CS6.401 Software Engineering Project Report

Team 10

February 19, 2024

1 Task 1

1.1 Identify Relevant Classes

1.1.1 Book Addition & Display Subsystem

BaseResource

- Represents a base resource class for RESTful resources related to books.
- Contains fields like request, appKey, and principal for managing HTTP requests and authentication.
- Provides methods for authentication and checking base functions.

BookResource

- Extends the BaseResource class and represents a resource for managing books.
- Handles operations related to books in the system.
- Provides methods for adding, deleting, updating, and retrieving book information.It also includes methods for managing book covers, listing books, importing files, and marking books as read.

Book

- Represents a model class for books stored in the database.
- Contains attributes such as id, title, subtitle, author, description, isbn10, isbn13, pageCount, language, and publishDate.
- Provides getter and setter methods for accessing and updating book attributes.

UserBook

- Represents a model class for user-book relationships stored in the database.
- Includes attributes like id, bookId, userId, createDate, deleteDate, and readDate.
- Provides methods for retrieving and updating user-book relationships.

Tag

- Represents a model class for tags(or bookshelf tag) associated with userBook.
- Contains attributes such as id, name, userId, createDate, deleteDate, and color.
- Provides methods for managing tag attributes.

User

- Represents a model class for users in the system.
- Includes attributes like id, localeId, roleId, username, password, email, theme, firstConnection, createDate, and deleteDate.
- Provides methods for accessing and updating user information.

UserBookDao

- Represents a Data Access Object (DAO) for managing user-book relationships in the database.
- Provides methods for creating, deleting, and retrieving user-book relationships based on various criteria.

BookDao

- Represents a Data Access Object (DAO) for managing book data in the database.
- Provides methods for creating, retrieving, and updating book information.

TagDao

- Represents a Data Access Object (DAO) for managing tag data in the database.
- Provides methods for creating, updating, deleting, and retrieving tag information.

AppContext

- Represents the context of the application containing various resources and services.
- Includes fields for event buses, data services, Facebook services, and executor services.
- Provides methods for accessing and managing application resources.

BookDataService

- Represents a service for handling book-related data operations.
- Includes methods for searching books, downloading thumbnails, and managing book data from external sources like Google Books and Open Library.

FacebookService

- Represents a service for interacting with Facebook API.
- Provides methods for startup, shutdown, validating permissions, synchronizing contacts, updating user data, and publishing actions related to user-book interactions.

IPrincipal

- Represents an interface for managing principal information related to security.
- Provides methods for checking if a user is anonymous, retrieving user ID, locale, date-time zone, and email.

ValidationUtil

- Represents a utility class for performing various validation checks.
- Includes methods for validating email, HTTP URL, alphanumeric strings, length constraints, date, locale, and theme.

PaginatedList

- Represents a generic paginated list for storing and managing lists of items.
- Includes attributes like limit, offset, resultCount, and resultList.
- Provides methods for managing paginated lists and retrieving results.

PaginatedLists

- Represents a utility class for managing paginated lists.
- Provides methods for creating paginated lists, executing count and result queries, and executing paginated queries.

SortCriteria

- Represents criteria for sorting paginated lists.
- Includes attributes like column and asc to specify the sorting column and order.
- Provides methods for accessing sort criteria attributes.

UserBookCriteria

- Criteria object used to filter user-book relationships based on specific conditions.
- Includes attributes like userId, search, read, and tagIdList to refine search results.
- Provides getter and setter methods for each criterion.

BookImportedEvent

- Represents an event triggered when a book is imported into the system.
- Contains details about the user who imported the book and the file that was imported.
- Provides methods to retrieve and set user and file information and generate string representations.

1.1.2 Bookshelf Management Subsystem

TagResource

- Manages operations related to tags within the Bookshelf or Tag Management Subsystem.
- Provides functionalities for listing, adding, updating, and deleting tags.
- Utilizes TagDao for data access and ValidationUtil for input validation.

IPrincipal

- Interface defining methods for user principal information within the Bookshelf or Tag Management Subsystem.
- Contains methods like isAnonymous(), getId(), getLocale(), getDateTimeZone(), and getEmail().

UserPrincipal

- Represents user principal information within the Bookshelf or Tag Management Subsystem.
- Provides methods for accessing and setting user principal attributes.

TagDao

- Manages data access operations related to tags within the Bookshelf or Tag Management Subsystem.
- Provides methods for retrieving, updating, creating, and deleting tags.
- Also includes methods for fetching tags by user ID, userBook ID, and name.

Tag

- Represents tags associated with userBooks within the Bookshelf or Tag Management Subsystem.
- Provides methods for accessing and modifying tag attributes.

UserBookTag

- Represents the relationship between userBooks and tags within the Bookshelf or Tag Management Subsystem.
- Provides methods for accessing and setting userBookTag attributes.

User

- Represents user information within the Bookshelf or Tag Management Subsystem.
- Provides methods for accessing and updating user attributes.

Book

- Represents book information within the Bookshelf or Tag Management Subsystem.
- Provides methods for accessing and updating book attributes.

UserBook

- Represents the relationship between users and books within the Bookshelf or Tag Management Subsystem.
- Provides methods for accessing and updating userBook relationships.

ValidationUtil

- Provides utility methods for input validation within the Bookshelf or Tag Management Subsystem.
- Includes methods for validating strings, email addresses, URLs, colors, dates, locales, and themes.

ThreadLocalContext

- Manages the thread-local context for entity managers within the Bookshelf or Tag Management Subsystem.
- Provides methods for retrieving the thread-local context, cleaning up the context, checking transactional context, and managing the entity manager.

TagDto

- Represents data transfer objects for tags within the Bookshelf or Tag Management Subsystem.
- Provides methods for accessing and setting tagDto attributes.

1.1.3 User Management Subsystem

Principal Interface

- Represents a principal interface.
- Sub-interface `IPrincipal` extends `Principal` and includes methods for managing user authentication details.

UserPrincipal Class

- Implements `IPrincipal` interface.
- Manages user authentication details such as id, name, locale, email, and base function set.
- Provides methods for accessing and updating user details.

WebApplicationException Classes

- Represents exceptions for web applications.
- `ClientException`, `ForbiddenClientException`, and `ServerException` are specialized exceptions.
- They include methods for handling different types of exceptions.

ValidationUtil Class

- Provides utility methods for validation.
- Static methods include validation for required fields and length constraints.

TokenBasedSecurityFilter Class

- Implements the `Filter` interface.
- Manages token-based security for authentication.
- Includes methods for initializing, destroying, and filtering requests based on token authentication.

BaseFunction Enum

- Represents base functions related to user roles, specifically the 'ADMIN' function.

BaseResource Class

- Represents an abstract base resource class.
- Contains fields for managing HTTP requests and authentication, such as `request`, `appKey`, and `principal`.
- Provides methods for authentication and checking base functions.

UserResource Class

- Extends `BaseResource` class and represents a resource for managing users.
- Handles operations related to user registration, login, logout, updating user details, and more.
- Includes methods for various user-related actions.

Constants Class

- Contains constants related to the application, such as default locale, timezone, theme, admin password, and user role.

AuthenticationToken and User Classes

- Model classes representing authentication tokens and users stored in the database.
- Include attributes like id, userId, creation date, and additional user details.
- Provide getter and setter methods for accessing and updating attributes.

PaginatedList, PaginatedLists, and SortCriteria Classes

- Utility classes for handling paginated lists, sorting criteria, and executing queries.
- Support pagination and sorting operations.

AuthenticationTokenDao, RoleBaseFunctionDao, and UserDao Classes

- DAO (Data Access Object) classes for handling database operations related to authentication tokens, role-base functions, and users.
- Include methods for creating, updating, deleting, and retrieving entities from the database.

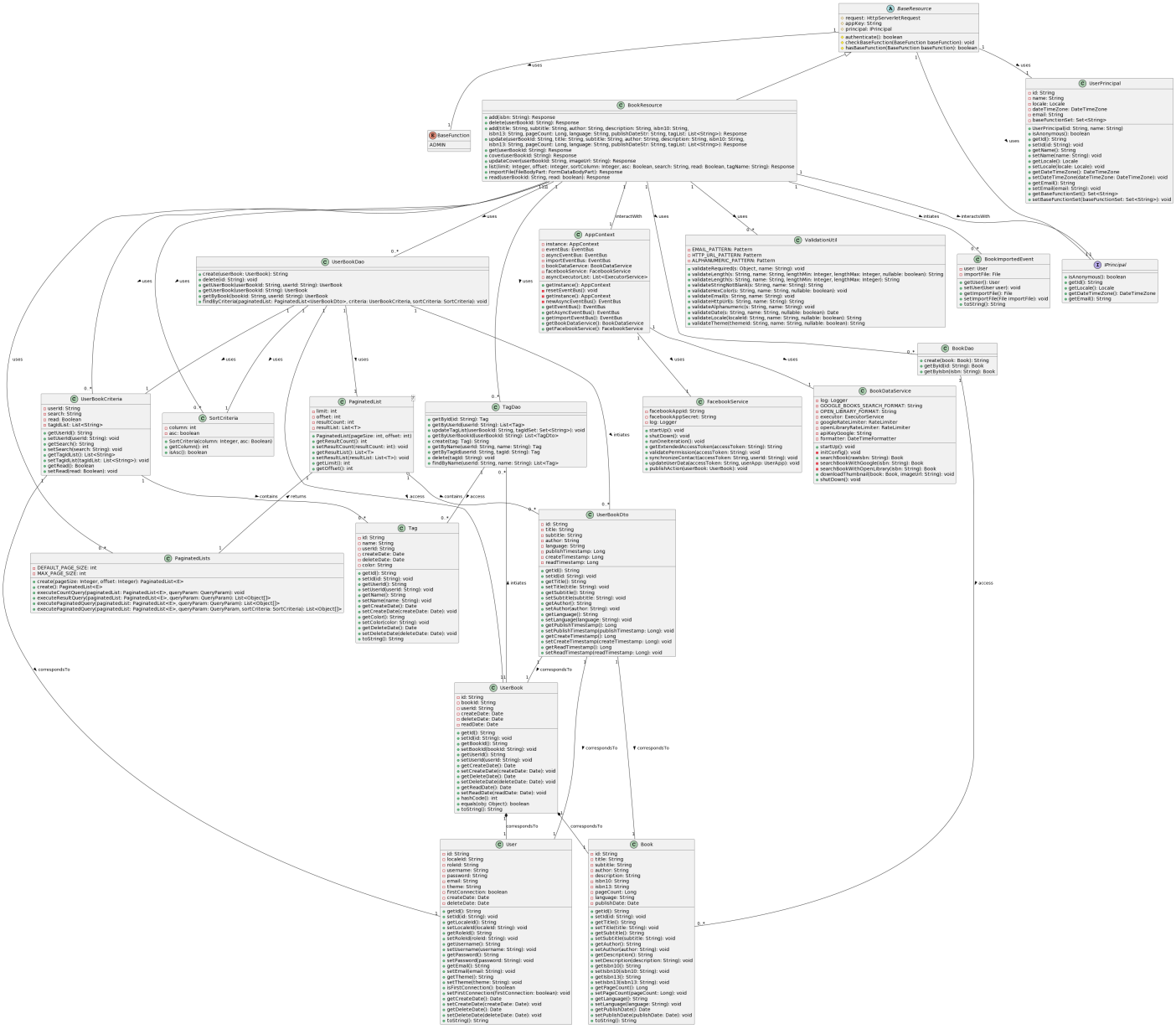
UserDto Class

- Data transfer object representing a simplified view of a user for paginated lists.
- Includes attributes like id, localeId, username, email, and create timestamp.

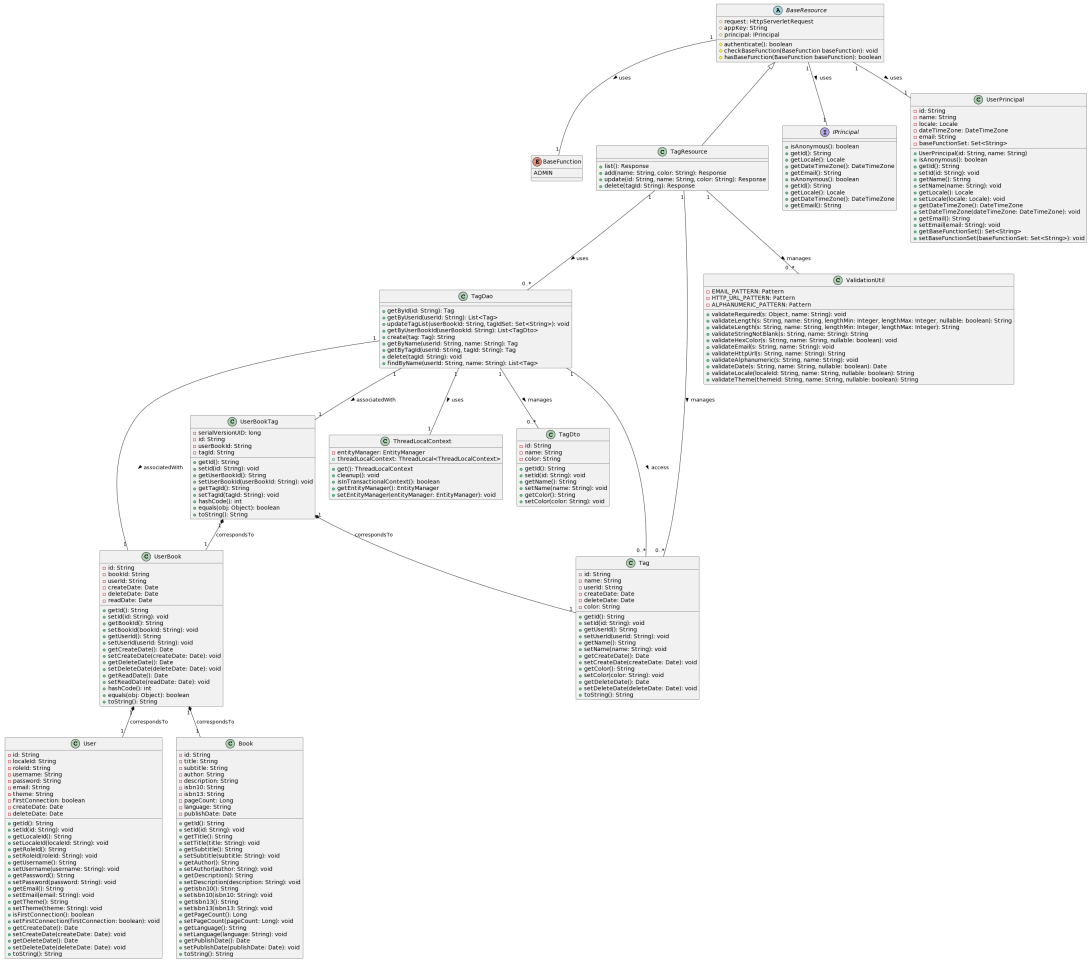
1.2 Create UML Diagrams

The UML images can be found in the *docs/Task1/Images* folder in the repository. Refer to the SVG images present in this folder.

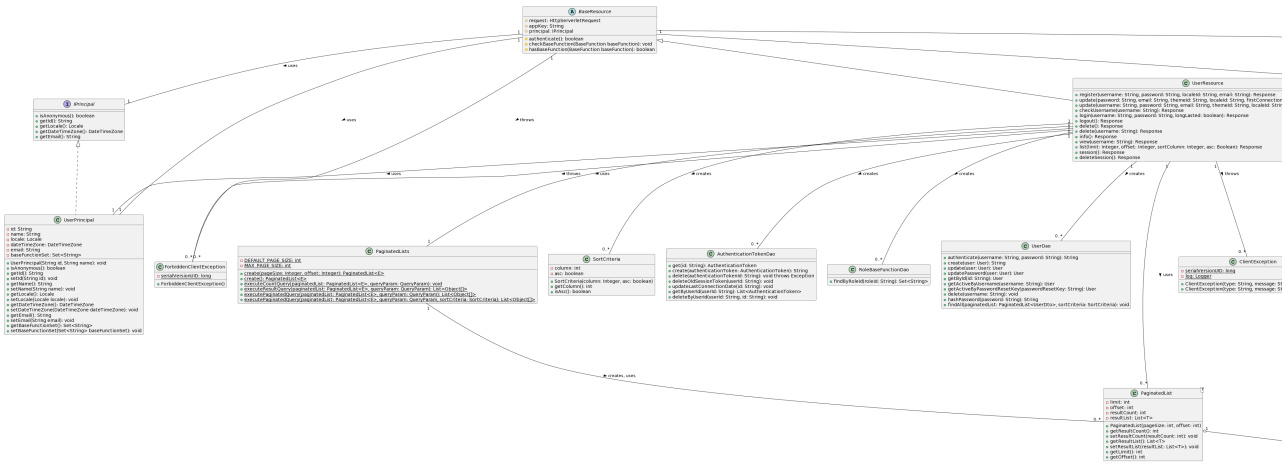
1.2.1 Book Addition & Display Subsystem



1.2.2 Bookshelf Management Subsystem



1.2.3 User Management Subsystem



1.3 Observations, Assumptions and Comments

- **Observations and Comments**: For generating the UML diagrams for the 3 subsystems, we mainly utilized 5 java classes within the *books/rest/resource* directory of the *books-web* folder - AppResource, BaseResource, BookResource, TagResource and UserResource. The BookResource, TagResource and UserResource classes were the primary classes for the Book Addition & Display, Bookshelf Management and User Management subsystems, respectively. We also observed that the abstract class *BaseResource* was the parent class for these 3 primary classes.
- **Assumptions**: For simplifying the UML diagrams, we have removed the packages and inheritance relationships for in-build java classes like `WebApplicationException`. We have also not included less important dependencies (not very relevant to the subsystem) in the UML for each subsystem.

2 Task 2

2.1 Identifying Design Smells

2.1.1 Leaky Encapsulation

Defining Constant instead of Duplicating Literals

If a specific string needs to be changed or updated in the future (for example, if the naming convention changes), it would be necessary to find and update all occurrences of it in the codebase. This increases the maintenance burden and the likelihood of introducing errors.

Closing opened files

When dealing with resource intensive tasks, it is essential to explicitly release resources once they're no longer needed because failure to do so can lead to various problems such as,

- Resource Leakage
- Memory Leaks
- Performance Degradation
- Security Vulnerabilities

2.1.2 Missing Encapsulation

Hide type specification within diamond operator ("<>")

This change enhances maintainability by reducing redundancy in the code. If the type of the list needs to be changed later, you only need to modify it in one place (the declaration of myList) rather than in both the declaration and instantiation. Additionally, it improves readability by removing unnecessary clutter, making the code easier to understand for other developers who may need to work on it in the future.

2.1.3 Modularization

Brain Method (Reduce Lines Of Code)

A "Brain Method" refers to a section of code within a software program that is overly complex or convoluted, making it difficult to understand, maintain, or debug. This issue is classified under maintainability because it affects the ease with which the codebase can be maintained and updated over time. The reasons why Brain Method is considered detrimental to maintainability is:

- Complexity
- Readability

- Debugging
- Scalability

Remove unnecessary nullcheck

In Java, the instanceof operator is used to check whether an object is an instance of a particular class or interface. When instanceof is applied to a null reference, it always returns false because null is not an instance of any class. Here is why removing unnecessary null checks related to instanceof contributes to better maintainability:

- Clarity and Readability
- Reduced Risk of Bugs
- Simplified Logic
- Consistency
- Performance

2.1.4 Incomplete Abstraction

Extract method isEmpty()

Here is why using isEmpty() is better for maintainability.

- Readability and Understandability
- Reduced Complexity
- Encapsulation of Logic
- Consistency and Standardization
- Ease of Maintenance

2.1.5 Unnecessary Abstraction

Use specific Exception instead of general

When troubleshooting or maintaining software, it's crucial to have clear and informative error messages. Using dedicated exceptions with descriptive names and informative error messages makes it easier for developers to understand what went wrong when an error occurs. This improves the maintainability of the codebase because:

- Readability
- Debugging
- Modifiability

2.2 Code Metrics

For the current evaluation, we selected CodeMR (https://www.codemr.co.uk/) as our primary tool for extracting essential code metrics, chosen for its proven reliability and comprehensive analysis capabilities. Our analysis focused on six key metrics: (1) Complexity, (2) Size, (3) Coupling, (4) Lack of Cohesion (LCoh), (5) Response for a Class (RFC), and (6) Coupling Between Object Classes (CBO). The precise definition of complexity used by CodeMR remains uncertain to us.

2.2.1 Metric Definitions

- **Complexity** refers to the level of difficulty in understanding or modifying the software, with a direct correlation to the potential for introducing errors during updates due to intricate interactions among components.
- **Size**, a traditional metric, is quantified by counting lines of code or methods, where a higher count could signify a class or method is overly burdensome, suggesting a need for division or indicating potential maintainability challenges.
- **Coupling** describes the degree of interdependence between classes, identified through various indicators like attribute references, service calls, method relationships, and subclassing. Systems with high coupling often face challenges with changes affecting multiple classes, increased maintenance effort, and difficulties in class reuse.
- **Lack of Cohesion (LCoh)** evaluates the relatedness of a class's methods. Classes exhibiting high cohesion (or low lack of cohesion) are associated with robustness, reliability, reusability, and clarity, whereas low cohesion indicates potential issues with maintenance, testing, reuse, and comprehension.
- **Response For a Class (RFC)** counts the potential methods invoked in reaction to a message received by an object of a class, including the complete call graph from any called method. A high RFC value suggests greater complexity and potential coupling with other classes, necessitating extensive testing and maintenance.
- **Coupling Between Object Classes (CBO)** measures a class's direct ties to other classes, excluding inheritance relationships, by tallying classes used by or using the class in question. Higher CBO values point to challenges in maintainability, reusability, and testing due to the intricate dependencies on other classes.

This analysis aimed to provide a clear picture of the project's structural integrity, highlighting areas that may benefit from refactoring to improve maintainability, reliability, and performance (see Figure 1).

Classes with High Coupling, High Complexity, Low Cohesion: None of the classes fall into this category, indicating that there isn't a convergence of all three negative attributes in any single class. This is a positive sign for the maintainability and understandability of the codebase.

Classes with High Coupling, High Complexity: *BookResource* and *UserResource* are identified with high complexity and high coupling. These classes likely perform a multitude of tasks, interacting heavily with multiple other classes. This can make them difficult to maintain and understand. Refactoring to reduce complexity and coupling can improve maintainability and potential performance.

Classes with High Coupling: *ConnectResource* and *FacebookService* display high coupling, suggesting a strong interdependency with other classes. This can lead to difficulties when changes are made in one class that ripple through to others. Strategies to reduce coupling could include introducing interfaces or abstract classes, thereby promoting looser coupling and increased module independence.

Response For a Class (RFC) and Coupling Between Object Classes (CBO): The *BookResource* class has a notably high RFC, indicating it could be responding to many other classes, thereby increasing its complexity. The *UserResource* class also has a high RFC, suggesting similar complexity and a need for extensive testing and maintenance. Both *BookResource* and *UserResource* have a relatively high CBO, further implying that changes in other classes could necessitate changes within these classes, impacting maintainability and reusability.

General Observations:

1. The majority of classes (116 out of 120) do not exhibit problematic levels of complexity, coupling, or cohesion, which is an excellent indicator of a well-structured codebase.

Figure 1: A screenshot of CodeMR's dashboard. This is a visual representation for high-level code metric analyses.

2. The classes that do show higher metrics should be the focus of refactoring efforts. The target would be to simplify complex interactions, reduce coupling to minimize the impact of changes, and improve cohesion so that classes have a single, well-defined responsibility.

In conclusion, the codebase appears to be in good health, with only a few classes requiring attention. We focuse refactoring efforts on the identified classes to significantly improve the overall quality and maintainability of the project.

3 Task 3

3.1 Code Metrics: Refactored Code

We observe a significant improvement across the code metrics post refactoring (see Figure 2).

Improvements in Coupling, Complexity, and Cohesion: We observe that there are no classes with high coupling, high complexity, or low cohesion. This is a stark contrast to the initial assessment, where certain classes were flagged for high complexity and coupling. The absence of such classes post-refactoring suggests successful efforts to decouple classes and simplify their interactions.

Coupling: In the initial assessment, *BookResource* and *UserResource* displayed high coupling, which has been addressed as shown in the refactored codebase. The coupling for all classes listed is now low. This change would likely make the classes more maintainable and promote easier testing and reuse.

Complexity: The complexity of the classes has been reduced uniformly. The initial assessment indicated *BookResource* and *UserResource* as highly complex, which is no longer the case in the refactored codebase. This reduction in complexity can lead to a decrease in the likelihood of defects, easier maintenance, and potentially better performance.

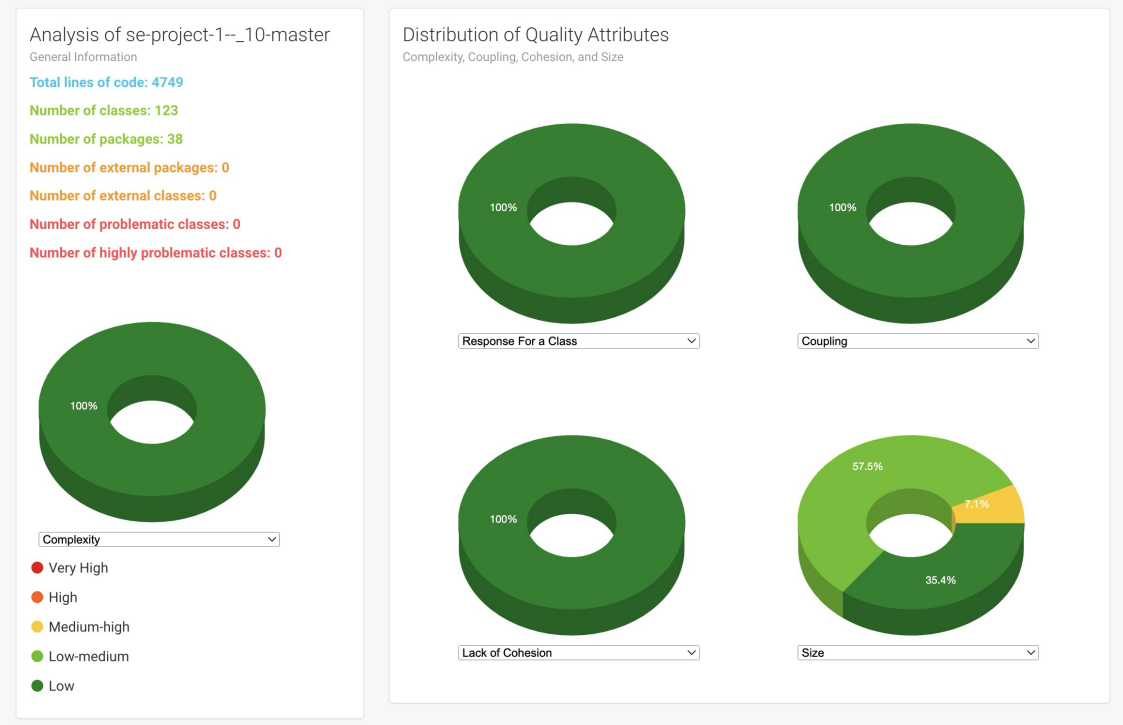


Figure 2: A screenshot of CodeMR's dashboard for the refactored codebase. We observe that all problematic classes have been resolved.

Size (LOC): There was a noticeable reduction in lines of code (LOC) for *BookResource*, dropping from 366 to 338 LOC. A reduction in LOC is typically associated with the elimination of redundant code and improvements in code efficiency, which aligns with the objectives of refactoring.

Cohesion (LCoh): Since there are no classes with low cohesion in the post-refactoring analysis, it can be inferred that the cohesiveness of classes has been improved. This would enhance the single-responsibility principle within the classes, making them more focused on their tasks, easier to understand, and less error-prone.

General Trends and Factors Contributing to Changes: The trends across all metrics indicate a consistent improvement, which is likely the result of targeted refactoring strategies aimed at reducing complexity and coupling while enhancing cohesion. The use of design patterns, modularization, and code simplification techniques are potential factors that contributed to these positive outcomes.

In conclusion, the refactoring efforts have resulted in a codebase that exhibits lower complexity and coupling, as well as improved cohesion. These improvements are conducive to a more robust, maintainable, and scalable codebase. The fact that these positive changes are consistent across the board suggests that the refactoring was comprehensive and addressed the key areas identified in the initial assessment.

3.2 LLM Output Analysis

3.2.1 Brain Method

Brain method refers to a method within a class that exhibits characteristics suggesting it performs too many tasks and lacks proper organization. These methods often tend to be lengthy with high Lines Of Code thereby increasing code complexity. These methods were further modularized as part of the refactoring process. The refactored code solves the *Insufficient Modularization* design smell.

Initial Code

```
/**
 * Add a book book manually.
 *
 * @param title Title
 * @param description Description
 * @return Response
 * @throws JSONException
 */
@PUT
@Path("manual")
@Produces(MediaType.APPLICATION_JSON)
public Response add(
        @FormParam("title") String title,
        @FormParam("subtitle") String subtitle,
        @FormParam("author") String author,
        @FormParam("description") String description,
        @FormParam("isbn10") String isbn10,
        @FormParam("isbn13") String isbn13,
        @FormParam("page_count") Long pageCount,
        @FormParam("language") String language,
        @FormParam("publish_date") String publishDateStr,
        @FormParam("tags") List<String> tagList) throws JSONException {
    if (!authenticate()) {
        throw new ForbiddenClientException();
    }

    // Validate input data
    title = ValidationUtil.validateLength(title, "title", 1, 255, false);
    subtitle = ValidationUtil.validateLength(subtitle, "subtitle", 1, 255, true);
    author = ValidationUtil.validateLength(author, "author", 1, 255, false);
    description = ValidationUtil.validateLength(description, "description", 1, 4000,
        true);
    isbn10 = ValidationUtil.validateLength(isbn10, "isbn10", 10, 10, true);
    isbn13 = ValidationUtil.validateLength(isbn13, "isbn13", 13, 13, true);
    language = ValidationUtil.validateLength(language, "language", 2, 2, true);
    Date publishDate = ValidationUtil.validateDate(publishDateStr, "publish_date",
        false);

    if (Strings.isNullOrEmpty(isbn10) && Strings.isNullOrEmpty(isbn13)) {
        throw new ClientException("ValidationError", "At least one ISBN number is
            mandatory");
    }

    // Check if this book is not already in database
    BookDao bookDao = new BookDao();
    Book bookIsbn10 = bookDao.getByIsbn(isbn10);
    Book bookIsbn13 = bookDao.getByIsbn(isbn13);
    if (bookIsbn10 != null || bookIsbn13 != null) {
        throw new ClientException("BookAlreadyAdded", "Book already added");
    }

    // Create the book
    Book book = new Book();
    book.setId(UUID.randomUUID().toString());

    if (title != null) {
        book.setTitle(title);
```

```
55        }
56        if (subtitle != null) {
57            book.setSubtitle(subtitle);
58        }
59        if (author != null) {
60            book.setAuthor(author);
61        }
62        if (description != null) {
63            book.setDescription(description);
64        }
65        if (isbn10 != null) {
66            book.setIsbn10(isbn10);
67        }
68        if (isbn13 != null) {
69            book.setIsbn13(isbn13);
70        }
71        if (pageCount != null) {
72            book.setPageCount(pageCount);
73        }
74        if (language != null) {
75            book.setLanguage(language);
76        }
77        if (publishDate != null) {
78            book.setPublishDate(publishDate);
79        }
80
81        bookDao.create(book);
82
83        // Create the user book
84        UserBookDao userBookDao = new UserBookDao();
85        UserBook userBook = new UserBook();
86        userBook.setUserId(principal.getId());
87        userBook.setBookId(book.getId());
88        userBook.setCreateDate(new Date());
89        userBookDao.create(userBook);
90
91        // Update tags
92        if (tagList != null) {
93            TagDao tagDao = new TagDao();
94            Set<String> tagSet = new HashSet<>();
95            Set<String> tagIdSet = new HashSet<>();
96            List<Tag> tagDbList = tagDao.getByUserId(principal.getId());
97            for (Tag tagDb : tagDbList) {
98                tagIdSet.add(tagDb.getId());
99            }
100           for (String tagId : tagList) {
101               if (!tagIdSet.contains(tagId)) {
102                   throw new ClientException("TagNotFound", MessageFormat.format("Tag not
                          found: {0}", tagId));
103               }
104               tagSet.add(tagId);
105           }
106           tagDao.updateTagList(userBook.getId(), tagSet);
107       }
108
109       // Returns the book ID
110       JSONObject response = new JSONObject();
111       response.put("id", userBook.getId());
112       return Response.ok().entity(response).build();
```

```
113 }
114
115 /**
116  * Updates the book.
117  *
118  * @param title Title
119  * @param description Description
120  * @return Response
121  * @throws JSONException
122  */
123 @POST
124 @Path("{id: [a-z0-9\\-]+}")
125 @Produces(MediaType.APPLICATION_JSON)
126 public Response update(
127         @PathParam("id") String userBookId,
128         @FormParam("title") String title,
129         @FormParam("subtitle") String subtitle,
130         @FormParam("author") String author,
131         @FormParam("description") String description,
132         @FormParam("isbn10") String isbn10,
133         @FormParam("isbn13") String isbn13,
134         @FormParam("page_count") Long pageCount,
135         @FormParam("language") String language,
136         @FormParam("publish_date") String publishDateStr,
137         @FormParam("tags") List<String> tagList) throws JSONException {
138     if (!authenticate()) {
139         throw new ForbiddenClientException();
140     }
141
142     // Validate input data
143     title = ValidationUtil.validateLength(title, "title", 1, 255, true);
144     subtitle = ValidationUtil.validateLength(subtitle, "subtitle", 1, 255, true);
145     author = ValidationUtil.validateLength(author, "author", 1, 255, true);
146     description = ValidationUtil.validateLength(description, "description", 1, 4000,
        true);
147     isbn10 = ValidationUtil.validateLength(isbn10, "isbn10", 10, 10, true);
148     isbn13 = ValidationUtil.validateLength(isbn13, "isbn13", 13, 13, true);
149     language = ValidationUtil.validateLength(language, "language", 2, 2, true);
150     Date publishDate = ValidationUtil.validateDate(publishDateStr, "publish_date",
        true);
151
152     // Get the user book
153     UserBookDao userBookDao = new UserBookDao();
154     BookDao bookDao = new BookDao();
155     UserBook userBook = userBookDao.getUserBook(userBookId, principal.getId());
156     if (userBook == null) {
157         throw new ClientException("BookNotFound", "Book not found with id " +
            userBookId);
158     }
159
160     // Get the book
161     Book book = bookDao.getById(userBook.getBookId());
162
163     // Check that new ISBN number are not already in database
164     if (!Strings.isNullOrEmpty(isbn10) && book.getIsbn10() != null && !book.getIsbn10
        ().equals(isbn10)) {
165         Book bookIsbn10 = bookDao.getByIsbn(isbn10);
166         if (bookIsbn10 != null) {
167             throw new ClientException("BookAlreadyAdded", "Book already added");
```

```
168             }
169         }
170 
171         if (!Strings.isNullOrEmpty(isbn13) && book.getIsbn13() != null && !book.getIsbn13
                ().equals(isbn13)) {
172             Book bookIsbn13 = bookDao.getByIsbn(isbn13);
173             if (bookIsbn13 != null) {
174                 throw new ClientException("BookAlreadyAdded", "Book already added");
175             }
176         }
177 
178         // Update the book
179         if (title != null) {
180             book.setTitle(title);
181         }
182         if (subtitle != null) {
183             book.setSubtitle(subtitle);
184         }
185         if (author != null) {
186             book.setAuthor(author);
187         }
188         if (description != null) {
189             book.setDescription(description);
190         }
191         if (isbn10 != null) {
192             book.setIsbn10(isbn10);
193         }
194         if (isbn13 != null) {
195             book.setIsbn13(isbn13);
196         }
197         if (pageCount != null) {
198             book.setPageCount(pageCount);
199         }
200         if (language != null) {
201             book.setLanguage(language);
202         }
203         if (publishDate != null) {
204             book.setPublishDate(publishDate);
205         }
206 
207         // Update tags
208         if (tagList != null) {
209             TagDao tagDao = new TagDao();
210             Set<String> tagSet = new HashSet<>();
211             Set<String> tagIdSet = new HashSet<>();
212             List<Tag> tagDbList = tagDao.getByUserId(principal.getId());
213             for (Tag tagDb : tagDbList) {
214                 tagIdSet.add(tagDb.getId());
215             }
216             for (String tagId : tagList) {
217                 if (!tagIdSet.contains(tagId)) {
218                     throw new ClientException("TagNotFound", MessageFormat.format("Tag not
                            found: {0}", tagId));
219                 }
220                 tagSet.add(tagId);
221             }
222             tagDao.updateTagList(userBookId, tagSet);
223         }
224 
```

```
    // Returns the book ID
    JSONObject response = new JSONObject();
    response.put("id", userBookId);
    return Response.ok().entity(response).build();
}
```

Manual Change

```
// Validate input data
private Date validateInput(String title, String subtitle, String author, String
    description, String isbn10, String isbn13,
    String language, String publishDateStr) throws JSONException{
    ValidationUtil.validateLength(title, "title", 1, 255, false);
    ValidationUtil.validateLength(subtitle, "subtitle", 1, 255, true);
    ValidationUtil.validateLength(author, "author", 1, 255, false);
    ValidationUtil.validateLength(description, "description", 1, 4000, true);
    ValidationUtil.validateLength(isbn10, "isbn10", 10, 10, true);
    ValidationUtil.validateLength(isbn13, "isbn13", 13, 13, true);
    ValidationUtil.validateLength(language, "language", 2, 2, true);
    Date publishDate = ValidationUtil.validateDate(publishDateStr, "publish_date",
        false);

    if (Strings.isNullOrEmpty(isbn10) && Strings.isNullOrEmpty(isbn13)) {
        throw new ClientException("ValidationError", "At least one ISBN number is
            mandatory");
    }

    return publishDate;
}

// Create and set the values of the book
private void setBook(Book book, UserBook userBook, String title, String subtitle,
    String author, String description, String isbn10, String isbn13,
    Long pageCount, String language, Date publishDate, List<String> tagList) throws
        JSONException {
    if (title != null) {
        book.setTitle(title);
    }
    if (subtitle != null) {
        book.setSubtitle(subtitle);
    }
    if (author != null) {
        book.setAuthor(author);
    }
    if (description != null) {
        book.setDescription(description);
    }
    if (isbn10 != null) {
        book.setIsbn10(isbn10);
    }
    if (isbn13 != null) {
        book.setIsbn13(isbn13);
    }
    if (pageCount != null) {
        book.setPageCount(pageCount);
    }
    if (language != null) {
```

```
45          book.setLanguage(language);
46      }
47      if (publishDate != null) {
48          book.setPublishDate(publishDate);
49      }
50      if (tagList != null) {
51          TagDao tagDao = new TagDao();
52          Set<String> tagSet = new HashSet<>();
53          Set<String> tagIdSet = new HashSet<>();
54          List<Tag> tagDbList = tagDao.getByUserId(principal.getId());
55          for (Tag tagDb : tagDbList) {
56              tagIdSet.add(tagDb.getId());
57          }
58          for (String tagId : tagList) {
59              if (!tagIdSet.contains(tagId)) {
60                  throw new ClientException("TagNotFound", MessageFormat.format("Tag not
                        found: {0}", tagId));
61              }
62              tagSet.add(tagId);
63          }
64          tagDao.updateTagList(userBook.getId(), tagSet);
65      }
66  }
67
68  /**
69   * Add a book book manually.
70   *
71   * @param title Title
72   * @param description Description
73   * @return Response
74   * @throws JSONException
75   */
76  @PUT
77  @Path("manual")
78  @Produces(MediaType.APPLICATION_JSON)
79  public Response add(
80          @FormParam("title") String title,
81          @FormParam("subtitle") String subtitle,
82          @FormParam("author") String author,
83          @FormParam("description") String description,
84          @FormParam("isbn10") String isbn10,
85          @FormParam("isbn13") String isbn13,
86          @FormParam("page_count") Long pageCount,
87          @FormParam("language") String language,
88          @FormParam("publish_date") String publishDateStr,
89          @FormParam("tags") List<String> tagList) throws JSONException {
90      if (!authenticate()) {
91          throw new ForbiddenClientException();
92      }
93
94      Date publishDate = validateInput(title, subtitle, author, description, isbn10,
            isbn13, language, publishDateStr);
95
96      // Check if this book is not already in database
97      BookDao bookDao = new BookDao();
98      Book bookIsbn10 = bookDao.getByIsbn(isbn10);
99      Book bookIsbn13 = bookDao.getByIsbn(isbn13);
100     if (bookIsbn10 != null || bookIsbn13 != null) {
101         throw new ClientException("BookAlreadyAdded", "Book already added");
```

```
102         }
103
104         // Create the book
105         Book book = new Book();
106         book.setId(UUID.randomUUID().toString());
107         bookDao.create(book);
108
109         // Create the user book
110         UserBookDao userBookDao = new UserBookDao();
111         UserBook userBook = new UserBook();
112         userBook.setUserId(principal.getId());
113         userBook.setBookId(book.getId());
114         userBook.setCreateDate(new Date());
115         userBookDao.create(userBook);
116
117         // Set the values of the book
118         setBook(book, userBook, title, subtitle, author, description, isbn10, isbn13,
                pageCount, language, publishDate, tagList);
119
120         // Returns the book ID
121         JSONObject response = new JSONObject();
122         response.put("id", userBook.getId());
123         return Response.ok().entity(response).build();
124     }
125
126     /**
127      * Updates the book.
128      *
129      * @param title Title
130      * @param description Description
131      * @return Response
132      * @throws JSONException
133      */
134     @POST
135     @Path("{id: [a-z0-9\\-]+}")
136     @Produces(MediaType.APPLICATION_JSON)
137     public Response update(
138             @PathParam("id") String userBookId,
139             @FormParam("title") String title,
140             @FormParam("subtitle") String subtitle,
141             @FormParam("author") String author,
142             @FormParam("description") String description,
143             @FormParam("isbn10") String isbn10,
144             @FormParam("isbn13") String isbn13,
145             @FormParam("page_count") Long pageCount,
146             @FormParam("language") String language,
147             @FormParam("publish_date") String publishDateStr,
148             @FormParam("tags") List<String> tagList) throws JSONException {
149         if (!authenticate()) {
150             throw new ForbiddenClientException();
151         }
152
153         Date publishDate = validateInput(title, subtitle, author, description, isbn10,
                isbn13, language, publishDateStr);
154
155         // Get the user book
156         UserBookDao userBookDao = new UserBookDao();
157         BookDao bookDao = new BookDao();
158         UserBook userBook = userBookDao.getUserBook(userBookId, principal.getId());
```

```
159     if (userBook == null) {
160         throw new ClientException("BookNotFound", "Book not found with id " +
                userBookId);
161     }
162 
163     // Get the book
164     Book book = bookDao.getById(userBook.getBookId());
165 
166     // Check that new ISBN number are not already in database
167     if (!Strings.isNullOrEmpty(isbn10) && book.getIsbn10() != null && !book.getIsbn10
            ().equals(isbn10)) {
168         Book bookIsbn10 = bookDao.getByIsbn(isbn10);
169         if (bookIsbn10 != null) {
170             throw new ClientException("BookAlreadyAdded", "Book already added");
171         }
172     }
173 
174     if (!Strings.isNullOrEmpty(isbn13) && book.getIsbn13() != null && !book.getIsbn13
            ().equals(isbn13)) {
175         Book bookIsbn13 = bookDao.getByIsbn(isbn13);
176         if (bookIsbn13 != null) {
177             throw new ClientException("BookAlreadyAdded", "Book already added");
178         }
179     }
180 
181     // Update the book
182     setBook(book, userBook, title, subtitle, author, description, isbn10, isbn13,
            pageCount, language, publishDate, tagList);
183 
184     // Returns the book ID
185     JSONObject response = new JSONObject();
186     response.put("id", userBookId);
187     return Response.ok().entity(response).build();
188 }
```

LLM Suggested Change

```
1  private void validateInput(String title, String subtitle, String author, String
       description, String isbn10, String isbn13,
2          Long pageCount, String language, String publishDateStr, List<String> tagList)
               {
3      // Validation logic
4  }
5 
6  private Book createBook(String title, String subtitle, String author, String
       description, String isbn10, String isbn13,
7          Long pageCount, String language, String publishDateStr) {
8      // Create and return a Book instance
9  }
10 
11 private void checkIfBookAlreadyExists(BookDao bookDao, Book book) {
12     // Check if the book already exists in the database
13 }
14 
15 private void saveBook(Book book, BookDao bookDao) {
16     // Save the book to the database
17 }
```

```
18
19  private UserBook createUserBook(Book book) {
20      // Create and return a UserBook instance
21  }
22
23  private void updateUserBookTags(List<String> tagList, UserBook userBook) {
24      // Update user book tags
25  }
26
27  private Response buildResponse(UserBook userBook) {
28      // Build and return the response
29  }
30
31  @PUT
32  @Path("manual")
33  @Produces(MediaType.APPLICATION_JSON)
34  public Response add(
35          @FormParam("title") String title,
36          @FormParam("subtitle") String subtitle,
37          @FormParam("author") String author,
38          @FormParam("description") String description,
39          @FormParam("isbn10") String isbn10,
40          @FormParam("isbn13") String isbn13,
41          @FormParam("page_count") Long pageCount,
42          @FormParam("language") String language,
43          @FormParam("publish_date") String publishDateStr,
44          @FormParam("tags") List<String> tagList) throws JSONException {
45
46      if (!authenticate()) {
47          throw new ForbiddenClientException();
48      }
49
50      validateInput(title, subtitle, author, description, isbn10, isbn13, pageCount,
            language, publishDateStr, tagList);
51
52      Book book = createBook(title, subtitle, author, description, isbn10, isbn13,
            pageCount, language, publishDateStr);
53
54      BookDao bookDao = new BookDao();
55      checkIfBookAlreadyExists(bookDao, book);
56
57      saveBook(book, bookDao);
58
59      UserBook userBook = createUserBook(book);
60
61      updateUserBookTags(tagList, userBook);
62
63      return buildResponse(userBook);
64  }
```

The LLM suggested excessive modularisation by creating numerous functions for each every task which could have been avoided.

Link to the prompt - LINK

3.2.2 Using Specific Exceptions

Many files contained the usage of generic exceptions instead of specific exception. We refactored the code to make use of specific exceptions to avoid the unnecessary abstraction design smell.

Initial Code

```
public void delete(String authenticationTokenId) throws Exception {
    EntityManager em = ThreadLocalContext.get().getEntityManager();
    AuthenticationToken authenticationToken = em.find(AuthenticationToken.class,
        authenticationTokenId);
    if (authenticationToken != null) {
        em.remove(authenticationToken);
    } else {
        throw new Exception("Token not found: " + authenticationTokenId);
    }
}
```

Manual Change

```
/**
 * Exception thrown when a token is not found.
 */
public class TokenNotFoundException extends Exception {
    public TokenNotFoundException(String message) {
        super(message);
    }
}

/**
 * Deletes the authentication token.
 *
 * @param authenticationTokenId Authentication token ID
 * @throws Exception
 */
public void delete(String authenticationTokenId) throws Exception {
    EntityManager em = ThreadLocalContext.get().getEntityManager();
    AuthenticationToken authenticationToken = em.find(AuthenticationToken.class,
        authenticationTokenId);
    if (authenticationToken != null) {
        em.remove(authenticationToken);
    } else {
        throw new TokenNotFoundException("Token not found: " + authenticationTokenId);
    }
}
```

LLM Suggested Change

```
public void deleteAuthenticationToken(String authenticationTokenId) throws Exception {
    EntityManager em = getCurrentEntityManager();
    AuthenticationToken authenticationToken = findAuthenticationTokenById(em,
        authenticationTokenId);
    if (authenticationToken != null) {
        removeAuthenticationToken(em, authenticationToken);
    } else {
        throwTokenNotFoundException(authenticationTokenId);
    }
}

// Method to get the EntityManager from the ThreadLocalContext
private EntityManager getCurrentEntityManager() {
```

```
    return ThreadLocalContext.get().getEntityManager();
}

// Method to find the AuthenticationToken by its ID
private AuthenticationToken findAuthenticationTokenById(EntityManager em, String
    authenticationTokenId) {
    return em.find(AuthenticationToken.class, authenticationTokenId);
}

// Method to remove the AuthenticationToken from the EntityManager
private void removeAuthenticationToken(EntityManager em, AuthenticationToken
    authenticationToken) {
    em.remove(authenticationToken);
}

// Method to throw an exception for a token not found
private void throwTokenNotFoundException(String authenticationTokenId) throws
    Exception {
    throw new Exception("Token not found: " + authenticationTokenId);
}
```

The LLM recognises the need for a specific exception handler however it creates unnecessary functions to improve modularisation which could have been avoided.

Link to the prompt - LINK

3.2.3 Removing Unnecessary Null Check

Many files contained the usage of unnecessary null checks. To deal with this design smell we changed the existing codebase to improve modularization by removing these unnecessary nullchecks.

Initial Code

```
if (appender == null || !(appender instanceof MemoryAppender)) {
    throw new ServerException("ServerError", "MEMORY appender not configured");
}
```

Manual Change

```
if (!(appender instanceof MemoryAppender)) {
    throw new ServerException("ServerError", "MEMORY appender not configured");
}
```

LLM Suggested Change

```
// Check if the appender is configured and of the correct type
if (appender == null || !isMemoryAppender(appender)) {
    throw new ServerException("ServerError", "MEMORY appender not configured");
}

// Method to check if the appender is an instance of MemoryAppender
private boolean isMemoryAppender(Appender appender) {
    return appender instanceof MemoryAppender;
}
```

The LLM refactors the code to improve modularisation by adding a function to handle a conditional check however it fails to recognise the usage of an unnecessary null check present in the code snippet.

3.2.4 Use of method isEmpty()

Files contained the usage of *array.size() > 0* statement to check if the array is not empty rather than using a more abstract *!array.isEmpty()* condition to verify that the array is not empty. Through this we handled an incomplete abstraction design smell.

Initial Code

```
if (l.size() > 0) {
    throw new AlreadyExistingUsernameException("AlreadyExistingUsername");
}
```

Manual Change

```
if (!l.isEmpty()) {
    throw new AlreadyExistingUsernameException("AlreadyExistingUsername");
}
```

LLM Suggested Change

```
// Check if the list contains elements
if (!isListEmpty(l)) {
    throwUsernameAlreadyExistsException();
}

// Method to check if the list is empty
private boolean isListEmpty(List<?> list) {
    return list.isEmpty();
}

// Method to throw an exception for an already existing username
private void throwUsernameAlreadyExistsException() throws Exception {
    throw new Exception("AlreadyExistingUsername");
}
```

The LLM identifies that the usage of *array.size() > 0* is a design smell and make the right changes by introducing a new function to check if the array is empty. However to handle the exception it adds an unnecessary function to introduce a new Unnecessary Abstraction design smell.

3.2.5 Closing opened files

In 3 files, a file input stream was opened in a class method, but not closed. This could lead to resource leaks. To fix this, we used try-catch-finally statements and try-with-resource statements to ensure that open file streams are closed when the method exits. The refactored code takes care of the *Leaky Encapsulation* design smell.

Initial Code

```
/**
 * Returns a book cover.
 *
 * @param id User book ID
 * @return Response
 * @throws JSONException
```

```
 */
@GET
@Path("{id: [a-z0-9\\-]+}/cover")
@Produces(MediaType.APPLICATION_OCTET_STREAM)
public Response cover(@PathParam("id") final String userBookId) throws JSONException {
    // Get the user book
    UserBookDao userBookDao = new UserBookDao();
    UserBook userBook = userBookDao.getUserBook(userBookId);

    // Get the cover image
    File file = Paths.get(DirectoryUtil.getBookDirectory().getPath(), userBook.
        getBookId()).toFile();
    InputStream inputStream = null;
    try {
        if (file.exists()) {
            inputStream = new FileInputStream(file);
        } else {
            inputStream = new FileInputStream(new File(getClass().getResource("/dummy.
                png").getFile()));
        }
    } catch (FileNotFoundException e) {
        throw new ServerException("FileNotFound", "Cover file not found", e);
    }

    return Response.ok(inputStream)
            .header("Content-Type", "image/jpeg")
            .header("Expires", new SimpleDateFormat("EEE, dd MMM yyyy HH:mm:ss Z").
                format(new Date().getTime() + 3600000))
            .build();
}
```

Manual Change

```
/**
 * Returns a book cover.
 *
 * @param id User book ID
 * @return Response
 * @throws JSONException
 */
@GET
@Path("{id: [a-z0-9\\-]+}/cover")
@Produces(MediaType.APPLICATION_OCTET_STREAM)
public Response cover(@PathParam("id") final String userBookId) throws JSONException {
    // Get the user book
    UserBookDao userBookDao = new UserBookDao();
    UserBook userBook = userBookDao.getUserBook(userBookId);

    // Get the cover image
    File file = Paths.get(DirectoryUtil.getBookDirectory().getPath(), userBook.
        getBookId()).toFile();

    byte[] fileContent;
    try (InputStream inputStream = file.exists() ? new FileInputStream(file) :
        getClass().getResourceAsStream("/dummy.png")) {

        if (inputStream == null) {
```

```
            throw new ServerException("FileNotFound", "Cover file not found");
        }

        // Read the file content into a byte array
        fileContent = IOUtils.toByteArray(inputStream);

    } catch (IOException e) {
        throw new ServerException("IOException", "Error reading cover file", e);
    }

    // Close the InputStream before building the response
    return Response.ok(fileContent)
            .header("Content-Type", "image/jpeg")
            .header("Expires", new SimpleDateFormat("EEE, dd MMM yyyy HH:mm:ss Z").
                format(new Date().getTime() + 3600000))
            .build();
}
```

LLM Suggested Change

```
/**
 * Returns a book cover.
 *
 * @param id User book ID
 * @return Response
 * @throws JSONException
 */
@GET
@Path("{id: [a-z0-9\\-]+}/cover")
@Produces(MediaType.APPLICATION_OCTET_STREAM)
public Response cover(
        @PathParam("id") final String userBookId) throws JSONException {
    // Get the user book
    UserBookDao userBookDao = new UserBookDao();
    UserBook userBook = userBookDao.getUserBook(userBookId);

    // Get the cover image
    File file = Paths.get(DirectoryUtil.getBookDirectory().getPath(), userBook.
        getBookId()).toFile();
    try (InputStream inputStream = file.exists()
            ? new FileInputStream(file)
            : getClass().getResourceAsStream("/dummy.png")) {
        if (inputStream == null) {
            throw new ServerException("FileNotFound", "Cover file not found");
        }

        return Response.ok(inputStream)
                .header("Content-Type", "image/jpeg")
                .header("Expires", new SimpleDateFormat("EEE, dd MMM yyyy HH:mm:ss Z")
                    .format(new Date().getTime() + 3600000))
                .build();
    } catch (IOException e) {
        throw new ServerException("IOException", "Error reading cover file", e);
    }
}
```

Here, the LLM gave some options like using *try-with-resource* and adding *finally* statement, but the books cover function was giving the default cover each time, when the file input stream was closed. After

asking the LLM for another way, it eventually gave the code I used in manual refactoring, where I read the file contents and save it to a variable (buffer) before closing it.

In the other 2 files, a *finally* statement was enough to close the file input stream opened earlier in the corresponding method.

Link to the prompt - LINK

3.2.6 Defining a constant instead of duplicating literals

Within a specific folder, multiple files had duplicate literals which were being used several times across the files. We refactored the code to introduce global variables in a new file that can be used across all files within that folder to avoid leaky encapsulation smell.

Initial Code

```
public void deleteByUserIdAndAppId(String userId, String appId) {
    EntityManager em = ThreadLocalContext.get().getEntityManager();
    Query q = em.createQuery("update UserApp ua set ua.deleteDate = :deleteDate where
        ua.userId = :userId and ua.appId = :appId and ua.deleteDate is null");
    q.setParameter("deleteDate", new Date());
    q.setParameter("userId", userId);
    q.setParameter("appId", appId);
    q.executeUpdate();
}
```

```
public UserApp getActiveByUserIdAndAppId(String userId, String appId) {
    EntityManager em = ThreadLocalContext.get().getEntityManager();
    StringBuilder sb = new StringBuilder("select distinct ua from UserApp ua");
    sb.append(" where ua.userId = :userId and ua.appId = :appId ");
    sb.append(" and ua.deleteDate is null ");
    sb.append(" order by ua.createDate desc ");
    Query q = em.createQuery(sb.toString());
    try {
        q.setParameter("userId", userId);
        q.setParameter("appId", appId);
        return (UserApp) q.getSingleResult();
    } catch (NoResultException e) {
        return null;
    }
}
```

Manual Change

Declaring Constants: (New file)

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

public class ConstantsDao {
    public static final String id = "id";
    public static final String name = "name";
    public static final String username = "username";
    public static final String tagId = "tagId";
    public static final String userId = "userId";
    public static final String userBookId = "userBookId";
    public static final String bookId = "bookId";
    public static final String appId = "appId";
    public static final String deleteDate = "deleteDate";
    public static final String longLasted = "longLasted";
```

```
    public static final String minDate = "minDate";
    public static final String currentDate = "currentDate";
    public static final String updateDate = "updateDate";
    public static final String passwordResetKey = "passwordResetKey";
}
```

Replacing literals:

```
public void deleteByUserIdAndAppId(String userId, String appId) {
    EntityManager em = ThreadLocalContext.get().getEntityManager();
    Query q = em.createQuery("update UserApp ua set ua.deleteDate = :deleteDate where
        ua.userId = :userId and ua.appId = :appId and ua.deleteDate is null");
    q.setParameter(ConstantsDao.deleteDate, new Date());
    q.setParameter(ConstantsDao.userId, userId);
    q.setParameter(ConstantsDao.appId, appId);
    q.executeUpdate();
}
```

```
public UserApp getActiveByUserIdAndAppId(String userId, String appId) {
    EntityManager em = ThreadLocalContext.get().getEntityManager();
    StringBuilder sb = new StringBuilder("select distinct ua from UserApp ua");
    sb.append(" where ua.userId = :userId and ua.appId = :appId ");
    sb.append(" and ua.deleteDate is null ");
    sb.append(" order by ua.createDate desc ");
    Query q = em.createQuery(sb.toString());
    try {
        q.setParameter(ConstantsDao.userId, userId);
        q.setParameter(ConstantsDao.appId, appId);
        return (UserApp) q.getSingleResult();
    } catch (NoResultException e) {
        return null;
    }
}
```

LLM Suggested Change

Declaring Constants: (Same file)

```
private static final String DELETE_DATE_PARAM = "deleteDate";
private static final String USER_ID_PARAM = "userId";
private static final String APP_ID_PARAM = "appId";
private static final String DELETE_DATE_IS_NULL = " and ua.deleteDate is null";
private static final String SELECT_USER_APP = "select ua from UserApp ua";
private static final String ORDER_BY_CREATE_DATE_DESC = " order by ua.createDate desc"
    ;
private static final String SELECT_DISTINCT_USER_APP = "select distinct ua from
    UserApp ua";
```

Replacing literals:

```
public void deleteByUserIdAndAppId(String userId, String appId) {
    EntityManager em = ThreadLocalContext.get().getEntityManager();
    Query q = em.createQuery("update UserApp ua set ua.deleteDate = :" +
        DELETE_DATE_PARAM + " where ua.userId = :" + USER_ID_PARAM + " and ua.appId = :
        " + APP_ID_PARAM + DELETE_DATE_IS_NULL);
    q.setParameter(DELETE_DATE_PARAM, new Date());
    q.setParameter(USER_ID_PARAM, userId);
    q.setParameter(APP_ID_PARAM, appId);
```

```
    q.executeUpdate();
}
```

```
public UserApp getActiveByUserIdAndAppId(String userId, String appId) {
    EntityManager em = ThreadLocalContext.get().getEntityManager();
    StringBuilder sb = new StringBuilder(SELECT_USER_APP);
    sb.append(" where ua.userId = :" + USER_ID_PARAM + " and ua.appId = :" +
        APP_ID_PARAM);
    sb.append(DELETE_DATE_IS_NULL);
    sb.append(ORDER_BY_CREATE_DATE_DESC);
    Query q = em.createQuery(sb.toString());
    try {
        q.setParameter(USER_ID_PARAM, userId);
        q.setParameter(APP_ID_PARAM, appId);
        return (UserApp) q.getSingleResult();
    } catch (NoResultException e) {
        return null;
    }
}
```

Since only 1 file was given to the LLM to refactor, the output we get from that makes use of excess constants since they are all private and local constants. By manual refactoring, we were able to introduce global constants that can be used across all files in a folder

Link to the prompt - LINK

3.2.7 Hiding type specification within diamond operator (”<>”)

Type specification within diamond operator becomes redundant because the variable declaration statement already contains type specification

Initial Code

```
/**
 * Returns tag list on a user book.
 * @param userBookId
 * @return
 */
@SuppressWarnings("unchecked")
public List<TagDto> getByUserBookId(String userBookId) {
    EntityManager em = ThreadLocalContext.get().getEntityManager();
    StringBuilder sb = new StringBuilder("select t.TAG_ID_C, t.TAG_NAME_C, t.
        TAG_COLOR_C from T_USER_BOOK_TAG bt ");
    sb.append(" join T_TAG t on t.TAG_ID_C = bt.BOT_IDTAG_C ");
    sb.append(" where bt.BOT_IDUSERBOOK_C = :userBookId and t.TAG_DELETEDATE_D is null
         ");
    sb.append(" order by t.TAG_NAME_C ");

    // Perform the query
    Query q = em.createNativeQuery(sb.toString());
    q.setParameter(ConstantsDao.userBookId, userBookId);
    List<Object[]> l = q.getResultList();

    // Assemble results
    List<TagDto> tagDtoList = new ArrayList<TagDto>();
    for (Object[] o : l) {
        int i = 0;
        TagDto tagDto = new TagDto();
```

```
        tagDto.setId((String) o[i++]);
        tagDto.setName((String) o[i++]);
        tagDto.setColor((String) o[i++]);
        tagDtoList.add(tagDto);
    }
    return tagDtoList;
}
```

Manual Change

```
/**
 * Returns tag list on a user book.
 * @param userBookId
 * @return
 */
@SuppressWarnings("unchecked")
public List<TagDto> getByUserBookId(String userBookId) {
    EntityManager em = ThreadLocalContext.get().getEntityManager();
    StringBuilder sb = new StringBuilder("select t.TAG_ID_C, t.TAG_NAME_C, t.
        TAG_COLOR_C from T_USER_BOOK_TAG bt ");
    sb.append(" join T_TAG t on t.TAG_ID_C = bt.BOT_IDTAG_C ");
    sb.append(" where bt.BOT_IDUSERBOOK_C = :userBookId and t.TAG_DELETEDATE_D is null
         ");
    sb.append(" order by t.TAG_NAME_C ");

    // Perform the query
    Query q = em.createNativeQuery(sb.toString());
    q.setParameter(ConstantsDao.userBookId, userBookId);
    List<Object[]> l = q.getResultList();

    // Assemble results
    List<TagDto> tagDtoList = new ArrayList<>();
    for (Object[] o : l) {
        int i = 0;
        TagDto tagDto = new TagDto();
        tagDto.setId((String) o[i++]);
        tagDto.setName((String) o[i++]);
        tagDto.setColor((String) o[i++]);
        tagDtoList.add(tagDto);
    }
    return tagDtoList;
}
```

LLM Suggested Change

```
// Assemble results
List<TagDto> tagDtoList = new ArrayList<>();
for (Object[] o : l) {
    int i = 0;
    TagDto tagDto = new TagDto();
    tagDto.setId((String) o[i++]);
    tagDto.setName((String) o[i++]);
    tagDto.setColor((String) o[i++]);
    tagDtoList.add(tagDto);
}
return tagDtoList;
```

LLM also suggested the same change.
Link to the prompt - LINK

4 Contribution

- Task 1 - Likhith and Vikram
- Task 2(a) - Aakash and Rahothvarman
- Task 2(b) - Priyanshul
- Task 3(a & c) - Aakash, Likhith, Rahothvarman and Vikram
- Task 3(b) - Priyanshul
- Bonus - Priyanshul