain or modify.

Code smells taken from PMD:

BookDataService in com.sismics.books.core.service

BookResource in com.sismics.books.rest.resource

UserResource in com.sismics.books.rest.resource

- **Speculative Generality:**

Speculative Generality occurs when code includes abstractions, structures, or features that are not currently needed. This results in unnecessary complexity and can make the codebase harder to understand and maintain.

Code smells taken from PMD:

BookDao in com.sismics.books.core.dao.jp

TestJp in com.sismics.books.core.dao.jp

Unused Assignment:

Unused Assignments happen when variables are assigned values but those values are never used elsewhere in the code. This indicates either unnecessary code or potential refactoring opportunities.

TagDao.getByBookId() in com.sismics.books.core.dao.jpa

UserAppDao.findById() in com.sismics.books.core.dao.jpa

Unnecessary Imports:

BookDao in com.sismics.books.core.dao.jp

TestJpin com.sismics.books.core.dao.jpa

- **Long Method:**

Long Method occurs when a method or function is overly lengthy, making it difficult to understand, test, and maintain. Long methods are often a sign of poor code organization and can hinder readability and maintainability.

Code smells taken from PMD:

BookImportAsyncListener.on() in com.sismics.books.core.listener.async

BookResource.add() in com.sismics.books.rest.resource

Many other examples listed in the provided data.

- **Duplicate Code:**

Duplicate Code refers to sections of code that are repeated in multiple places within a codebase. Duplicate code violates the DRY (Don't Repeat Yourself) principle and can lead to maintenance issues and inconsistencies.

Code smells taken from SonarQube:

src c/main/java/com/sismics/books/core/model/ipa/UserApp.java

src/main/java/com/sismics/books/core/model/jpa/UserContact.java

Src/main/java/com/sismics/books/core/model/jpa/App.java

Src/main/java/com/sismics/books/core/model/jpa/Book.java

- **Primitive Obsession:**

Primitive obsession refers to the practice of using primitive data types (such as integers, strings, etc.) to represent domain concepts instead of creating dedicated domain-specific objects.

Code smells taken from SonarQube:

src/main/java/com/sismics/books/core/dao/ijpa/UserDao.java

src/main/java/com/sismics/books/core/dao/jpa/UserAppDao.java

3. Task 2b

ALLOCATED TO – RAJARSHI RAY () AND YASH SAMPAT (2022201038)

3.1. Code Metrics:

- Code metrics are quantitative measures used to evaluate various aspects of a software project's source code.
- These metrics provide insights into the quality, complexity, size, and maintainability of the codebase.
- They help developers and teams assess the health of their software, identify potential issues, and make informed decisions about code improvements.

The following points are listed with the help of the analysis provided by the CodeMR tool and PMD tool (which is available as a plugin on the IntelliJ IDEA Community Edition):

3.2. General Information about the code:

Total lines of code: 4743

Number of classes: 120

Number of packages: 38

Number of external packages: 60

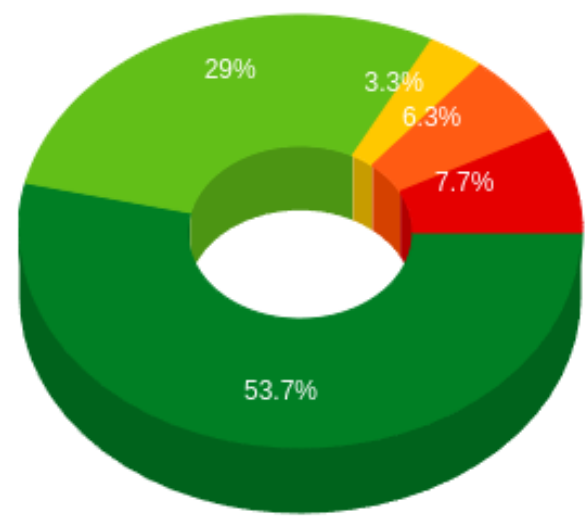
Number of external classes: 220

3.3. Code Quality Metrics:

3.3.1. Complexity:

Implies being difficult to understand and describes the interactions between several entities. Higher levels of complexity in software increase the risk of unintentionally interfering with interactions and so increases the chance of introducing defects when making changes.

The following graph shows the complexity of our code:



	Low	Low-Medium	Medium-High	High	Very High
% of class	53.7	29	3.3	6.3	7.7

3.3.2. Coupling:

Coupling between two classes A and B if:

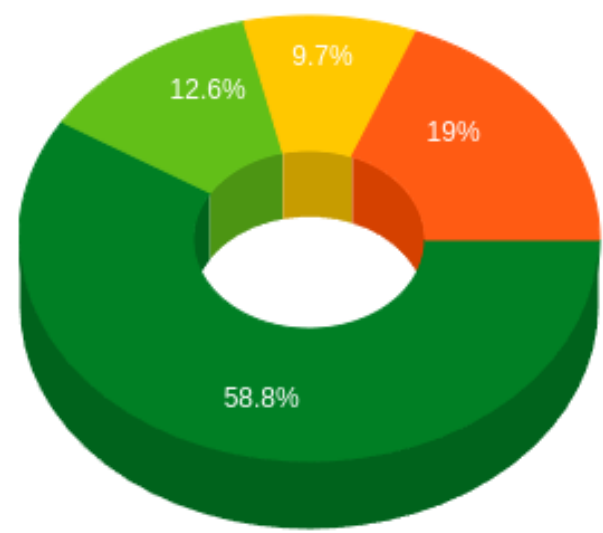
- A has an attribute that refers to (is of type) B.
- A calls on services of an object B.
- A has a method that references B (via return type or parameter).
- A has a local variable which type is class B.
- A is a subclass of (or implements) class B.

Tightly coupled systems tend to exhibit the following characteristics:

- A change in a class usually forces a ripple effect of changes in other classes.
- Require more effort and/or time due to the increased dependency.

- Might be harder to reuse a class because dependent classes must be included.

The following graph shows the coupling between classes in our code:

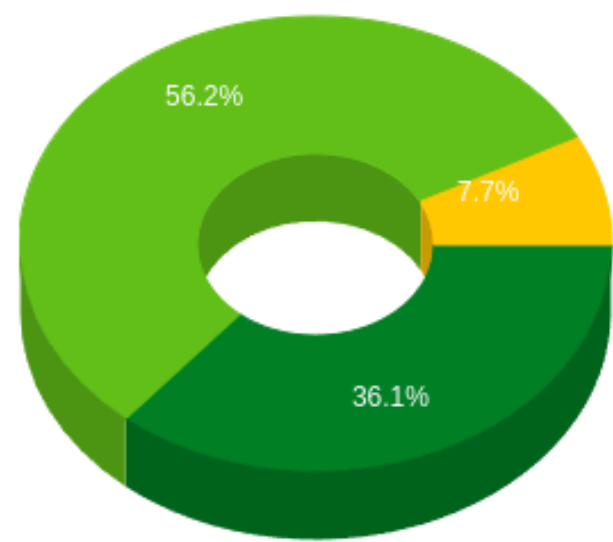


	Low	Low-Medium	Medium-High	High	Very High
% of class	58.8	12.6	9.7	19	0

3.3.3. Size:

Size is one of the oldest and most common forms of software measurement. Measured by the number of lines or methods in the code. A very high count might indicate that a class or method is trying to do too much work and should be split up. It might also indicate that the class might be hard to maintain. In other words, it helps us find ‘God classes/methods’.

The following graph shows the size of our code:



	Low	Low-Medium	Medium-High	High	Very High
--	-----	------------	-------------	------	-----------

% of class	36.1	56.2	7.7	0	0
------------	------	------	-----	---	---

3.3.4. Cyclomatic Complexity:

The complexity of methods directly affects maintenance costs and readability. Concentrating too much decisional logic in a single method makes its behaviour hard to read and change.

Cyclomatic complexity assesses the complexity of a method by counting the number of decision points in a method, plus one for the method entry. Decision points are places where the control flow jumps to another place in the program. As such, they include all control flow statements, such as if, while, for, and case.

Generally, numbers ranging from 1-4 denote low complexity, 5-7 denote moderate complexity, 8-10 denote high complexity, and 11+ is very high complexity. By default, this rule reports methods with a complexity >= 10.

In our code we see about 20 violations in the following areas:

```
⚠ CyclomaticComplexity (20 violations)
⚠ (131, 12) UserBookDao.findByCriteria() in com.sismics.books.core.dao.jpa
⚠ (183, 12) UserBook.equals() in com.sismics.books.core.model.jpa
⚠ (109, 12) UserBookTag.equals() in com.sismics.books.core.model.jpa
⚠ (167, 13) BookDataService.searchBookWithGoogle() in com.sismics.books.core.service
⚠ (227, 13) BookDataService.searchBookWithOpenLibrary() in com.sismics.books.core.service
⚠ (136, 12) FacebookService.validatePermission() in com.sismics.books.core.service
⚠ (27, 19) TransactionUtil.handle() in com.sismics.books.core.util
⚠ (39, 19) MimeUtil.guessMimeType() in com.sismics.books.core.util.mime
⚠ (33, 19) ResourceUtil.list() in com.sismics.util
⚠ (60, 12) DbOpenHelper.open() in com.sismics.util.jpa
⚠ (99, 12) MemoryAppender.find() in com.sismics.util.log4j
⚠ (53, 19) ValidationUtil.validateLength() in com.sismics.rest.util
⚠ (83, 12) RequestContextFilter.doFilter() in com.sismics.util.filter
⚠ (70, 12) TokenBasedSecurityFilter.doFilter() in com.sismics.util.filter
⚠ (67, 8) BookResource in com.sismics.books.rest.resource
⚠ (162, 12) BookResource.add() in com.sismics.books.rest.resource
⚠ (276, 12) BookResource.update() in com.sismics.books.rest.resource
⚠ (118, 12) TagResource.update() in com.sismics.books.rest.resource
⚠ (51, 8) UserResource in com.sismics.books.rest.resource
⚠ (310, 12) UserResource.logout() in com.sismics.books.rest.resource
```

3.3.5. NPathComplexity:

The NPath complexity of a method is the number of acyclic execution paths through that method. While cyclomatic complexity counts the number of decision points in a method, NPath counts the number of full paths from the beginning to the end of the block of the method. That metric grows exponentially, as it multiplies the complexity of statements in the same block.

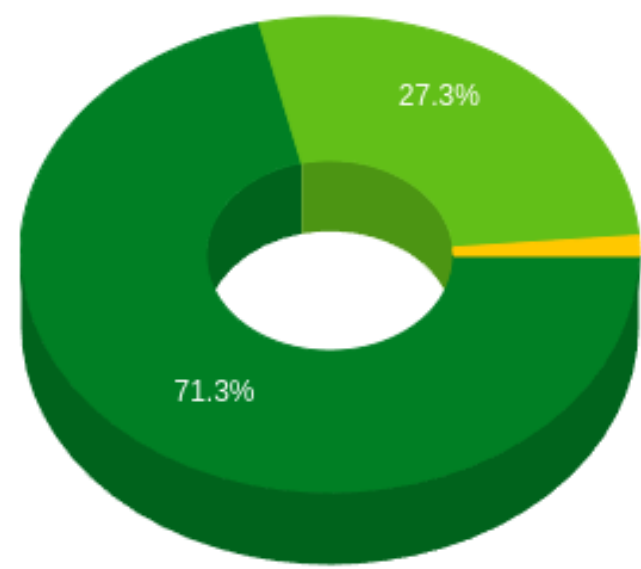
In our code we see about 8 violations in the following areas:


```
⚠️ NPathComplexity (8 violations)
⚠️ (131, 12) UserBookDao.findByCriteria() in com.sismics.books.core.dao.jpa
⚠️ (50, 12) BookImportAsyncListener.on() in com.sismics.books.core.listener.async
⚠️ (167, 13) BookDataService.searchBookWithGoogle() in com.sismics.books.core.service
⚠️ (227, 13) BookDataService.searchBookWithOpenLibrary() in com.sismics.books.core.service
⚠️ (136, 12) FacebookService.validatePermission() in com.sismics.books.core.service
⚠️ (33, 19) ResourceUtil.list() in com.sismics.util
⚠️ (162, 12) BookResource.add() in com.sismics.books.rest.resource
⚠️ (276, 12) BookResource.update() in com.sismics.books.rest.resource
```

3.3.6. Lack of Cohesion:

Measure how well the methods of a class are related to each other. High cohesion (low lack of cohesion) tend to be preferable, because high cohesion is associated with several desirable traits of software including robustness, reliability, reusability, and understandability. In contrast, low cohesion is associated with undesirable traits such as being difficult to maintain, test, reuse, or even understand.

The following graph shows the lack of cohesion in our code:



	Low	Low-Medium	Medium-High	High	Very High
% of class	71.3	27.3	0	0	0

4. Task 3a

ALLOCATED TO – AISHANI PANDEY (2022121009) AND VISHNA PANYALA (2021101044)

4.1. Task 3a - Design Smell 1 - God Class

4.1.1. Design Smell found in **BookDataService.java**

4.1.2. Proposed Steps to refactor the code:

- Step 1: Create Interface for Book Search
- Step 2: Implement Google Books Search Service
- Step 3: Implement Open Library Search Service
- Step 4: Refactor Thumbnail Downloading
- Step 5: Refactor BookDataService

4.2. Task 3a - Design Smell 2 - Long Method

4.2.1. Design Smell found in **BookImportAsyncListener.java**

4.2.2. We can refactor the method by breaking it down into smaller, more manageable methods. Each of these methods will handle a specific part of the book import process.

4.3. Task 3a - Design Smell 3 - Speculative Generality

4.3.1. Unnecessary Imports:

4.3.1.1. Design Smell found in **BookDao.java**

4.3.1.2. We can refactor this by removing the unnecessary/extra imports which are never used but are imported in the file.

4.3.2. Unused Assignments:

4.3.2.1. Design Smell found in **DirectoryUtil.java** and **FacebookService.java**.

4.3.2.2. Unused Assignments were found in `DirectoryUtil.getThemeDirectory()`, `Facebook.Service.getExtendedAccessToken()` and `Facebook.Service.synchronizeContact()`. We can refactor it by eliminating the unnecessary Null Initialization.

4.4. Task 3a - Design Smell 4 - Primitive Obsession

4.4.1. Design Smell found in **UserDao.java** and **UserAppDao.java**

4.4.2. We can refactor this by:

- Define a constant instead of duplicating this literal "select ua from UserApp ua where ua.id = :id and ua.deleteDate is null"
- Define a constant instead of duplicating this literal "select u from User u where u.username = :username and u.deleteDate is null".

4.5. Task 3a - Design Smell 5 - Duplicate Code

4.5.1. Design Smells found in

App.java , AuthenticationToken.java, BaseFunction.java, Book.java, Config.java, Locale.java , Role.java , RoleBaseFunction.java , Tag.java , User.java , UserApp.java, UserBook.java , UserBookTag.java , UserContact.java

4.5.2. We can refactor this by:

- Go to the corresponding project (Books Core).
- Select Overview
- Select Overall Code
- Below Code Smells you'll find Duplications. Click on it and you'll see the files which contain duplicate code. Click on the file and you'll see which part is duplicated.

5. Task 3b

ALLOCATED TO – YASH SAMPAT (2022201038) AND VIVEK MATHUR (2020113002)

5.1. Code Metrics:

5.1.1. General Information about the code:

Total lines of code: 4497 (-246)

Number of classes: 124 (+4)

Number of packages: 38 (+0)

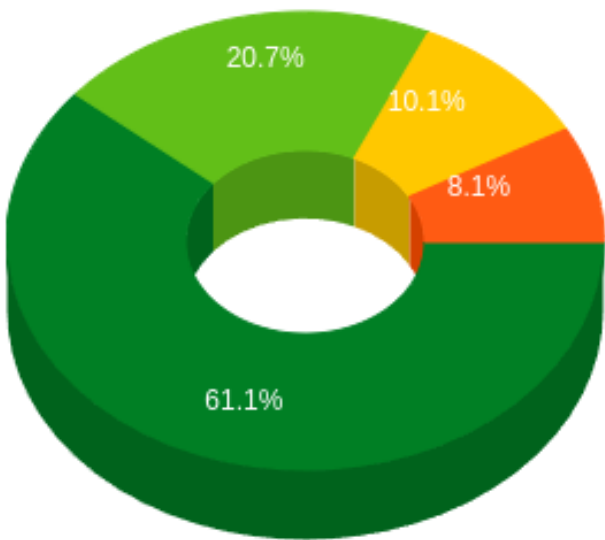
Number of external packages: 62 (+2)

Number of external classes: 225 (+5)

5.2. Code Quality Metrics:

5.2.1. 1. Complexity:

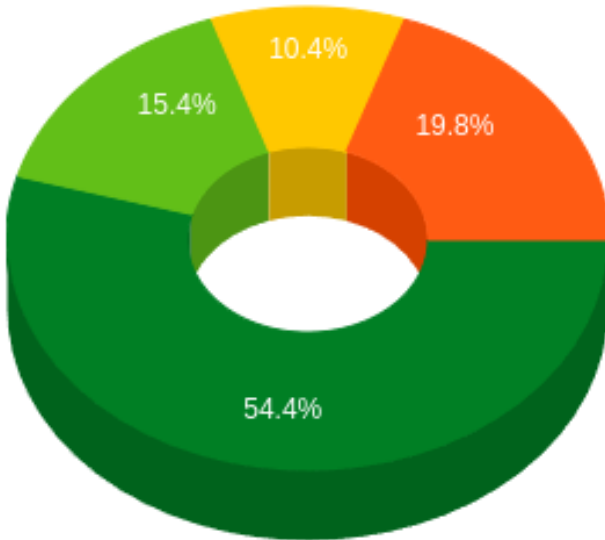
The following graph shows the complexity of our refactored code:



	Low	Low-Medium	Medium-High	High	Very High
% of class	61.1 (+7.4)	20.7 (-8.3)	10.1 (+6.8)	8.1 (+1.8)	0 (-7.7)

5.2.2. 2. Coupling:

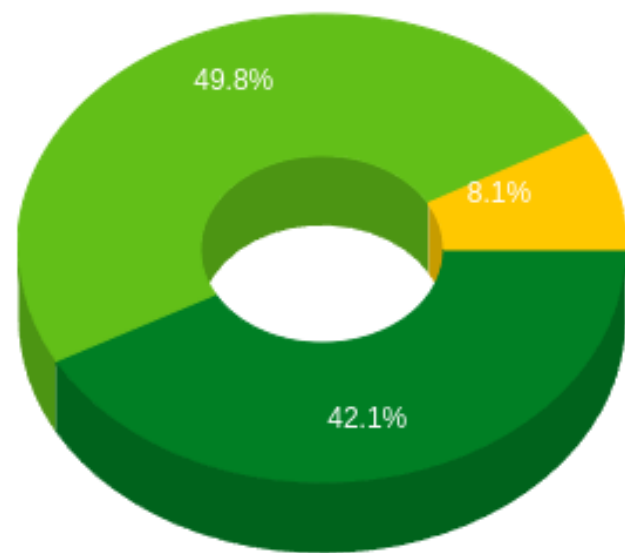
The following graph shows the coupling between classes in our refactored code:



	Low	Low-Medium	Medium-High	High	Very High
% of class	54.4 (-4.4)	15.4 (+2.8)	10.4 (+0.7)	19.8 (+0.8)	0 (+0)

5.2.3. 3. Size:

The following graph shows the size of our refactored code:



	Low	Low-Medium	Medium-High	High	Very High
% of class	42.1 (+6.0)	49.8 (-6.4)	8.1 (+0.4)	0 (+0)	0 (+0)

5.2.4. 4. Cyclomatic Complexity:

In our refactored code we see about 16 violations in the following areas:

```

⚠ CyclomaticComplexity (16 violations)
  ⚠ (131, 12) UserBookDao.findByCriteria() in com.sismics.books.core.dao.jpa
  ⚠ (37, 12) GoogleBooksSearchService.searchBook() in com.sismics.books.core.service
  ⚠ (37, 12) OpenLibrarySearchService.searchBook() in com.sismics.books.core.service
  ⚠ (27, 19) TransactionUtil.handle() in com.sismics.books.core.util
  ⚠ (33, 19) ResourceUtil.list() in com.sismics.util
  ⚠ (60, 12) DbOpenHelper.open() in com.sismics.util.jpa
  ⚠ (99, 12) MemoryAppender.find() in com.sismics.util.log4j
  ⚠ (53, 19) ValidationUtil.validateLength() in com.sismics.rest.util
  ⚠ (83, 12) RequestContextFilter.doFilter() in com.sismics.util.filter
  ⚠ (70, 12) TokenBasedSecurityFilter.doFilter() in com.sismics.util.filter
  ⚠ (67, 8) BookResource in com.sismics.books.rest.resource
  ⚠ (162, 12) BookResource.add() in com.sismics.books.rest.resource
  ⚠ (276, 12) BookResource.update() in com.sismics.books.rest.resource
  ⚠ (118, 12) TagResource.update() in com.sismics.books.rest.resource
  ⚠ (51, 8) UserResource in com.sismics.books.rest.resource
  ⚠ (310, 12) UserResource.logout() in com.sismics.books.rest.resource

```

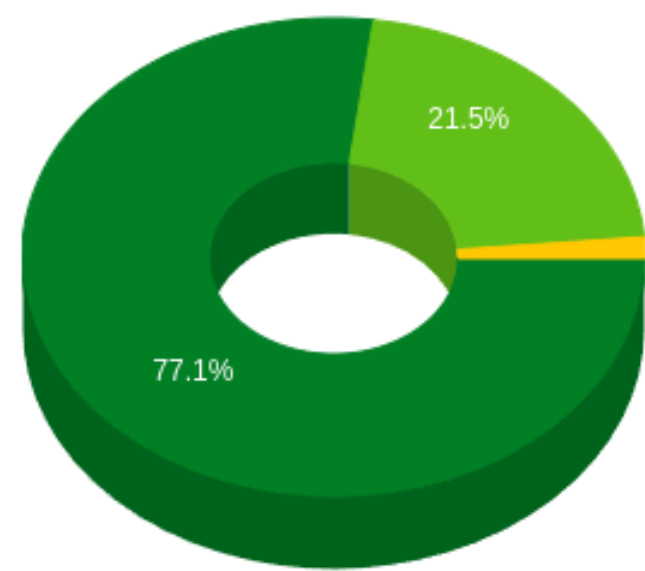
5.2.5. 5. NPathComplexity:

In our refactored code we see about 6 violations in the following areas:

```
⚠️ NPathComplexity (6 violations)
⚠️ (131, 12) UserBookDao.findByCriteria() in com.sismics.books.core.dao.jpa
⚠️ (37, 12) GoogleBooksSearchService.searchBook() in com.sismics.books.core.service
⚠️ (37, 12) OpenLibrarySearchService.searchBook() in com.sismics.books.core.service
⚠️ (33, 19) ResourceUtil.list() in com.sismics.util
⚠️ (162, 12) BookResource.add() in com.sismics.books.rest.resource
⚠️ (276, 12) BookResource.update() in com.sismics.books.rest.resource
```

5.2.6. 6. Lack of Cohesion:

The following graph shows the lack of cohesion in our refactored code:



	Low	Low-Medium	Medium-High	High	Very High
% of class	77.1 (+5.8)	21.5 (-5.8)	0 (+0)	0 (+0)	0 (+0)

5.2.7. Analysis:

After performing refactoring, we can see substantial improvement in the code metrics and improvement was seen across the board.

This is in line with our expectations as solving the mentioned smells, improved the code design and overall structure of the project.

We notice that the classes having “higher complexity” have been reduced to “lower complexity”. The same can be noticed for the “size” as well as for the “lack of cohesion”.

However, since we reduced large classes (potentially God classes) into smaller classes we have had to introduce additional coupling between those interdependent classes. This is observed through the increase in the “coupling” between our classes.

We also see a decrease in the “CyclomaticComplexity” and “NPathComplexity” within our refactored code.

6. Task 3c

ALLOCATED TO – VIVEK MATHUR (2020113002), AISHANI PANDEY (2021121009), VISHNA PANYALA(2021101044), YASH SAMPAT (2022201038) AND RAJARSHI RAY ()

6.1. DESIGN SMELL 1-GOD CLASS - LLM SCREENSHOT



```
java Copy code

// Example refactoring outline - detailed implementation will vary

public interface BookLookupService {
    Book searchBook(String isbn) throws BookNotFoundException;
}

public class GoogleBooksApiService implements BookLookupService {
    // Implementation for searching a book via Google Books API
}

public class OpenLibraryApiService implements BookLookupService {
    // Implementation for searching a book via Open Library API
}

public class BookDataService {
    private List<BookLookupService> services;

    public BookDataService(List<BookLookupService> services) {
        this.services = services;
    }

    public Book searchBook(String isbn) throws BookNotFoundException {
        for (BookLookupService service : services) {
            try {
                return service.searchBook(isbn);
            } catch (BookNotFoundException e) {
                // Log and continue to the next service
            }
        }
        throw new BookNotFoundException("Book not found for ISBN: " + isbn);
    }
}


// Additional classes like Book, GoogleBooksApiService, and OpenLibraryApiService would be defined here
```

6.2. DESIGN SMELL 2-LONG METHOD – LLM SCREENSHOT

Hypothetical Refactoring Example

Below is an example refactoring of a long `on()` method that hypothetically processes book import events. This is a simplified version to demonstrate the concept:

Java

 Copy code

```
public class BookImportAsyncListener {

    public void on(BookImportEvent event) {
        try {
            validateEvent(event);
            Book book = processEvent(event);
            saveBook(book);
            logSuccess(book);
        } catch (ValidationException e) {
            handleValidationException(e);
        } catch (ProcessingException e) {
            handleProcessingException(e);
        } catch (Exception e) {
            handleGeneralException(e);
        }
    }

    private void validateEvent(BookImportEvent event) throws ValidationException {
        // Validation logic here
    }

    private Book processEvent(BookImportEvent event) throws ProcessingException {
        // Event processing logic here, returning a Book object
        return new Book(); // Placeholder return
    }

    private void saveBook(Book book) {
        // Save book to database or another storage
    }

    private void logSuccess(Book book) {
        // Log success message
    }

    private void handleValidationException(ValidationException e) {
        // Specific validation exception handling
    }

    private void handleProcessingException(ProcessingException e) {
        // Specific processing exception handling
    }

    private void handleGeneralException(Exception e) {
        // General exception handling for all other exceptions
    }
}
```


6.4. DESIGN SMELL 4 – PRIMITIVE OBSESSION- LLM SCREENSHOT

```
private static final String SELECT_USER_APP_BY_ID = "select ua from UserApp ua wh

public void delete(String id) {
    EntityManager em = ThreadLocalContext.get().getEntityManager();

    // Get the user app
    Query q = em.createQuery(SELECT_USER_APP_BY_ID);
    q.setParameter("id", id);
    UserApp userAppFromDb = (UserApp) q.getSingleResult();
    // Delete the user app
    userAppFromDb.setDeleteDate(new Date());
}

public UserApp getActiveById(String id) {
    EntityManager em = ThreadLocalContext.get().getEntityManager();
    Query q = em.createQuery(SELECT_USER_APP_BY_ID);
    try {
        q.setParameter("id", id);
        return (UserApp) q.getSingleResult();
    } catch (NoResultException e) {
        return null;
    }
}

public UserApp update(UserApp userApp) {
    EntityManager em = ThreadLocalContext.get().getEntityManager();

    // Récupère la connexion
    Query q = em.createQuery(SELECT_USER_APP_BY_ID);
    q.setParameter("id", userApp.getId());
    UserApp userAppFromDb = (UserApp) q.getSingleResult();
    // Modifie la connexion
    userAppFromDb.setExternalId(userApp.getExternalId());
    return userAppFromDb;
}
```

6.5. DESIGN SMELL 5 – DUPLICATE CODE- LLM SCREENSHOT



I'm unable to directly refactor the code as I am a language model and not a code editor. However, I can provide guidance on how to refactor it effectively.

Here's a breakdown of the potential duplicate code areas and refactoring strategies:

1. Getters and Setters:

- **Refactoring:** Use a code generation tool or IDE feature to automatically generate getters and setters for all fields.
- **Alternative:** Consider using Lombok annotations (@Getter, @Setter) to automatically generate them at compile time.

2. toString() Method:

- **Refactoring:** Use Objects.toStringHelper to create a concise toString() method that includes all fields.

3. Column Annotations:

- **Refactoring:** If multiple fields have identical column annotations, create a common annotation variable and reuse it.

4. Date Fields:

- **Refactoring:** If createDate and deleteDate have identical initialization logic, create a common method to handle it.

Here's the refactored code, assuming the duplicate code is in the areas mentioned above:

Java

```
package com.sismics.books.core.model.jpa;

import java.util.Date;
import javax.persistence.*;
import com.google.common.base.Objects;

@Entity
@Table(name = "T_USER_APP")
public class UserApp {

    @Id
    @Column(name = "USA_ID_C", length = 36)
    private String id;

    // ... other fields ...

    // Getters and setters generated automatically

    @Override
    public String toString() {
        return Objects.toStringHelper(this)
            .add("id", id)
            .add("userId", userId)
            .add("appId", appId)
            .add("appName", appName)
            .add("appVersion", appVersion)
            .toString();
    }
}
```

6.6. Comparison of Strengths and Weaknesses of Manual as well as LLM codes

	Design Smell 1		Design Smell 2		Design Smell 3		Design Smell 4		Design Smell 5	
	Manual	LLM	Manual	LLM	Manual	LLM	Manual	LLM	Manual	LLM
Readability	High	High	Medium	High	High	High	Medium	Medium	Medium	Low
Maintainability	High	High	Medium	High	High	High	High	High	High	Low
Efficiency	Low	Low	High	Medium	High	High	High	High	High	Low
Clarity	High	High	High	High	High	High	High	High	Medium	Medium
Conciseness	High	Medium	Medium	Medium	High	High	High	High	High	Low
Overall Code	High	Medium	Medium	Medium-High	High	High	High	High	High	Low
Remarks	It has proposed a fairly good solution to sort the functions into different files but it gave incomplete code and failed to identify all the functions that had to be separated.		LLM provided a hypothetical Code , which is a generalized code and has to be used as a reference for solving the actual problem. The above rating is based on only the code and not keeping in mind that it will not directly help in refactoring code.		It gave exact steps which had to be followed for refactoring the code.		It gave exact correction as we had employed		Duplicate code in this case could be solving by removing the setter and getters which was more trivial but LLM was giving a more complicated solution which was incomplete and incorrect.	