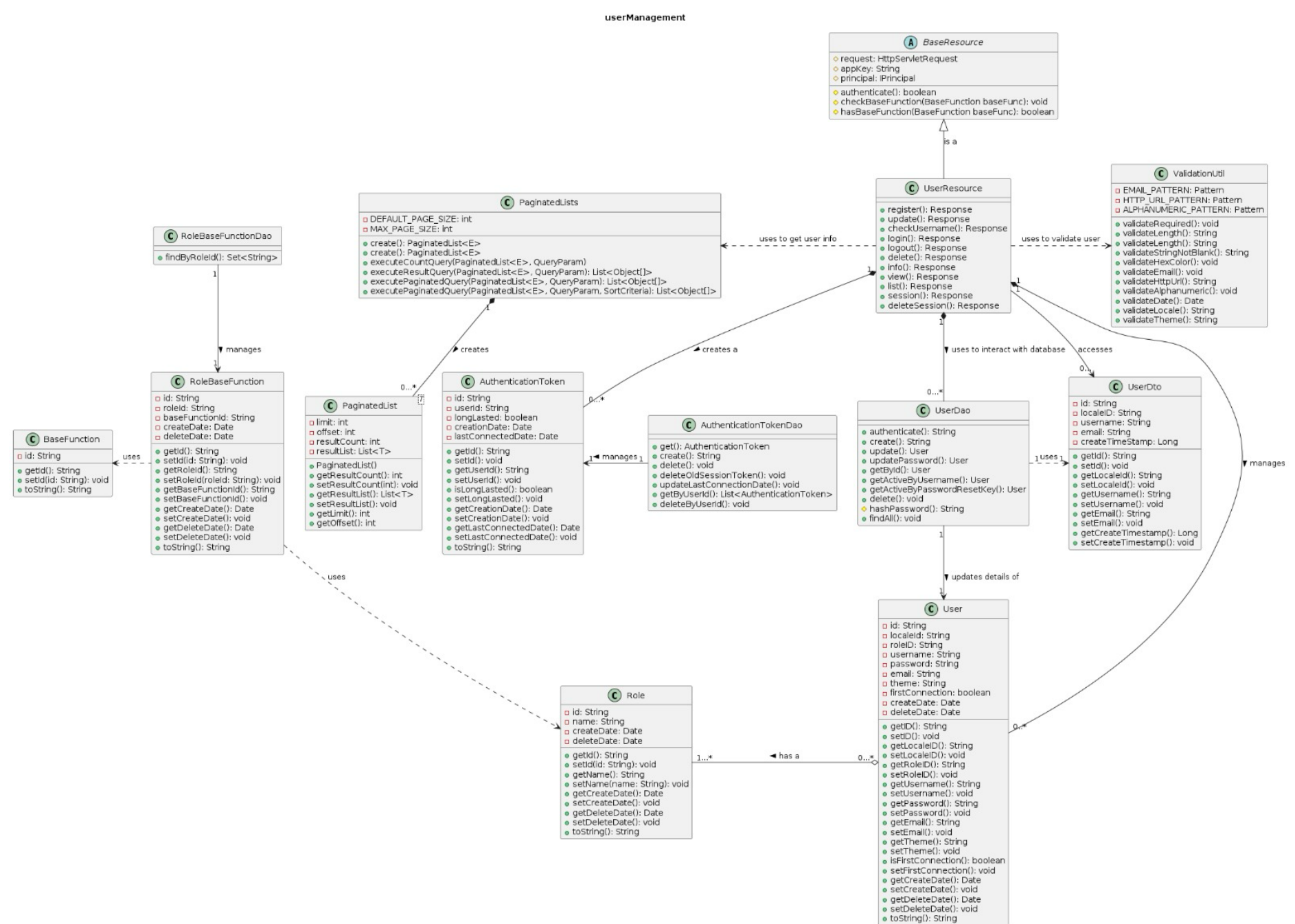


Software Engineering | Project 1 | Team 15

1. User Management Subsystem

The User Management Subsystem is responsible for user authentication, authorization, and session management. It comprises classes such as `UserResource`, `UserDao`, and `AuthenticationToken`, which collaborate to ensure the security and integrity of user interactions. By defining roles and base functions, the subsystem establishes a structured framework for managing user permissions and access levels. Overall, it facilitates essential functionalities like user registration, login, logout, and session tracking, guaranteeing a smooth and secure user experience within the application.



Classes

- BaseResource**: Serves as a base class for other REST resources, providing shared functionality and structure for building RESTful endpoints within the subsystem.

Role in the subsystem:

- Manages the authentication of users
- Offers methods to verify if the authenticated user has specific base functions, enforcing authorization checks.

- UserResource**: This class inherits from the 'BaseResource' class and handles user related operations within the User Management subsystem. It provides endpoints for actions such as user registration, updating user information, login, logout and managing user sessions.

Role in the subsystem:

- Acts as an endpoint for various user operations.
- Implements access control mechanisms, ensuring that sensitive operations require proper authentication and authorization, especially for admin-specific functionalities.

- Manages user sessions securely, providing login and logout functionalities, as well as allowing users to view and manage their active sessions.
- Integrates with Data Access Objects (DAOs) like 'UserDao' and 'AuthenticationTokenDao' to interact with the database.

3. **User:** This class represents an individual user entity with attributes like username, password, email and role.

Role in the subsystem:

- The class is used to create new user instances.
- Captures user information needed for operations managed by the 'UserResource' class.
- The 'roleID' attribute contributes to access control mechanisms, determining the user's role and associated permissions.

4. **UserDao:** This class acts as an interface between the User Management system and the underlying database, facilitating various operations related to user data.

Role in the subsystem:

- Handles authentication and facilitates creation and modification of user data.
- Ensures the secure handling of user passwords through hashing during creation and updates.
- Allows retrieval of user data based on various parameters like ID, username, or password reset key.
- Handles the deletion of user accounts, marking them as deleted and clearing linked data.

5. **UserDto:** It is a Data Transfer Object that comprises the id, username, email, and localeId. The purpose of DTOs is to package multiple parameters in one object, so as to reduce the number of method calls.

Role in the system:

- Used for transferring user-specific data between UserResource and the underlying database.

6. **PaginatedList:** Represents a subset of data for a specific page in a paginated query result. Maintains information about the page size, offset, total result count, and the list of records for the current page.

Role in the system:

- Enables the division of large datasets into smaller, manageable chunks or pages, optimizing the presentation and retrieval of data for users.

7. **PaginatedLists:** This class serves as a component for managing paginated lists within the user management subsystem, offering functionality for navigating through large datasets efficiently.

Role in the system:

- PaginatedLists facilitates the creation of paginated lists, allowing for dynamic configuration of page size and offset, providing flexibility in determining the amount of data displayed per page and the starting point for data retrieval.
- Manages the execution of both count and result queries, accurately determining the total number of results available and retrieving the appropriate subset of data for the current page.
- Supports sorting criteria, enabling developers to specify attributes and order for sorting data within pages. This flexibility enhances usability by allowing users to view data in preferred sequences.

8. **AuthenticationToken:** This class handles essential attributes and behaviors related to authentication tokens, ensuring the validation and effective management of user sessions in the system.

Role in the subsystem:

- The class facilitates the creation of authentication tokens upon successful user authentication.
- The 'longLasted' attribute allows the subsystem to distinguish between regular and long-lasting user sessions, providing flexibility in managing session durations.
- Allows last connection tracking, aiding in session monitoring and management.

9. **AuthenticationTokenDao:** This class is responsible for managing authentication tokens and facilitating secure user sessions within the system.

Role in the subsystem:

- Allows creation of new authentication tokens which are stored securely in the database.
- Allows retrieval and deletion of authentication tokens based on their id.
- The 'deleteOldSessionToken' method plays a crucial role in maintaining a clean and secure environment by removing outdated short-lived tokens.
- The 'updateLastConnectionDate' method ensures that the last connection date of an authentication token is updated, aiding in session monitoring.

10. **ValidationUtil:** This utility class provides a set of methods for validating various parameters and ensuring the integrity of user-related inputs.

Role in the subsystem:

- Used to validate user inputs during operations such as user creation, update, or authentication.
- Guarantees only existing and valid locale and theme identifiers are accepted during user operations.
- Enhances the robustness and reliability of the subsystem by enforcing stringent validation rules, thereby maintaining data integrity.

11. **BaseFunction:** Each base function corresponds to a specific functionality or permission that can be assigned to users directly or as part of a role.

Role in the subsystem:

- Acts as a key component in defining the base functions that collectively form roles. These roles, in turn, govern the permissions granted to users within the system.

12. **Role:** This class manages roles that group together sets of base functions or permissions.

Role in the subsystem:

- Roles define the access levels granted to users based on associated base functions.

13. **RoleBaseFunction:** This class establishes associations between roles and base functions.

Role in the subsystem:

- This class links roles to the specific base functions that define their permissions.
- Stores creation and deletion dates of the relation, preventing deleted associations from being fetched.

14. **RoleBaseFunctionDao:** This class is responsible for interacting with the underlying database to retrieve information related to role-base function associations.

Role in the subsystem:

- Used by the User Management Subsystem to fetch the set of base functions associated with a particular role.
- This class considers the deletion date to exclude deleted role-base function associations, ensuring only active associations are retrieved.

Strengths of the System

1. **Modularity:** The system is well-structured with different classes for different functionalities, facilitating easier development and updates.
2. **Security Measures:** The system implements authentication and authorization mechanisms to control access to resources and ensure data security. For example, it checks for valid authentication tokens and base function permissions before allowing access to certain endpoints.
3. **Session Management:** The system creates and removes session tokens as necessary. It provides easy-to-use endpoints for users to keep track of and manage their active sessions, making the system more secure and giving users more control over their experience.

Weaknesses of the System

1. **Complexity:** While the code is somewhat modular, some of the classes in the system contain a significant amount of logic, which may make it challenging to maintain and debug. Breaking them down further into utility classes would improve the readability and

maintenance of the system.

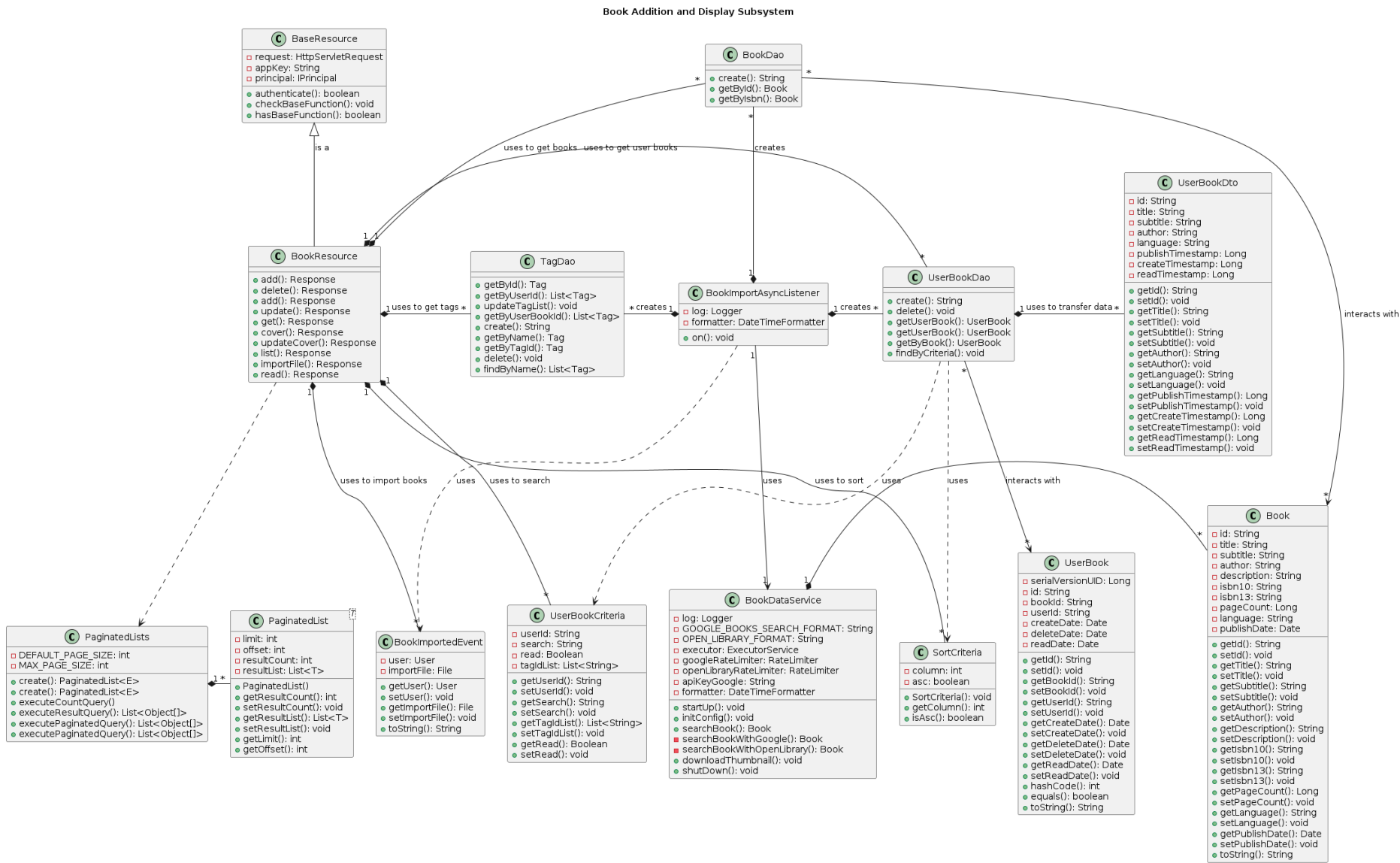
- 2. **Code Duplication:** Some of the class methods exhibit code duplication, particularly in input validation and error handling logic. Extracting common logic into helper methods would reduce redundancy and improve readability.
- 3. **Exception Handling:** While the system handles exceptions in certain cases, there are opportunities to improve exception handling and error reporting, especially for unexpected errors or edge cases.
- 4. **User Groups and Organization Units:** While the system currently has role-based access functions, it lacks the capability to organize users into groups or organizational units. This absence prevents the system from providing a structured approach to user management and limits the application of group-based permissions. Additionally, the system does not support the inheritance of permissions.

Assumptions

- 1. **RoleBaseFunction:** Manages the assignment of a role's set of base functions.
- 2. **Dao Functionality:** Each Dao manages data related to its corresponding function in the database. For example, the 'AuthenticationTokenDao' handles data associated with authentication tokens. Although 'AuthenticationToken' is linked to UserResource in the UML diagram, indicating its role in user authentication processes, the AuthenticationTokenDao is connected to AuthenticationToken, ensuring efficient management of token-related data without direct relation to 'UserResource'.

2. Book Addition and Display Subsystem

The Book Addition and Display Subsystem manages books and their related data in our application. Key components include classes like BookDao for database interactions, BookResource for handling CRUD operations, and BookImportAsyncListener for asynchronous book importation. It also utilizes DTOs and entities like UserBookDto and Tag to streamline data management. Overall, the subsystem ensures efficient book handling from creation to display, offering functionalities such as CRUD operations and cover image management.



Classes

- 1. **BookDao:** The BookDao class provides methods for the creation and retrieval of Books.
Role in the subsystem:
 - Uses an EntityManager object to perform operations on the persistence layer.

- Allows creation of new Books.
- Allows for retrieval of a Book either by using its ID (getById) or by querying the underlying database with the provided ISBN (getByIsbn).

2. **TagDao:** The TagDao class provides an interface between the system and the tag database. The operations provided in TagResource are carried out in this class via queries to the database.

Role in the subsystem:

- Uses an EntityManager object to perform operations on the persistence layer.
- Writes queries to:
 - Get a tag by ID.
 - List all tags for that user.
 - Update tag list.
 - List the tags on a book.
 - Create a new tag.
 - Search for tags by name.
 - Get a single tag by name.
 - Get a single tag by ID.
 - Delete a tag by ID.

3. **UserBookDao:** The UserBookDao class is the DAO for a user book. Provides for dealings with UserBooks.

Role in the subsystem:

- Lists all the books that satisfy a set of criteria (which are obtained from UserBookCriteria and SortCriteria).
- Provides an abstract interface to access books for UserBook.

4. **UserBookDto:** The UserBookDto class is the DTO for the class UserBookDao. Provides for dealings with UserBooks via UserBookDao.

Role in the subsystem:

- This DTO class is called by the corresponding DAO class to interact with the DB.

5. **BookImportedEvent:** The BookImportedEvent class represents an event raised when a user requests the import of books. It contains information about the user making the request and the temporary file that is intended for the import process.

Role in subsystem:

- Retrieves and sets User object and the imported file for a Book.

6. **Book:** The Book class represents an entity in the context of a book management system, specifically designed for Java Persistence API (JPA). This class is responsible for mapping the properties of a book to corresponding columns in a relational database table.

Role in subsystem:

- Retrieves and sets the unique identifier of a book, title of a book, subtitle of a book, author name, description, ISBN 10 or 13 code, page count, language and published date of a book.

7. **Tag:** The representation of the entity 'tag'. It stores relevant attributes along with the corresponding setters & getters.

Role in the subsystem:

- The object returned by the TagDao when we search for or get a single tag, is a Tag.

8. **UserBook:** A UserBook object represents a user copy of a particular book. It contains the user's and the book's ID.

Role in the subsystem:

- Stores the list of User's book with their respective IDs. Used by other classes to access a particular user's books.

9. **UserBookTag:** Tags associated with each UserBook object. They are serialised for each object.

Role in the subsystem:

- Created by the Entity Manager in TagDao when tags are being updated. As far as we can tell, objects of this class are not accessed or used anywhere in the codebase.
- Sets details in the internal database, which allows us to perform queries on it.

10. **UserBookCriteria:** Stores information about a UserBook search, such as userId, search query, read state, and tag ID list.

Role in the subsystem:

- This is where the search criteria are stored.

11. **SortCriteria:** Stores information related to how books should be sorted when queried.

Role in subsystem:

- This is where the sort criteria are stored.

12. **BaseResource:** The BaseResource class serves as the base class for REST resources in the application. It contains common functionalities and fields used by other resource classes.

Role in the subsystem:

- Manages the authentication of users.
- Offers methods to verify if the authenticated user has specific base functions, enforcing authorization checks.

13. **BookResource:** The BookResource class is responsible for handling CRUD requests related to books (userBook) in the application. It inherits from the BaseResource class. It contains methods for adding, deleting, updating, retrieving, and importing books, as well as managing book covers and read status.

Role in the subsystem:

- Allows the addition of new userBooks using either the provided ISBN or using manually provided details.
- Allows deletion of books specified by userBook ID.
- Updates the details and cover image of a book.
- Allows retrieval of details and cover image of a book, and retrieval of a list of books based on the provided criteria.
- Imports books from a file.
- Marks a userBook as read or unread.

14. **BookDataService:** The BookDataService class is responsible for fetching book information from external APIs such as Google Books and Open Library. Searches Google and Open Library using the ISBN of the specified book. If found, the requested book is created, and the Book class is instantiated.

Role in the subsystem:

- Using external APIs it searches for a book using the provided ISBN.
- If the book is found, it is added to the Books library.

15. **BookImportAsyncListener:** The BookImportAsyncListener class is responsible for processing book import requests asynchronously. It listens for BookImportedEvent events and processes the information from the event's associated file. It handles the creation and updatation of book and user book entries in the database. It also manages associated tags by interacting with TagDao, BookDao, and UserBookDao classes. It interacts with Book, Tag, and UserBook classes for creating books, tags, and UserBooks if not already present.

Role in the subsystem:

- Parses the information from the event's associated file.
- Creates or updates book and user book entries in the database.
- Manages associated tags by interacting with TagDao, BookDao, and UserBookDao classes.

Strengths of the System

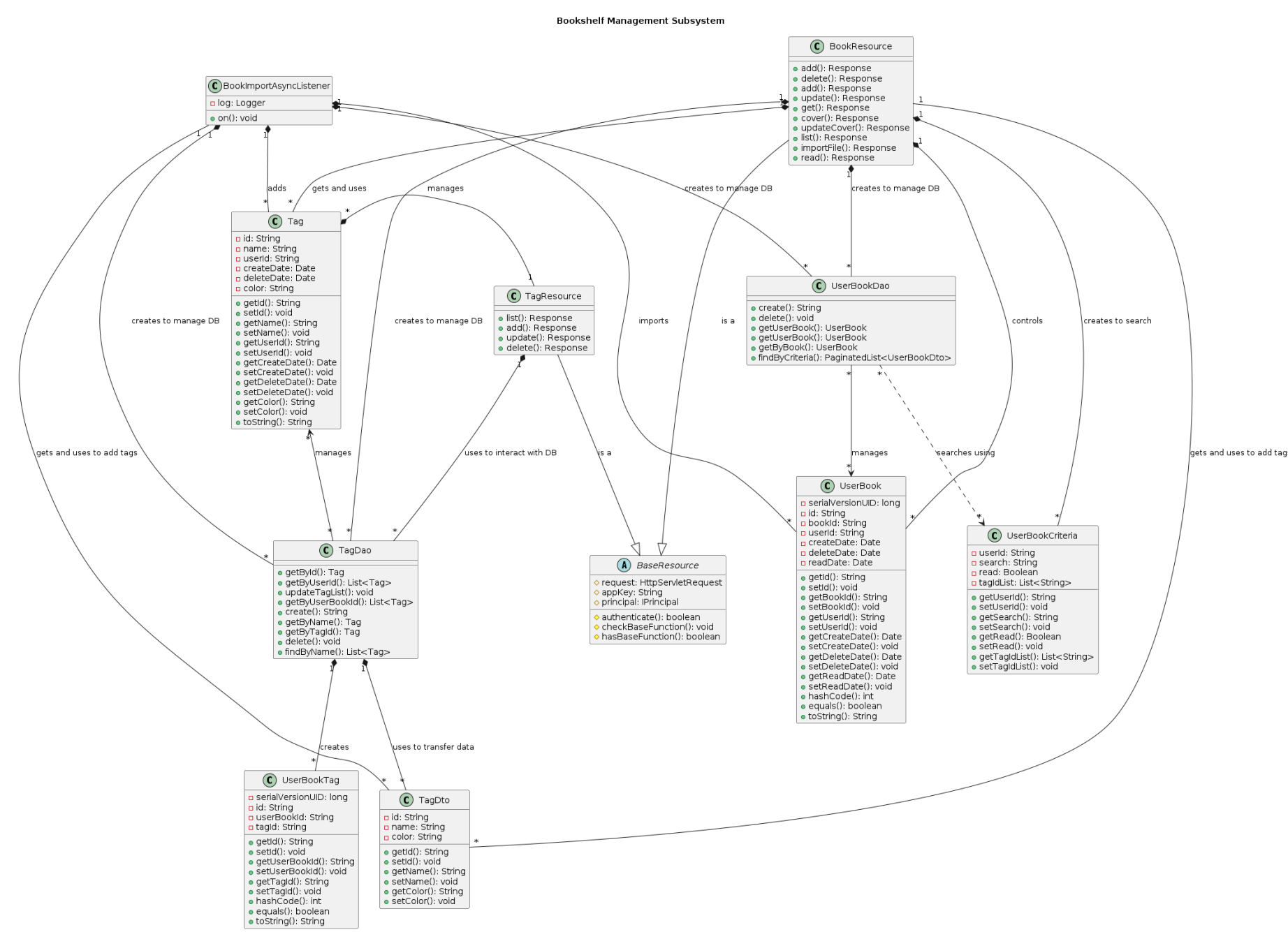
- 1. **Modularity:** The system is well-structured with different classes for different functionalities, facilitating easier development and updates.
- 2. **Book Addition Management:** The system boasts a versatile book addition subsystem, offering users multiple methods to add books. Users can input book details manually, leveraging external APIs like Goodreads and Google Books for automated retrieval and addition to the system. Additionally, books can be swiftly added using either ISBN 10 or 13 codes. For Android users, the option of a Barcode Scanner further enhances the book addition process.

Weaknesses of the System

- 1. **Complexity:** While the code is somewhat modular, some of the classes in the system contain a significant amount of logic, which may make it challenging to maintain and debug. Breaking them down further into utility classes would improve the readability and maintenance of the system.
- 2. **God Class:** In the system, certain classes display an overload of responsibilities, resembling what is commonly known as a "god class." One such instance is observed in the 'BookResource' class, where it has a lot of functionalities. This class appears to act as a central hub, coupled with multiple other classes, leading to code that is overly complex and difficult to maintain. By decomposing this class into smaller, more specialized counterparts, the system's architecture could be significantly improved, promoting better code organization and scalability.

3. Bookshelf Management Subsystem

The Bookshelf Management Subsystem manages and manipulates book tags within the application. Essential components such as BaseResource, TagResource, and TagDao handle authentication, tag operations, and database interactions. UserBook and UserBookDao manage user-specific book copies and search criteria, while TagDto streamlines data transfer for tag-related attributes. BookImportAsyncListener ensures efficient processing of book import requests.



Classes

1. **BaseResource:** Serves as a base class for other REST resources, providing shared functionality and structure for building RESTful endpoints within the subsystem. It is inherited by other classes.

Role in the subsystem:

- Provides a method to check if the user is authenticated.

2. **TagResource:** This class inherits from the Base Resource class and manages tags (also known as “bookshelves”). It provides endpoints for various tag-related functionality.

Role in the subsystem:

- Provides methods to list all tags and update/delete/add a tag.
- Implements access control mechanisms to ensure that only authenticated users can perform operations.
- Interfaces with the Tag Data Access Object (TagDao) to provide the aforementioned functionality.
- Validates input data when a new tag is created or an existing one is updated. Also checks for duplicates.

3. **TagDao:** Provides an interface between the system and the tag database. This is where the operations provided in TagResource are actually carried out via queries.

Role in the subsystem:

Writes queries to

- Get tag by id
- List all tags for that user
- Update tag list
- List the tags on a book
- Create a new tag
- Search for tags by name
- Get a single tag by name
- Get a single tag by ID
- Delete a tag by ID
- Uses an EntityManager object to perform operations on the persistence layer.

4. **UserBook:** A UserBook object represents a “copy” of a particular book for a particular user. It contains the user’s and the book’s Id.

Role in the subsystem:

- No explicit role, but is used within several other classes.

5. **UserBookDao:** It is the DAO for a user book. Provides for dealings with UserBooks.

Role in the subsystem:

- Its role is to list all the books that satisfy a set of criteria (in our subsystem, we are concerned with books that have a certain tag).

6. **UserBookCriteria:** Stores information about a UserBook search, such as: userId, search query, read state, and tag Id list.

Role in the subsystem:

- This is where the criteria (including tags) are stored.

7. **TagDto:** It is a Data Transfer Object that comprises the tag Id, name, and colour. The purpose of DTOs is to package multiple parameters in one object, so as to reduce the number of method calls.

- This is used across the codebase in place of the id/name/colour or any combination of them e.g. List<TagDto>.
- This is also what is returned by TagDao objects when they return a list of tags. In effect, a TagDto object represents a tag with relevant info, except multiple such objects can be made from one tag.

8. **BookImportAsyncListener:** This is a listener for import requests on books.

Role in the subsystem:

- When a book is imported, this object retrieves the list of tags (at GoodReads). If a tag does not already exist for that book, it is added. This process is facilitated by TagDao and aided by TagDto.

9. **Tag:** The representation of the entity 'tag'. It stores relevant attributes along with the corresponding setters & getters

Role in the subsystem:

- The object returned by the TagDao when we search for or get a single tag, is a Tag.

10. **UserBookTag:** Tags associated with each UserBook object. They are serialised for each object.

Role in the subsystem:

- Created by the Entity Manager in TagDao when tags are being updated. As far as we can tell, objects of this class are not accessed or used anywhere in the codebase.
- Sets details in the internal database, which allows us to perform queries on it.

11. **BookResource:** Inherits from BaseResource. Allows for creating, retrieving, deleting, importing, getting cover of, updating cover of, listing, and reading a UserBook. We are not concerned with most of it for this subsystem.

- While adding a book manually or when updating, its tags are passed as a list of strings. This is done with the help of TagDao.
- When "get"-ing a book, its tags are returned as a list of TagDto objects, which are then converted to a list of JSONObjects.
- When searching ("list"), the Tag object associated with the tag name is retrieved using a TagDao object; then, the tag-lists of the results are retrieved as a list of TagDto objects.

Strengths of the System

1. **High Cohesion and Low Complexity:** The system demonstrates strong cohesion among its classes, ensuring that each component has a clear and focused responsibility. Additionally, the low complexity of the codebase enhances manageability and reduces the likelihood of errors.
2. **Convenient Book Addition:** The system offers multiple methods for adding books, including ISBN lookup and integration with external databases like Goodreads and OpenLibrary. This variety provides users with flexibility and convenience in acquiring their desired books.
3. **Tag System for Book Organization:** The inclusion of a tag system enables users to categorize their books effectively, facilitating easy organization and retrieval of books based on specific criteria.

Weaknesses of the System

1. **Tedious Book Retrieval by Tag:** Although the tag system allows for efficient categorization, the process of finding books based on tags may be cumbersome. Users may need to scroll through a long list of books associated with a particular tag, which can impede accessibility and user experience.
2. **Lack of Generalized Tags:** The absence of broader, pre-defined tags such as "sci-fi," "mathematics," or other common genres/themes may hinder users from quickly browsing through relevant categories. Introducing more generalized tags could streamline the browsing experience and enhance user satisfaction.

Assumptions

1. Since each time an operation is performed on a Tag/Book, a new DAO object is created, and when a book is imported a single DAO object can deal with multiple Tag/Book objects, it is a many-to-many relationship. The same goes for UserBookCriteria's relationship with the UserBook DAO.

Design Smells

BaseResource.java

- **Broken Hierarchy** - Broken hierarchy design smell for `BaseResource`. Only the `request`, `principal` variables and the `authenticate` function were being used by most of the inherited classes. The `hasBaseFunction` and `checkBaseFunction` functions weren't being used. `appKey` is possibly not being used anywhere.

Hence we refactored `BaseResource` into

- `BaseResource`

```

public abstract class BaseResource {
    /**
     * Injects the HTTP request.
     */
    @Context
    protected HttpServletRequest request;

    /**
     * Principal of the authenticated user.
     */
    protected IPrincipal principal;

    /**
     * This method is used to check if the user is authenticated.
     *
     * @return True if the user is authenticated and not anonymous
     */
    protected boolean authenticate() {
        Principal principal = (Principal) request.getAttribute(TokenBasedSecurityFilter.PRINCIPAL_ATTRIBUTE);
        if (principal != null && principal instanceof IPrincipal) {
            this.principal = (IPrincipal) principal;
            return !this.principal.isAnonymous();
        } else {
            return false;
        }
    }
}

```

- `ExtendedBaseResource` - `AppResource` and `UserResource` inherit from this

```

public abstract class ExtendedBaseResource {
    /**
     * Injects the HTTP request.
     */
    @Context
    protected HttpServletRequest request;

    /**
     * Application key.
     */
    @QueryParam("app_key")
    protected String appKey;

    /**
     * Principal of the authenticated user.
     */
    protected IPrincipal principal;

    /**
     * This method is used to check if the user is authenticated.
     *
     * @return True if the user is authenticated and not anonymous
     */
    protected boolean authenticate() {
        Principal principal = (Principal) request.getAttribute(TokenBasedSecurityFilter.PRINCIPAL_ATTRIBUTE);
        if (principal != null && principal instanceof IPrincipal) {
            this.principal = (IPrincipal) principal;
            return !this.principal.isAnonymous();
        } else {
            return false;
        }
    }
}

```



```

/**
 * Checks if the user has a base function. Throw an exception if the check fails.
 *
 * @param baseFunction Base function to check
 * @throws JSONException
 */
protected void checkBaseFunction(BaseFunction baseFunction) throws JSONException {
    if (!hasBaseFunction(baseFunction)) {
        throw new ForbiddenClientException();
    }
}

/**
 * Checks if the user has a base function.
 *
 * @param baseFunction Base function to check
 * @return True if the user has the base function
 * @throws JSONException
 */
protected boolean hasBaseFunction(BaseFunction baseFunction) throws JSONException {
    if (principal == null || !(principal instanceof UserPrincipal)) {
        return false;
    }
    Set<String> baseFunctionSet = ((UserPrincipal) principal).getBaseFunctionSet();
    return baseFunctionSet != null && baseFunctionSet.contains(baseFunction.name());
}
}

```

- <https://github.com/Raghav010/books-refac/pull/11>
- Branch: `baseresource-inheritance`



created ExtendedBaseResource, fixed the broken hierarchy in BaseResource - merge
1

#11 by Raghav010 was merged yesterday

- **LLM Refactoring**
 - <https://chat.openai.com/share/cff00a0b-06af-4634-8e46-41d8930a8139>
 - The output generated by ChatGPT is initially bad since it doesn't even create a new class. It just renames the extra functions (`hasBaseFunction` and `checkBaseFunction`) but doesn't remove them. It didn't seem to understand what broken hierarchy was essentially. When I asked it for two classes since its broken hierarchy, it refactored and made two classes (`AuthenticatedResource` and `BaseResource`), with one inheriting the other. This was satisfactory, but in this particular scenario, it was too deep an inheritance. Since both the abstract classes themselves aren't instantiated but `AuthenticatedResource` extends `BaseResource`. **Mediocre to good.**

TagDao.java

- **Multifaceted Abstraction** - `TagDao` seems to be performing functions for different classes. There are two classes `Tag` and `UserBookTag` and `TagDao` has functions pertaining to creating, deleting and getting both class objects. Hence logically `TagDao` should be split into

`TagDao` - only `Tag` related functions

- `getId()`
- `getByUserId()`
- `create()`
- `getByName()`
- `getByTagId()`
- `delete()`
- `findByName()` - unused

```

/**
 * Tag DAO.

```

```

*
* @author bgamard
*/
public class TagDao {
    /**
     * Gets a tag by its ID.
     *
     * @param id Tag ID
     * @return Tag
     */
    public Tag getById(String id) {
        EntityManager em = ThreadLocalContext.get().getEntityManager();
        try {
            return em.find(Tag.class, id);
        } catch (NoResultException e) {
            return null;
        }
    }

    /**
     * Returns the list of all tags.
     *
     * @return List of tags
     */
    @SuppressWarnings("unchecked")
    public List<Tag> getByUserId(String userId) {
        EntityManager em = ThreadLocalContext.get().getEntityManager();
        Query q = em.createQuery("select t from Tag t where t.userId = :userId and t.deleteDate is null");
        q.setParameter("userId", userId);
        return q.getResultList();
    }

    /**
     * Creates a new tag.
     *
     * @param tag Tag
     * @return New ID
     * @throws Exception
     */
    public String create(Tag tag) {
        // Create the UUID
        tag.setId(UUID.randomUUID().toString());

        // Create the tag
        EntityManager em = ThreadLocalContext.get().getEntityManager();
        tag.setCreateDate(new Date());
        em.persist(tag);

        return tag.getId();
    }

    /**
     * Returns a tag by name.
     * @param userId User ID
     * @param name Name
     * @return Tag
     */
    public Tag getByName(String userId, String name) {
        EntityManager em = ThreadLocalContext.get().getEntityManager();
        Query q = em.createQuery("select t from Tag t where t.name = :name and t.userId = :userId and t.deleteDate is null");
        q.setParameter("userId", userId);
        q.setParameter("name", name);
        try {

```

```

        return (Tag) q.getSingleResult();
    } catch (NoResultException e) {
        return null;
    }
}

/**
 * Returns a tag by ID.
 * @param userId User ID
 * @param tagId Tag ID
 * @return Tag
 */
public Tag getByTagId(String userId, String tagId) {
    EntityManager em = ThreadLocalContext.get().getEntityManager();
    Query q = em.createQuery("select t from Tag t where t.id = :tagId and t.userId = :userId and t.deleteDate is null");
    q.setParameter("userId", userId);
    q.setParameter("tagId", tagId);
    try {
        return (Tag) q.getSingleResult();
    } catch (NoResultException e) {
        return null;
    }
}

/**
 * Deletes a tag.
 *
 * @param tagId Tag ID
 */
public void delete(String tagId) {
    EntityManager em = ThreadLocalContext.get().getEntityManager();

    // Get the tag
    Query q = em.createQuery("select t from Tag t where t.id = :id and t.deleteDate is null");
    q.setParameter("id", tagId);
    Tag tagDb = (Tag) q.getSingleResult();

    // Delete the tag
    Date dateNow = new Date();
    tagDb.setDeleteDate(dateNow);

    UserBookTagDao UBTagDao = new UserBookTagDao();
    UBTagDao.delete(tagId);
}

/**
 * Search tags by name.
 *
 * @param name Tag name
 * @return List of found tags
 */
@SuppressWarnings("unchecked")
public List<Tag> findByName(String userId, String name) {
    EntityManager em = ThreadLocalContext.get().getEntityManager();
    Query q = em.createQuery("select t from Tag t where t.name like :name and t.userId = :userId and t.deleteDate is null");
    q.setParameter("userId", userId);
    q.setParameter("name", "%" + name + "%");
    return q.getResultList();
}
}

```

`UserBookTagDao` - only `UserBookTag` related functions

- `updateTagList()` - creates tags for UserBooks

- `getByUserBookId()` - gets UserBookTags
- `delete()` - deletes UserBookTags - used in `delete()` of `TagDao`

```
public class UserBookTagDao {

    /**
     * Update tags on a user book. or creates a userbooktag
     *
     * @param userBookId
     * @param tagIdSet
     */
    public void updateTagList(String userBookId, Set<String> tagIdSet) {
        // Delete old tag links
        EntityManager em = ThreadLocalContext.get().getEntityManager();
        Query q = em.createQuery("delete UserBookTag bt where bt.userBookId = :userBookId");
        q.setParameter("userBookId", userBookId);
        q.executeUpdate();

        // Create new tag links
        for (String tagId : tagIdSet) {
            UserBookTag userBookTag = new UserBookTag();
            userBookTag.setId(UUID.randomUUID().toString());
            userBookTag.setUserBookId(userBookId);
            userBookTag.setTagId(tagId);
            em.persist(userBookTag);
        }
    }

    /**
     * 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_USR");
        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("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;
    }

    /**
     * Deletes a user book tag.
     */
}
```

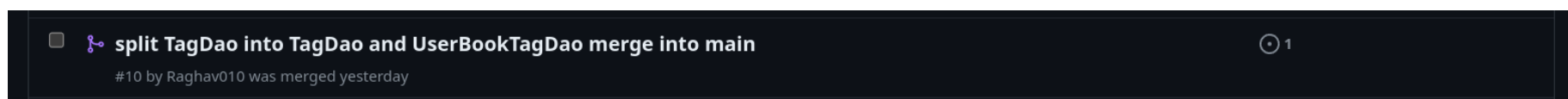
```

    *
    * @param tagId Tag ID
    */
    public void delete(String tagId) {
        EntityManager em = ThreadLocalContext.get().getEntityManager();

        // Delete linked data
        Query q = em.createQuery("delete UserBookTag bt where bt.tagId = :tagId");
        q.setParameter("tagId", tagId);
        q.executeUpdate();
    }
}

```

- <https://github.com/Raghav010/books-refac/pull/10>
- Branch: `tag-dao-split`

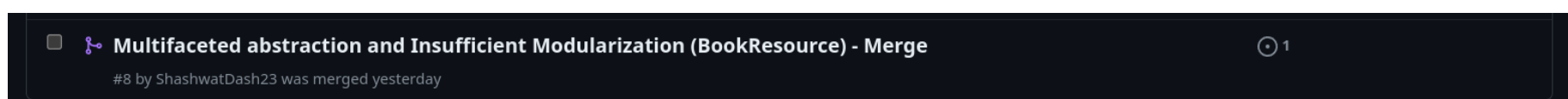


- **LLM Refactoring**
 - <https://chat.openai.com/share/bd44116f-e28a-41f6-b524-2d2721e3418f>
 - The output produced by chatGPT is pretty satisfactory. But even this required prompting GPT twice for it to create the other `UserBookTagDao` class. The other class it creates is on point, except it misses the delete function. It essentially did a high-level function-wise split of the original `TagDao` class. **Excellent**

BookResource.java

- **Multifaceted Abstraction** - The `BookResource` class exhibits multifaceted abstraction by combining functionality for both `Book` and `UserBook` operations, resulting in a codebase exceeding 600 lines with considerable complexity. This approach compromises readability and maintainability. To mitigate these issues, we introduced a helper class `HelperBookResource` dedicated to handling book-related operations such as adding and updating. By offloading these responsibilities to the helper function, `BookResource` now focuses solely on managing `UserBook` functionalities. This restructuring enhances code organization and clarity, promoting better maintainability and readability.

- <https://github.com/Raghav010/books-refac/pull/8>
- Branch: `multifaceted-abstraction-BookResource`



For example, the add function has changed from:

```

@PUT
@Produces(MediaType.APPLICATION_JSON)
public Response add(
    @FormParam("isbn") String isbn) throws JSONException {
    if (!authenticate()) {
        throw new ForbiddenClientException();
    }

    // Validate input data
    ValidationUtil.validateRequired(isbn, "isbn");

    // Fetch the book
    BookDao bookDao = new BookDao();
    Book book = bookDao.getByIsbn(isbn);
    if (book == null) {
        // Try to get the book from a public API
        try {

```



```

        book = AppContext.getInstance().getBookDataService().searchBook(isbn);
    } catch (Exception e) {
        throw new ClientException("BookNotFound", e.getCause().getMessage(), e);
    }

    // Save the new book in database
    bookDao.create(book);
}

// Create the user book if needed
UserBookDao userBookDao = new UserBookDao();
UserBook userBook = userBookDao.getByBook(book.getId(), principal.getId());
if (userBook == null) {
    userBook = new UserBook();
    userBook.setUserId(principal.getId());
    userBook.setBookId(book.getId());
    userBook.setCreateDate(new Date());
    userBookDao.create(userBook);
} else {
    throw new ClientException("BookAlreadyAdded", "Book already added");
}

JSONObject response = new JSONObject();
response.put("id", userBook.getId());
return Response.ok().entity(response).build();
}

```

To:

```

public Response add(
    @FormParam("isbn") String isbn) throws JSONException {
    if (!authenticate()) {
        throw new ForbiddenClientException();
    }

    // Validate input data
    ValidationUtil.validateRequired(isbn, "isbn");

    // Get the book from helper
    HelperBookResource helperBookResource = new HelperBookResource();
    Book book = helperBookResource.add(isbn);

    // Create the user book if needed
    UserBookDao userBookDao = new UserBookDao();
    UserBook userBook = userBookDao.getByBook(book.getId(), principal.getId());
    if (userBook == null) {
        userBook = createUserBook(book, userBookDao);
    } else {
        throw new ClientException("BookAlreadyAdded", "Book already added");
    }

    System.out.println("Done!!!");

    JSONObject response = new JSONObject();
    response.put("id", userBook.getId());
    return Response.ok().entity(response).build();
}

```

- **LLM Refactoring**

- <https://chat.openai.com/share/778611a5-91cd-43f1-8ed4-47aeb5df298b>
- **Code readability:**

The LLM-generated code suggested a separation of responsibilities by extracting functionalities into three separate classes: `BookService`, `UserBookService`, and `TagService`. This modular approach can enhance code readability by promoting a more focused and cohesive design. However, it may introduce higher coupling in the `BookResource` class, as it now depends on multiple other classes.

On the other hand, the manual refactoring method improves readability by delegating the core book functionality to a helper function, while keeping the `BookResource` class focused on handling user book-related functionality. This approach simplifies the responsibilities of the `BookResource` class, making it more concise and easier to understand.

Code Maintainability:

In the AI-generated code, breaking down functionality into separate classes can make things easier to maintain in the long run. Each class has a specific responsibility, so if we need to make changes or fix bugs, it would be easier to find.

Both the LLM-generated and manually refactored versions adhere to the single responsibility principle, which promotes code clarity and maintainability by ensuring that each class or component has a clear and focused purpose. By dividing functionality into different classes, both approaches improve clarity compared to the original code, making it easier to understand and modify.

AuthenticationTokenDao.java

Common with TagDao.java, UserAppDao.java, UserBookDao.java, UserContactDao.java, UserDao.java

- **Missing Abstraction** - We observe that each of the DAOs mentioned above have `create` and `delete` methods. To provide an abstraction for these DAOs which did not exist, an interface `Dao` was created in the `books-core/src/main/java/com/sismics/books/core/interfaces` directory.
- [Created DAO interface with create and delete methods - merge by Raghav010 · Pull Request #12 · Raghav010/books-refac \(github.com\)](#).
- Branch name: `create-delete-interface`

```
/**
 * Basic functionalities related to DAO - Data Access Object
 * @param <T>
 */
public interface Dao<T> {
    /**
     * creates an object (specified by T)
     * @param obj
     * @throws Exception
     * @return String
     */
    String create(T obj) throws Exception;

    /**
     * deletes an object instance
     * @param id
     * @throws Exception
     */
    void delete(String id) throws Exception;
}
```

UserResource.java

- **Missing Abstraction** - Decomposed `UserResource` into 3 helper functions - a `createUser` helper function, a validation helper function, and a helper function to get session token.
 - <https://github.com/Raghav010/books-refac/pull/18>
 - Branch Name: `16-insufficient-modularization-userresource-class`

 **16 insufficient modularization userresource class - merge**
#18 by Raghav010 was merged 1 hour ago

- **LLM Refactoring**
 - <https://chat.openai.com/share/961a9d64-69d0-4d86-86c2-66fbd56ed2ea>

- The refactoring approach taken by the LLM involves decomposing the monolithic `UserResource` class into separate classes, each catering to distinct responsibilities such as user registration, authentication, session management, and user information handling. This strategy aligns with the principles of modularity and Single Responsibility Principle (SRP) in software engineering.

While the LLM's solution offers clear delineation of responsibilities and enhances code organization, it introduces potential drawbacks when considering the system's architectural constraints and coupling concerns. Notably, the method-bound nature of these classes to specific RESTful endpoints (Paths) complicates the process of further splitting the classes without necessitating frontend modifications. This could introduce friction when attempting to evolve the system's architecture in the future.

Furthermore, the creation of additional classes may inadvertently heighten coupling within the system, particularly if these classes rely heavily on shared resources or tightly-coupled interactions. This could impede the system's flexibility and hinder its ability to adapt to changing requirements or technological advancements.

In contrast, the initial approach of consolidating helper functions within the `UserResource` class, supplemented by a dedicated helper class, represents an alternative strategy to improve code cohesiveness. By encapsulating related functionalities within a cohesive unit, this approach mitigates the risk of excessive class proliferation and potential coupling issues. However, it may sacrifice some clarity and separation of concerns compared to the modular approach.

In summary, while the LLM's refactoring strategy offers advantages in terms of code organization and adherence to SRP, it necessitates careful consideration of architectural constraints and potential coupling implications. Balancing modularity with architectural flexibility is crucial to ensure the long-term maintainability and extensibility of the codebase.

Constants.java

- **Leaky Encapsulation** - In our original code, the `Constants` class had a public constant `DEFAULT_ADMIN_PASSWORD`, which meant that any class in the application could access and potentially modify this sensitive information. This violated the principle of encapsulation, as the password should be tightly controlled and accessed only within the appropriate context. By moving the `DEFAULT_ADMIN_PASSWORD` variable from the `Constants` class to the `UserResource` class and making it private, we addressed the issue of leaking encapsulation. Now, the password is encapsulated within the `UserResource` class, restricting access to it from other parts of the application. This ensures that the sensitive information is only accessible within the context where it's needed, improving security and maintaining better control over the codebase.
 - <https://github.com/Raghav010/books-refac/pull/14>
 - Branch: `13-leaky-encapsulation-in-constantsjava`



For example, the add function has changed from:

```
package com.sismics.books.core.constant;

/**
 * Application constants.
 *
 * @author jtremeaux
 */
public class Constants {
    /**
     * Default locale.
     */
    public static final String DEFAULT_LOCALE_ID = "en";

    /**
     * Default timezone ID.
     */
    public static final String DEFAULT_TIMEZONE_ID = "Europe/London";

    /**
     * Default theme ID.
     */
    public static final String DEFAULT_THEME_ID = "default.less";
}
```

```

/**
 * Administrator's default password ("admin").
 */
public static final String DEFAULT_ADMIN_PASSWORD = "$2a$05$6Ny3Tjrw3aVAL1or2SlcR.fhuDgPKp5jp.P9fBXwVl

/**
 * Default generic user role.
 */
public static final String DEFAULT_USER_ROLE = "user";
}

```

and

```

/**
 * Returns the information about the connected user.
 *
 * @return Response
 * @throws JSONException
 */
@GET
@Produces(MediaType.APPLICATION_JSON)
public Response info() throws JSONException {
    JSONObject response = new JSONObject();
    if (!authenticate()) {
        response.put("anonymous", true);

        // Check if admin has the default password
        UserDao userDao = new UserDao();
        User adminUser = userDao.getById("admin");
        if (adminUser != null && adminUser.getDeleteDate() == null) {
            response.put("is_default_password", Constants.DEFAULT_ADMIN_PASSWORD.equals(adminUser.getI

        }
    } else {
        response.put("anonymous", false);
        UserDao userDao = new UserDao();
        User user = userDao.getById(principal.getId());
        response.put("username", user.getUsername());
        response.put("email", user.getEmail());
        response.put("theme", user.getTheme());
        response.put("locale", user.getLocaleId());
        response.put("first_connection", user.isFirstConnection());
        JSONArray baseFunctions = new JSONArray(((UserPrincipal) principal).getBaseFunctionSet());
        response.put("base_functions", baseFunctions);
        response.put("is_default_password", hasBaseFunction(BaseFunction.ADMIN) && Constants.DEFAULT_

    }

    return Response.ok().entity(response).build();
}

```

To:

```

package com.sismics.books.core.constant;

/**
 * Application constants.
 *
 * @author jtremeaux
 */
public class Constants {
    /**
     * Default locale.
     */
    public static final String DEFAULT_LOCALE_ID = "en";
}

```

```

/**
 * Default timezone ID.
 */
public static final String DEFAULT_TIMEZONE_ID = "Europe/London";

/**
 * Default theme ID.
 */
public static final String DEFAULT_THEME_ID = "default.less";

/**
 * Default generic user role.
 */
public static final String DEFAULT_USER_ROLE = "user";

public static final int MIN_USERNAME_LEN = 3;
public static final int MAX_USERNAME_LEN = 50;

public static final int MIN_EMAIL_LEN = 3;
public static final int MAX_EMAIL_LEN = 50;
public static final boolean EMAIL_NULLABLE = true;

public static final int MIN_PWD_LEN = 8;
public static final int MAX_PWD_LEN = 50;
public static final boolean PWD_NULLABLE = true;

public static final boolean LOCALE_ID_NULLABLE = true;

public static final boolean THEME_ID_NULLABLE = true;
}

```

and

```

/**
 * Administrator's default password ("admin").
 */
private static final String DEFAULT_ADMIN_PASSWORD = "$2a$05$6Ny3TjrW3aVAL1or2S1cR.fhuDgPKp5jp.P9fBXwV

/**
 * Returns the information about the connected user.
 *
 * @return Response
 * @throws JSONException
 */
@GET
@Produces(MediaType.APPLICATION_JSON)
public Response info() throws JSONException {
    JSONObject response = new JSONObject();
    if (!authenticate()) {
        response.put("anonymous", true);

        // Check if admin has the default password
        UserDao userDao = new UserDao();
        User adminUser = userDao.getById("admin");
        if (adminUser != null && adminUser.getDeleteDate() == null) {
            response.put("is_default_password", DEFAULT_ADMIN_PASSWORD.equals(adminUser.getPassword()));
        }
    } else {
        response.put("anonymous", false);
        UserDao userDao = new UserDao();
        User user = userDao.getById(principal.getId());
        response.put("username", user.getUsername());
    }
}

```

```

        response.put("email", user.getEmail());
        response.put("theme", user.getTheme());
        response.put("locale", user.getLocaleId());
        response.put("first_connection", user.isFirstConnection());
        JSONArray baseFunctions = new JSONArray(((UserPrincipal) principal).getBaseFunctionSet());
        response.put("base_functions", baseFunctions);
        response.put("is_default_password", hasBaseFunction(BaseFunction.ADMIN) && DEFAULT_ADMIN_PASSWORD);
    }

    return Response.ok().entity(response).build();
}

```

- **LLM Refactoring**

- <https://chat.openai.com/share/bc79de2a-caec-48c4-b871-bc3e7dd31aa2>
- ChatGPT's response is similar to ours, but we initialized the `DEFAULT_ADMIN_PASSWORD` in `Constants` and `UserResource`, respectively. Both methods have their pros and cons.

Pros of our method:

1. **Encapsulation:** Our approach encapsulates the password directly within the class where it's used (`UserResource`). This makes it clear that the password is specific to the functionality of the `UserResource` class and doesn't need to be accessed from outside.
2. **Reduced Dependency:** By keeping the password within the `UserResource` class, you reduce the dependency on the `Constants` class. This can be beneficial if the `Constants` class is meant to hold more general constants and you want to minimize coupling between different parts of your codebase.
3. **Simplicity:** Our approach may be more straightforward, especially if there's only one class that needs access to this specific constant. There's no need to introduce additional methods or dependencies for accessing the password.

Cons of our method:

1. **Reusability:** If there's a possibility that other classes might need to access this constant in the future, storing it in a separate `Constants` class with a getter method provides a more reusable and maintainable solution.

Role.java and RoleBasedFunction.java

- **Unused Abstraction** - In your codebase, the classes `Role` and `RoleBasedFunction` were identified as unused abstractions, indicating that they were unnecessary in the context of the application. Additionally, the `RoleBasedFunctionDao` class was directly used by the `User` class for assigning roles to users, bypassing the need for `Role` and `RoleBasedFunction`.

- <https://github.com/Raghav010/books-refac/pull/15>
- Branch Name: `6-remove-class-role-from-rolejava`



- This situation points to a design smell of unnecessary complexity and potential redundancy. To address this issue, you correctly identified and removed the unused `Role` and `RoleBasedFunction` classes, simplifying the codebase and reducing unnecessary complexity. Additionally, since the `User` class directly used `RoleBasedFunctionDao` without relying on `Role` and `RoleBasedFunction`, removing these classes did not impact the functionality of the code. This refactoring improves the maintainability and readability of the codebase by eliminating unnecessary abstractions and ensuring that the code accurately reflects the system's functionality.

Role

```

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

import com.google.common.base.Objects;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.Table;
import java.util.Date;

```

```

/**
 * Role (set of base functions).
 *
 * @author jtremeaux
 */
@Entity
@Table(name = "T_ROLE")
public class Role {
    /**
     * Role ID.
     */
    @Id
    @Column(name = "ROL_ID_C", length = 36)
    private String id;

    /**
     * Role name.
     */
    @Column(name = "ROL_NAME_C", nullable = false, length = 50)
    private String name;

    /**
     * Creation date.
     */
    @Column(name = "ROL_CREATEDATE_D", nullable = false)
    private Date createDate;

    /**
     * Deletion date.
     */
    @Column(name = "ROL_DELETEDATE_D")
    private Date deleteDate;

    /**
     * Getter of id.
     *
     * @return id
     */
    public String getId() {
        return id;
    }

    /**
     * Setter of id.
     *
     * @param id id
     */
    public void setId(String id) {
        this.id = id;
    }

    /**
     * Getter of name.
     *
     * @return name
     */
    public String getName() {
        return name;
    }

    /**
     * Setter of name.
     *
     * @param name name

```



```

        */
    public void setName(String name) {
        this.name = name;
    }

    /**
     * Getter of createDate.
     *
     * @return createDate
     */
    public Date getCreateDate() {
        return createDate;
    }

    /**
     * Setter of createDate.
     *
     * @param createDate createDate
     */
    public void setCreateDate(Date createDate) {
        this.createDate = createDate;
    }

    /**
     * Getter of deleteDate.
     *
     * @return deleteDate
     */
    public Date getDeleteDate() {
        return deleteDate;
    }

    /**
     * Setter of deleteDate.
     *
     * @param deleteDate deleteDate
     */
    public void setDeleteDate(Date deleteDate) {
        this.deleteDate = deleteDate;
    }

    @Override
    public String toString() {
        return Objects.toStringHelper(this)
            .add("id", id)
            .add("name", name)
            .toString();
    }
}

```

RoleBasedFunction

```

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

import com.google.common.base.Objects;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.Table;
import java.util.Date;

/**

```

```

* Role base function.
*
* @author jtremeaux
*/
@Entity
@Table(name = "T_ROLE_BASE_FUNCTION")
public class RoleBaseFunction {
    /**
     * Role base function ID.
     */
    @Id
    @Column(name = "RBF_ID_C", length = 36)
    private String id;

    /**
     * Role ID.
     */
    @Column(name = "RBF_IDROLE_C", nullable = false, length = 36)
    private String roleId;

    /**
     * Base function ID.
     */
    @Column(name = "RBF_IDBASEFUNCTION_C", nullable = false, length = 36)
    private String baseFunctionId;

    /**
     * Creation date.
     */
    @Column(name = "RBF_CREATEDATE_D", nullable = false)
    private Date createDate;

    /**
     * Deletion date.
     */
    @Column(name = "RBF_DELETEDATE_D")
    private Date deleteDate;

    /**
     * Getter of id.
     *
     * @return id
     */
    public String getId() {
        return id;
    }

    /**
     * Setter of id.
     *
     * @param id id
     */
    public void setId(String id) {
        this.id = id;
    }

    /**
     * Getter of roleId.
     *
     * @return roleId
     */
    public String getRoleId() {
        return roleId;
    }
}

```

```

/**
 * Setter of roleId.
 *
 * @param roleId roleId
 */
public void setRoleId(String roleId) {
    this.roleId = roleId;
}

/**
 * Getter of baseFunctionId.
 *
 * @return baseFunctionId
 */
public String getBaseFunctionId() {
    return baseFunctionId;
}

/**
 * Setter of baseFunctionId.
 *
 * @param baseFunctionId baseFunctionId
 */
public void setBaseFunctionId(String baseFunctionId) {
    this.baseFunctionId = baseFunctionId;
}

/**
 * Getter of createDate.
 *
 * @return createDate
 */
public Date getCreateDate() {
    return createDate;
}

/**
 * Setter of createDate.
 *
 * @param createDate createDate
 */
public void setCreateDate(Date createDate) {
    this.createDate = createDate;
}

/**
 * Getter of deleteDate.
 *
 * @return deleteDate
 */
public Date getDeleteDate() {
    return deleteDate;
}

/**
 * Setter of deleteDate.
 *
 * @param deleteDate deleteDate
 */
public void setDeleteDate(Date deleteDate) {
    this.deleteDate = deleteDate;
}

```

```

@Override
public String toString() {
    return Objects.toStringHelper(this)
        .add("id", id)
        .add("userId", roleId)
        .add("baseFunctionId", baseFunctionId)
        .toString();
}
}

```

Relevant Snippet from RoleBasedFunctionDao

```

public class RoleBaseFunctionDao {
    /**
     * Find the set of base functions of a role.
     *
     * @param roleId Role ID
     * @return Set of base functions
     */
    @SuppressWarnings("unchecked")
    public Set<String> findByRoleId(String roleId) {
        EntityManager em = ThreadLocalContext.get().getEntityManager();
        StringBuilder sb = new StringBuilder("select rbf.RBF_IDBASEFUNCTION_C from T_ROLE_BASE_FUNC");
        sb.append(" where rbf.RBF_IDROLE_C = :roleId and rbf.RBF_DELETEDATE_D is null");
        sb.append(" and r.ROL_ID_C = rbf.RBF_IDROLE_C and r.ROL_DELETEDATE_D is null");
        Query q = em.createNativeQuery(sb.toString());
        q.setParameter("roleId", roleId);
        return Sets.newHashSet(q.getResultList());
    }
}

```

As we can see, it takes `String roleId` from the `User` class and does not need `Role` and `RoleBasedFunction`.

Overall LLM Refactoring Conclusions

- It is observed that LLM-based refactoring is less time-consuming and more effective when it comes to removing relatively straightforward smells, such as long methods and magic numbers. It can easily handle the extraction of similar code from two different methods as it can identify the visible pattern in the methods provided to it. LLM-based refactoring is thus helpful in effectively modularizing the provided code and ensuring better readability.
- In some cases, manual intervention may be required to make sure it does not end up unnecessarily modularizing the code provided, as this would affect the understandability of the refactored code.
- Looks like GPT is good at understanding and refactoring small snippets of code with straightforward logic over long pieces of code with complex relationships and logic.
- Overall, the LLM generated a pretty satisfactory response, some of which could aid in refactoring manually. LLM would struggle to completely deal with refactoring this completely due to its large size. It performs better for smaller code snippets.

Code Smells

BookDataService.java

- In both `searchBookWithGoogle` and `searchBookWithOpenLibrary` functions there exists a part of the function which is responsible for connecting to the service for searching books. This process of opening a connection to the URL and getting the stream from the url is abstracted away into the function `getStream`. This function is used by both functions for connecting to their respective search urls.
 - <https://github.com/Raghav010/books-refac/commit/4c9b49b6f28673780375e3559696f7fc72e7e1c9>
 - **LLM Refactoring**
 - <https://chat.openai.com/share/bde1755c-dd0a-48ab-8e34-5340a2189f24>

- Due to the relatively simple nature of extraction, the ChatGPT generated refactoring is pretty similar and just as accurate compared to the manually refactored code

Refactored UserAppDao, BookDataService, EMF - extracted methods4c9b49b

kapilrk-04 committed 2 days ago

- Removed redundant variable `InputStream`
 - <https://github.com/Raghav010/books-refac/commit/0432a26f08f63e95b74ab3fcb4e6a1ddf95b0345>
 - **LLM Refactoring**
 - implemented it correctly as expected. **Excellent**

removed redundant variables from BookDataService.java0432a26

Raghav010 committed 1 minute ago

EMF.java

- This code smell extracts `getProps()` out of the `getEntityManagerProperties()`, since the `getProps()` functionaility is used in multiple areas.
 - <https://github.com/Raghav010/books-refac/commit/4c9b49b6f28673780375e3559696f7fc72e7e1c9>
 - **LLM Refactoring**
 - <https://chat.openai.com/share/d92fcc88-ef6c-469c-985f-205368b36403>
 - While ChatGPT extracts `getProps()` out of the `getEntityManagerProperties()` method, it performs unnecessary modularization by performing another extraction on the same code, with its `loadPropertiesFromFile()` method.

Refactored UserAppDao, BookDataService, EMF - extracted methods4c9b49b

kapilrk-04 committed 2 days ago

UserAppDao.java

- A list of `UserAppDto` are created in functions `findConnectedByUserId` and `findByAppId`. This repeated creation of `List<UserAppDto>` is abstracted away into a function `getUserAppDtoList`
 - <https://github.com/Raghav010/books-refac/commit/4c9b49b6f28673780375e3559696f7fc72e7e1c9>
 - **LLM Refactoring**
 - <https://chat.openai.com/share/6f5fc15f-42ec-4d7c-9905-f877a1ca35fc>
 - Due to the relatively simple nature of extraction, the ChatGPT generated refactoring is pretty similar and just as accurate compared to the manually refactored code.

Refactored UserAppDao, BookDataService, EMF - extracted methods4c9b49b

kapilrk-04 committed 2 days ago

BookImportAsyncListener.java

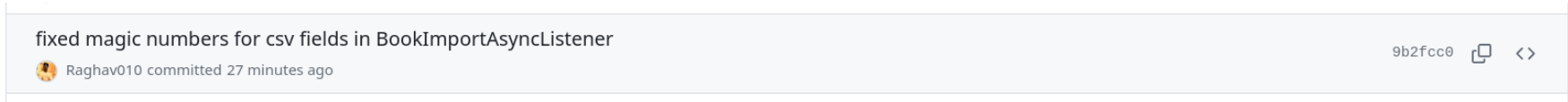
- **Long Method** Refactored the `on` function, to make it shorter. Delegated responsibilities inside the function to
 - `getOrCreateBook` - creates a book if it doesnt already exist from the isbn
 - `getOrCreateUserBook` - creates a userBook if it doesnt already exist, assigns the book to a user
 - `createTags` - creates tags for the user from the tags on the book
 - `createUserBookTags` - assigns the previously created tags to the user book
 - <https://github.com/Raghav010/books-refac/commit/18e007c2f85481b1d315564bebaaf96eb3ead87b>

[refactored on\(\) method of BookImportAsyncListener into smaller methods](#)18e007c

Raghav010 committed 1 refactored on() method of BookImportAsyncListener into smaller methods

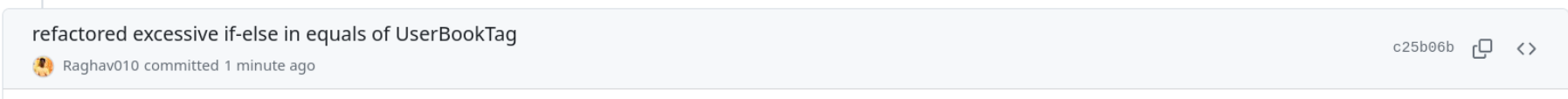
- **LLM Refactoring**
 - <https://chat.openai.com/share/7c58558f-8003-4d6e-8afa-ddd1ed095796>

- Here gpt separated out sufficient number of functions from the big `on` function. It cause unnecessary modularization, there were way too many helper functions being used in `on` . This would just cause extra confusion, and decrease readability due to small things (in a logical sense, not lines of code) also being decomposed into separate functions. In manual refactoring we can see that only sections with enough operational logic are decomposed into separate functions. Gpt also missed out on making `createUserBookTags` . The four functions in manual refactoring follow SRP, since each function takes care of a particular class related operations.
- **Magic Numbers** Created magic numbers for standard re-occurring variables , which were mainly the field numbers(column numbers) of the goodreads csv export file. Field for isbn, bookId etc.
 - **LLM Refactoring**
 - <https://chat.openai.com/share/7c58558f-8003-4d6e-8afa-ddd1ed095796>
 - It created another class for all the magic variables which is unnecessary at this level. The complexity of the situation doesnt warrant another class. It also missed out on the BookId field
 - <https://github.com/Raghav010/books-refac/commit/9b2fcc068f4e76940999d6b5009405ca363639c2>



UserBookTag.java

- Removed excessive if-else in `equals` function, shortened it
 - <https://github.com/Raghav010/books-refac/commit/c25b06bc6d57356219aa2c36fe7ad203e16bd326>
 - **LLM Refactoring**
 - <https://chat.openai.com/share/f96949c9-5420-461a-b15d-bbc27ab5bf6f>
 - Gave a really good solution. It used the `java.util.Objects.equal()` method to shorten quite a bit of code. It did take a few tries for it to understand that another `Objects` (from `com.google`)class was conflicting with the `java.util.Objects` it was saying to import. **Excellent**



UserBook.java

- Removed excessive if-else in `equals` function, shortened it
 - <https://github.com/Raghav010/books-refac/commit/12e3587a9c4e912523576a31e6f58dabb9ee8041>



Code Metrics

Lines of Code / Non-Comment Source Statements (LOC)

This is the number of lines of code in the project, not including comments and headers. Some versions of this metric also exclude blank lines.

LOC is a metric that is easy to understand even for people who are not technically inclined. It is also easy to compute and digest.

Quality

- **Complexity:** Generally, a larger number of lines of code can indicate increased complexity within the software. Higher complexity often correlates with a higher potential for bugs and errors, as it becomes more challenging to understand, debug, and maintain the codebase.
- **Code Clarity:** Large code bases with excessive lines of code can be more difficult to read and comprehend, leading to decreased code clarity. This lack of clarity can introduce ambiguity and increase the likelihood of introducing bugs during development.

- **Testing Burden:** More lines of code typically require more extensive testing efforts to ensure adequate coverage and identify potential defects. This can result in increased testing time and resources required, impacting the overall quality assurance process.

Maintainability

- **Code Maintenance:** A larger codebase with more lines of code can increase the time and effort required for ongoing maintenance activities such as adding features, fixing bugs, and updating the software to meet new requirements. This is because developers need to navigate through more code and understand its interactions to make changes safely.
- **Documentation:** Maintaining thorough and up-to-date documentation becomes more critical as the size of the codebase grows. Without adequate documentation, understanding and modifying the code become increasingly challenging, leading to potential maintenance issues.

Performance:

- **Execution Time:** While the number of lines of code itself does not directly correlate with performance, overly complex or inefficient code structures can impact runtime performance. Redundant computations, excessive memory usage, and inefficient algorithms can lead to slower execution times, particularly in resource-constrained environments.
- **Startup Time:** In applications where startup time is critical, such as web servers or mobile apps, a larger codebase can lead to longer initialization times. This can affect user experience and perception of performance.
- **Memory Usage:** Larger code bases may require more memory to execute, especially if the code includes extensive libraries or dependencies. This can be a concern in environments with limited memory resources, such as embedded systems or mobile devices.

Cyclomatic Complexity

Cyclomatic complexity is a software metric used to indicate the complexity of a program. It is typically calculated by counting decision points within the code, such as branches (if statements, loops, etc.) and measuring the number of unique paths that can be taken through the codebase.

The cyclomatic complexity of a structured program is defined with reference to the control-flow graph of the program, a directed graph containing the basic blocks of the program, with an edge between two basic blocks if control may pass from the first to the second.

The complexity M is then defined as

$$M = E - N + 2 P$$

where

E = the number of edges of the graph.

N = the number of nodes of the graph.

P = the number of connected components.

Software Quality

- **Bug Proneness:** High cyclomatic complexity often correlates with an increased likelihood of defects and bugs. This is because complex code paths are harder to understand, test, and maintain, making it more likely for developers to overlook potential issues.
- **Readability:** Complex code with high cyclomatic complexity can be challenging to understand, even for experienced developers. This lack of readability can hinder code reviews, collaboration, and overall comprehension of the software.
- **Code Quality Metrics:** Higher complexity values can indicate areas of the codebase that require refactoring or optimization to improve maintainability and reduce the risk of defects.

Maintainability:

- **Code Maintenance:** High cyclomatic complexity makes code maintenance more difficult and time-consuming. Developers must navigate through numerous code paths, increasing the risk of introducing new bugs or unintended side effects when making changes.
- **Refactoring Challenges:** Refactoring complex code with high cyclomatic complexity can be challenging. Breaking down complex logic into smaller, more manageable components requires careful planning and may involve restructuring large portions of the codebase.
- **Documentation:** Maintaining clear and up-to-date documentation becomes essential in complex code bases with high cyclomatic complexity. Documentation helps developers understand the rationale behind complex logic and facilitates future modifications or enhancements.

Performance:

- **Execution Time:** Highly complex code may lead to slower execution times due to the increased overhead associated with evaluating multiple code paths. Optimizing complex code to improve performance often involves identifying and eliminating redundant or unnecessary branches.

- **Resource Usage:** Complex code can consume more computational resources, such as memory and CPU cycles, especially if it involves nested loops or recursive algorithms. Optimizing such code to reduce cyclomatic complexity can help conserve resources and improve overall system performance.

NPath Complexity

The NPath complexity of a method is the number of execution paths possible for a given code i.e. the number of ways the given code can get executed (ignoring cycles) e.g. if there are 2 if-else statements, the number of paths is $2 * 2 = 4$.

Software Quality

- **Bug Proneness:** Similar to cyclomatic complexity, high NPath complexity is often associated with a higher likelihood of bugs and defects. The increased number of possible code paths makes it more challenging to anticipate and test all potential scenarios thoroughly.
- **Code Readability:** High NPath complexity can result in code that is difficult to understand and reason about. Code with numerous nested branches and loops can be convoluted and obscure, making it harder for developers to maintain and debug.
- **Code Review Difficulty:** Reviewing code with high NPath complexity can be more time-consuming and error-prone. Developers may struggle to grasp the full extent of the code's behavior, leading to oversights and missed opportunities for optimization or improvement.

Maintainability

- **Code Maintenance:** Complex code with high NPath complexity is more challenging to maintain and modify. Developers may hesitate to make changes for fear of introducing unintended side effects or breaking existing functionality.
- **Refactoring Complexity:** Refactoring code to reduce NPath complexity can be difficult and risky. Breaking down complex logic into smaller, more manageable pieces requires careful planning and thorough testing to ensure that the behavior of the code remains unchanged.
- **Documentation Needs:** Clear and comprehensive documentation becomes even more critical in code bases with high NPath complexity. Documenting the rationale behind complex logic and providing examples of common usage scenarios can help future developers understand and modify the code more effectively.

Performance

- **Execution Time:** High NPath complexity can negatively impact execution time, especially in performance-critical applications. The numerous possible code paths increase the computational overhead required to evaluate each scenario, potentially leading to slower overall performance.
- **Resource Usage:** Complex code with high NPath complexity may consume more memory and CPU resources, particularly if it involves nested loops or recursive algorithms. Optimizing such code to reduce NPath complexity can help conserve resources and improve overall system efficiency.

Weighted Method Count (WMC)

The Weighted Method Count or Weighted Method per Class metric was originally defined in A Metrics Suite for Object Oriented Design by Chidamber & Kemerer.

The WMC metric is defined as the sum of complexities of all methods declared in a class. This metric is a good indicator how much effort will be necessary to maintain and develop a particular class.

Software Quality

- **Class Complexity:** High WMC values indicate that a class contains many methods, potentially making it more complex and harder to understand. This complexity can increase the likelihood of defects and bugs, as developers may struggle to comprehend and maintain intricate code structures.
- **Encapsulation:** Classes with high WMC values may violate the principle of encapsulation by incorporating too many responsibilities. Such classes become less cohesive and more tightly coupled, making it challenging to isolate and modify individual components without affecting others.
- **Code Reusability:** Complex classes with high WMC values may be less reusable due to their tightly coupled nature. Reusing or extending such classes in other parts of the system may introduce unintended dependencies and complexity, leading to code duplication or inconsistency.

Maintainability

- **Code Maintenance:** Classes with high WMC values are often harder to maintain and modify. Developers may find it daunting to navigate through numerous methods and understand their interactions, increasing the risk of introducing errors or unintended side effects when making changes.

- **Refactoring Challenges:** Refactoring complex classes with high WMC values can be challenging. Breaking down large classes into smaller, more cohesive units requires careful planning and testing to ensure that the behavior of the class remains unchanged.
- **Testability:** Highly complex classes may be more difficult to test thoroughly. The presence of numerous methods with varying degrees of complexity can make it challenging to achieve adequate test coverage, leading to gaps in the testing strategy and potential quality issues.

Performance

- **Runtime Overhead:** Classes with high WMC values may incur additional runtime overhead due to the increased number of methods and their associated complexity. Method invocations and interactions within the class may contribute to higher computational costs, impacting overall system performance.
- **Memory Usage:** Complex classes may consume more memory resources, particularly if they instantiate large data structures or maintain extensive state. Optimizing such classes to reduce WMC can help conserve memory and improve overall system efficiency.

Coupling (CBO)

Coupling is the degree of interdependence between software modules. It is a measure of how closely connected two routines or modules are; the strength of the relationships between modules.

Software Quality

- **Dependency Management:** High coupling between modules can lead to increased dependency management challenges. Changes in one module may require modifications in multiple dependent modules, increasing the risk of introducing defects and bugs.
- **Code Complexity:** Coupled modules tend to be more complex due to their inter-dependencies. This complexity can make it harder to understand and maintain the codebase, potentially leading to decreased software quality.
- **Testing Difficulty:** Testing coupled modules can be more challenging since changes in one module may have unintended consequences on other modules. This increases the complexity of test scenarios and may result in inadequate test coverage.

Maintainability

- **Code Isolation:** Coupled modules are less isolated and modular, making it harder to modify or replace individual components without affecting others. This can hinder code maintenance efforts and make it difficult to refactor or optimize the codebase.
- **Dependency Hell:** High coupling can lead to "dependency hell," where modifying one module triggers a cascade of changes across the codebase. This makes it harder to implement changes quickly and increases the likelihood of introducing errors.
- **Refactoring Challenges:** Refactoring coupled modules can be challenging since changes in one module may require corresponding changes in dependent modules. This increases the risk of introducing regressions and complicates the refactoring process.

Performance

- **Runtime Overhead:** Coupled modules may incur additional runtime overhead due to increased communication and interaction between components. This can impact overall system performance, especially in latency-sensitive applications.
- **Resource Usage:** Coupled modules may consume more memory and CPU resources, particularly if they maintain shared state or tightly integrate with each other. This can lead to inefficiencies and scalability issues, especially in resource-constrained environments.

Cohesion (LCAM)

Cohesion refers to the degree to which the elements inside a module belong together. If the methods that serve a class tend to be similar in many aspects, then the class is said to have high cohesion. In a highly cohesive system, code readability and reusability is increased, while complexity is kept manageable. The cohesion metric is calculated on a scale of 0-1, with higher values indicating lower cohesion among method.

Cohesion is usually contrasted with coupling. Low coupling often correlates with high cohesion, and vice versa. Low coupling is often thought to be a sign of a well-structured computer system and a good design, and when combined with high cohesion, supports the general goals of high readability and maintainability.

Software Quality

- **Functional Clarity:** High cohesion leads to clearer and more focused modules, where each element contributes to a specific functionality or task. This clarity enhances the readability and understandability of the codebase, reducing the likelihood of bugs and defects.
- **Reduced Complexity:** Cohesive modules are typically less complex since they have a clear and well-defined purpose. This simplifies code comprehension and maintenance, resulting in higher overall software quality.

- **Modifiability:** Cohesive modules are easier to modify and extend since changes are localized to specific functional areas. This promotes code reuse and facilitates future enhancements, improving the overall quality and flexibility of the software.

Maintainability

- **Code Isolation:** Cohesive modules are more isolated and encapsulated, making it easier to maintain and refactor individual components without affecting the rest of the system. This reduces the risk of unintended side effects and makes the codebase more resilient to changes.
- **Dependency Management:** Modules with high cohesion tend to have fewer external dependencies, reducing the complexity of dependency management. This simplifies code maintenance efforts and minimizes the risk of dependency-related issues.
- **Refactoring Flexibility:** Cohesive modules are easier to refactor since changes are confined to specific functional areas. Developers can modify or optimize cohesive modules with confidence, knowing that the changes won't have unintended consequences on other parts of the system.

Performance

- **Execution Efficiency:** Cohesive modules can improve runtime performance since they typically involve less overhead and communication compared to loosely coupled modules. This can lead to faster execution times and reduced resource consumption, especially in performance-critical applications.
- **Memory Usage:** Cohesive modules may consume less memory since they tend to have fewer dependencies and shared state. This can lead to more efficient memory utilization and better overall system performance, particularly in resource-constrained environments.

Response for Class (RFC)

The Response for Class (RFC) metric is the total number of methods that can potentially be executed in response to a message received by an object of a class. This number is the sum of the methods of the class, and all distinct methods are invoked directly within the class methods. Additionally, inherited methods are counted, but overridden methods are not, because only one method of a particular signature will always be available to an object of a given class.

Software Quality

- **Complexity:** High RFC values can indicate complex classes with numerous responsibilities. Classes with many methods may have higher cognitive complexity, making them harder to understand and maintain. This can lead to decreased software quality and increased likelihood of bugs.
- **Coupling:** Classes with high RFC values may have strong dependencies on other classes, leading to increased coupling and reduced modularity. This can make the codebase more fragile and less flexible to changes.
- **Testability:** Testing classes with high RFC values can be challenging due to the increased number of methods and potential interactions between them. It may be more difficult to achieve adequate test coverage, leading to lower software quality.

Maintainability

- **Code Complexity:** High RFC values can make classes more difficult to maintain. With numerous methods, it becomes harder to understand the class's behavior and identify areas for modification or improvement.
- **Refactoring Difficulty:** Refactoring classes with high RFC values can be challenging. Breaking down large classes into smaller, more manageable components may require significant effort and careful consideration to avoid introducing new bugs or breaking existing functionality.
- **Dependency Management:** Classes with high RFC values may have many dependencies on other classes, making it harder to isolate changes and maintain clear boundaries between components. This can increase the risk of unintended side effects and make it harder to reason about the codebase.

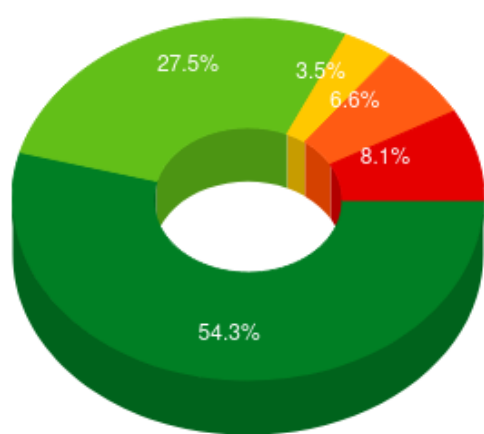
Performance

- **Runtime Overhead:** Classes with high RFC values may incur additional runtime overhead due to the increased number of methods and potential method invocations. This can impact overall system performance, especially in latency-sensitive applications.
- **Resource Usage:** Classes with high RFC values may consume more memory and CPU resources, particularly if they maintain extensive state or have many dependencies. This can lead to inefficiencies and scalability issues, especially in resource-constrained environments.

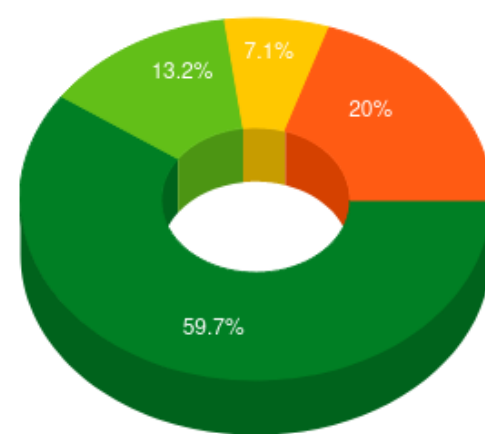
Code Metrics for some of the important classes before refactoring

Class	LOC	LCAM (0-1)	RFC	CBO	WMC	Cyclo Complexity	NPath
BookResource	366	0.662	205	29	79	79	423,038
UserResource	297	0.631	160	21	61	62	300
BookImportAsyncListener	68	0.0	26	3	2	19	5,409
TagDao	84	0.472	36	4	14	10	16
UserDao	98	0.580	74	10	18	14	23
UserBook	75	0.631	24	2	21	13	132
UserBookTag	58	0.633	21	2	26	13	132
UserAppDao	119	0.333	43	3	17	13	17
BookDataService	146	0.429	93	14	39	41	15,949

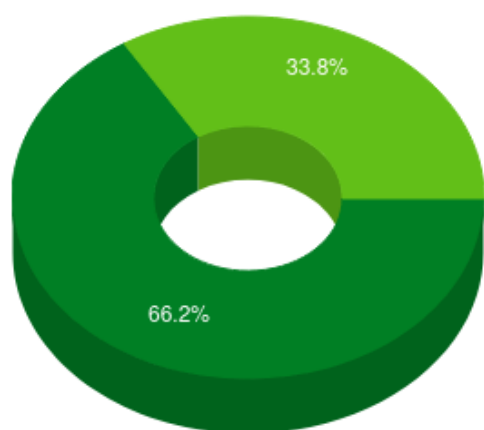
Pie charts generated by CodeMR



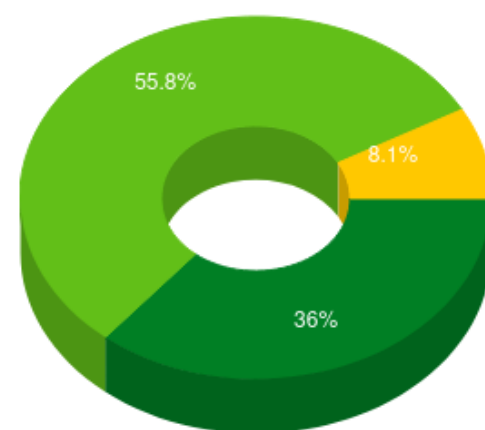
Complexity



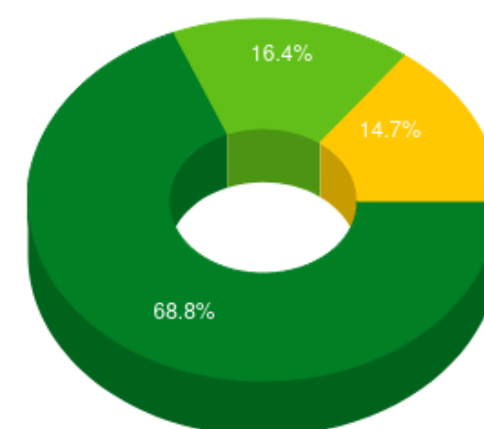
Coupling



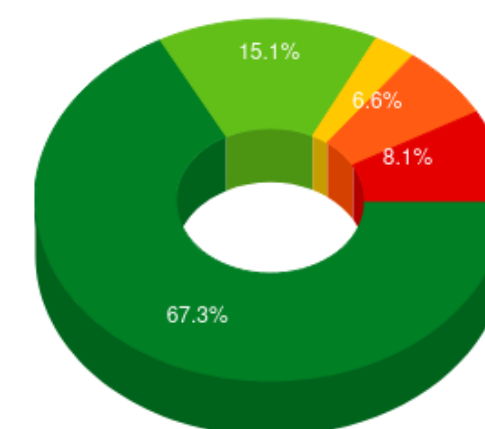
Lack of Cohesion



Size



Weighted Method Count



Response For a Class



Cyclomatic Complexity for methods having value > 7 using CheckStyle

```

  v BookDataService.java : 2 item(s)
    Cyclomatic Complexity is 11 (max allowed is 7). (167:5) [CyclomaticComplexity]
    Cyclomatic Complexity is 17 (max allowed is 7). (227:5) [CyclomaticComplexity]
  v BookImportAsyncListener.java : 1 item(s)
    Cyclomatic Complexity is 17 (max allowed is 7). (57:13) [CyclomaticComplexity]
  v BookResource.java : 2 item(s)
    Cyclomatic Complexity is 19 (max allowed is 7). (159:5) [CyclomaticComplexity]
    Cyclomatic Complexity is 24 (max allowed is 7). (273:5) [CyclomaticComplexity]
  v DbOpenHelper.java : 2 item(s)
    Cyclomatic Complexity is 12 (max allowed is 7). (60:5) [CyclomaticComplexity]
    Cyclomatic Complexity is 8 (max allowed is 7). (171:5) [CyclomaticComplexity]
  v FacebookService.java : 2 item(s)
    Cyclomatic Complexity is 10 (max allowed is 7). (136:5) [CyclomaticComplexity]
    Cyclomatic Complexity is 9 (max allowed is 7). (177:5) [CyclomaticComplexity]
  v IntentIntegrator.java : 1 item(s)
    Cyclomatic Complexity is 8 (max allowed is 7). (395:3) [CyclomaticComplexity]
  v MemoryAppender.java : 1 item(s)
    Cyclomatic Complexity is 10 (max allowed is 7). (99:5) [CyclomaticComplexity]
  v MimeTypeUtil.java : 1 item(s)
    Cyclomatic Complexity is 22 (max allowed is 7). (39:5) [CyclomaticComplexity]
  v RequestContextFilter.java : 1 item(s)
    Cyclomatic Complexity is 15 (max allowed is 7). (82:5) [CyclomaticComplexity]

```

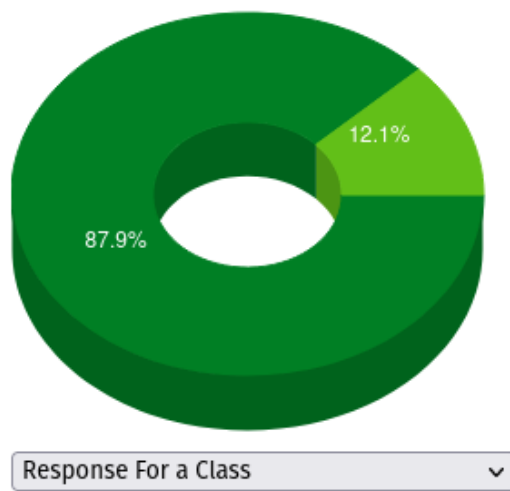
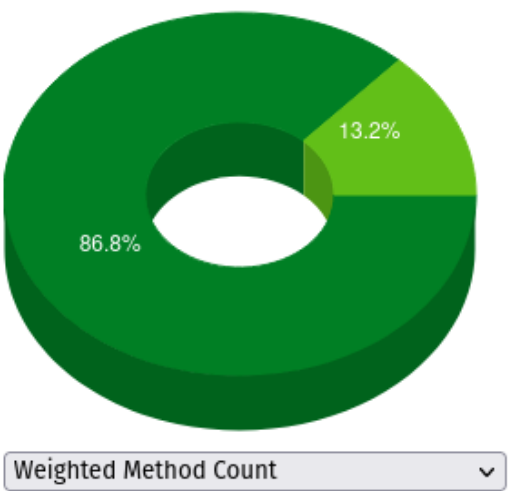
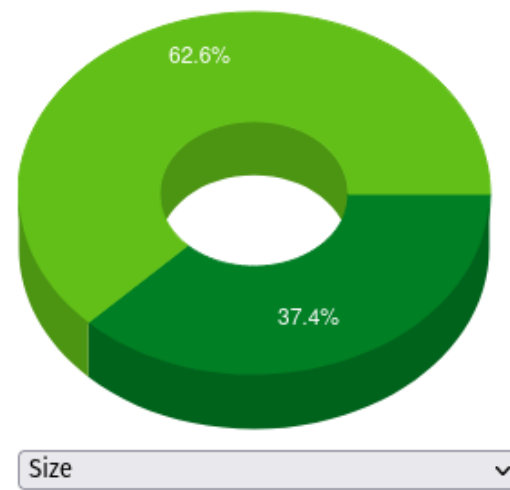
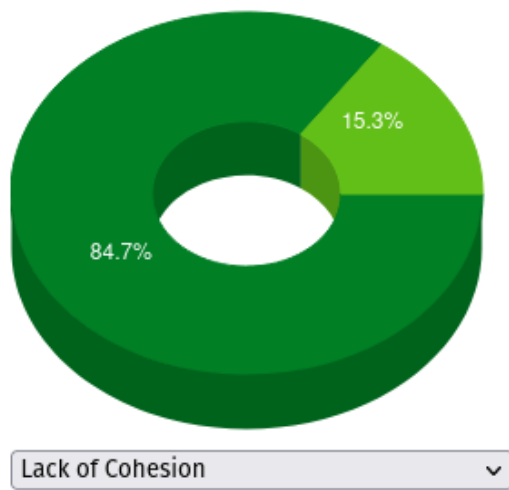
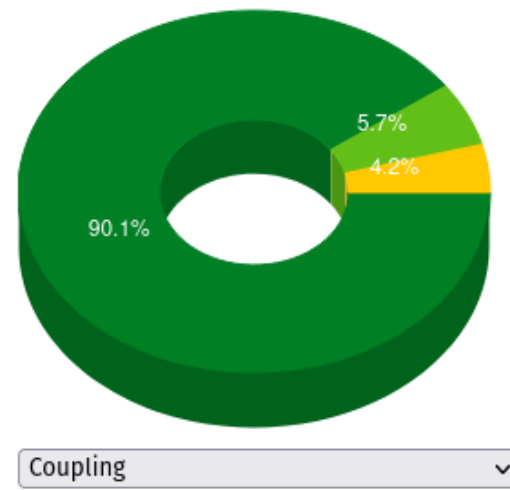
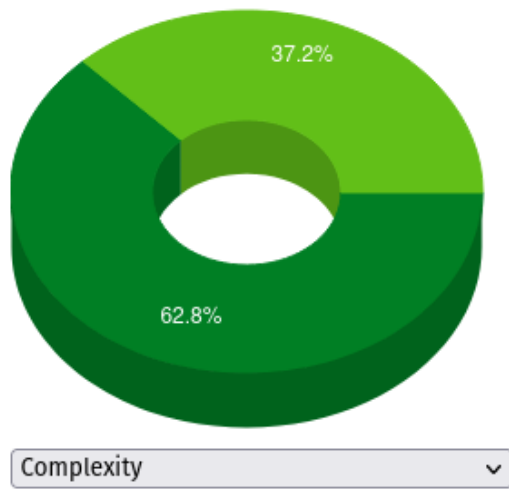
```

  v ResourceUtil.java : 1 item(s)
    Cyclomatic Complexity is 13 (max allowed is 7). (33:5) [CyclomaticComplexity]
  v TagResource.java : 1 item(s)
    Cyclomatic Complexity is 8 (max allowed is 7). (115:5) [CyclomaticComplexity]
  v TokenBasedSecurityFilter.java : 1 item(s)
    Cyclomatic Complexity is 11 (max allowed is 7). (69:5) [CyclomaticComplexity]
  v TransactionUtil.java : 1 item(s)
    Cyclomatic Complexity is 12 (max allowed is 7). (27:5) [CyclomaticComplexity]
  v UserBook.java : 1 item(s)
    Cyclomatic Complexity is 10 (max allowed is 7). (182:5) [CyclomaticComplexity]
  v UserBookDao.java : 1 item(s)
    Cyclomatic Complexity is 11 (max allowed is 7). (131:5) [CyclomaticComplexity]
  v UserBookTag.java : 1 item(s)
    Cyclomatic Complexity is 10 (max allowed is 7). (108:5) [CyclomaticComplexity]
  v UserResource.java : 2 item(s)
    Cyclomatic Complexity is 8 (max allowed is 7). (120:5) [CyclomaticComplexity]
    Cyclomatic Complexity is 8 (max allowed is 7). (307:5) [CyclomaticComplexity]
  v ValidationUtil.java : 1 item(s)
    Cyclomatic Complexity is 8 (max allowed is 7). (53:5) [CyclomaticComplexity]

```

Code Metrics After Refactoring

Class	LOC	LCAM (0-1)	RFC	CBO	WMC	Cyclo Complexity	NPath
BookResource	280	0.0	14	4	1	48	145
UserResource	285	0.0	2	1	4	60	280
BookImportAsyncListener	86	0.45	59	13	16	23	49
TagDao	53	0.333	17	3	10	6	12
UserDao	98	0.58	17	3	10	14	23
UserBook	63	0.633	18	0	20	8	10
UserBookTag	46	0.519	12	0	14	8	10
UserAppDao	100	0.333	28	2	11	7	11
BookDataService	144	0.417	59	5	40	41	15949



Cyclomatic Complexity for methods having value > 7 using CheckStyle


```

Checkstyle found 21 item(s) in 16 file(s)
  ✓ BookDataService.java : 2 item(s)
    Cyclomatic Complexity is 11 (max allowed is 7). (170:5) [CyclomaticComplexity]
    Cyclomatic Complexity is 17 (max allowed is 7). (234:5) [CyclomaticComplexity]
  ✓ BookResource.java : 1 item(s)
    Cyclomatic Complexity is 10 (max allowed is 7). (229:5) [CyclomaticComplexity]
  ✓ DbOpenHelper.java : 2 item(s)
    Cyclomatic Complexity is 12 (max allowed is 7). (60:5) [CyclomaticComplexity]
    Cyclomatic Complexity is 8 (max allowed is 7). (171:5) [CyclomaticComplexity]
  ✓ FacebookService.java : 2 item(s)
    Cyclomatic Complexity is 10 (max allowed is 7). (136:5) [CyclomaticComplexity]
    Cyclomatic Complexity is 9 (max allowed is 7). (177:5) [CyclomaticComplexity]
  ✓ HelperBookResource.java : 2 item(s)
    Cyclomatic Complexity is 10 (max allowed is 7). (66:5) [CyclomaticComplexity]
    Cyclomatic Complexity is 8 (max allowed is 7). (146:5) [CyclomaticComplexity]
  ✓ IntentIntegrator.java : 1 item(s)
    Cyclomatic Complexity is 8 (max allowed is 7). (395:3) [CyclomaticComplexity]
  ✓ MemoryAppender.java : 1 item(s)
    Cyclomatic Complexity is 10 (max allowed is 7). (99:5) [CyclomaticComplexity]
  ✓ MimeTypeUtil.java : 1 item(s)
    Cyclomatic Complexity is 22 (max allowed is 7). (39:5) [CyclomaticComplexity]
  ✓ RequestContextFilter.java : 1 item(s)
    Cyclomatic Complexity is 15 (max allowed is 7). (82:5) [CyclomaticComplexity]
  ✓ ResourceUtil.java : 1 item(s)
    Cyclomatic Complexity is 13 (max allowed is 7). (33:5) [CyclomaticComplexity]

```

```

  ✓ TagResource.java : 1 item(s)
    Cyclomatic Complexity is 8 (max allowed is 7). (121:5) [CyclomaticComplexity]
  ✓ TokenBasedSecurityFilter.java : 1 item(s)
    Cyclomatic Complexity is 11 (max allowed is 7). (69:5) [CyclomaticComplexity]
  ✓ TransactionUtil.java : 1 item(s)
    Cyclomatic Complexity is 12 (max allowed is 7). (27:5) [CyclomaticComplexity]
  ✓ UserBookDao.java : 1 item(s)
    Cyclomatic Complexity is 8 (max allowed is 7). (133:5) [CyclomaticComplexity]
  ✓ UserResource.java : 2 item(s)
    Cyclomatic Complexity is 8 (max allowed is 7). (121:5) [CyclomaticComplexity]
    Cyclomatic Complexity is 8 (max allowed is 7). (302:5) [CyclomaticComplexity]
  ✓ ValidationUtil.java : 1 item(s)
    Cyclomatic Complexity is 8 (max allowed is 7). (53:5) [CyclomaticComplexity]

```

Analysis after refactoring

1. BookResource:

- There's a significant decrease in LOC from 366 to 280, indicating a reduction in code size.
- LCAM drops from 0.662 to 0.0, suggesting improved cohesion within the class.
- RFC decreases from 205 to 14, indicating better-defined responsibilities.
- WMC and Cyclomatic Complexity decrease notably, indicating simplified logic.
- NPath decreases, implying fewer execution paths and possibly simpler methods.

2. UserResource:

- LOC decreases slightly from 297 to 285.
- LCAM drops to 0.0, indicating better cohesion.
- RFC decreases from 160 to 2, suggesting clearer responsibilities.
- WMC increases slightly, while Cyclomatic Complexity increases significantly, possibly indicating more complex logic.

3. BookImportAsyncListener:

- LOC decreases from 68 to 86.
- LCAM improves from 0.0 to 0.45, suggesting better cohesion.
- RFC increases from 26 to 59, possibly indicating additional responsibilities.
- WMC, Cyclomatic Complexity, and NPath increase, suggesting more complex methods.

4. **TagDao, UserDao, UserBook, UserBookTag, UserAppDao, BookDataService:**

- These classes generally show decreases in LOC and improvements in LCAM after refactoring.
- RFC generally decreases, indicating clearer responsibilities.
- WMC and Cyclomatic Complexity generally decrease, indicating simplified logic.
- NPath generally decreases or remains stable, suggesting simpler methods.

Overall Observations:

- Refactoring efforts have generally led to reductions in LOC and improvements in cohesion across many classes.
 - There's a trend towards clearer responsibilities, as indicated by decreases in RFC.
 - However, there are instances where certain metrics have increased, suggesting potential areas for further optimization or where complexity has been introduced unintentionally.
-