

CS6.401 Software Engineering

Project 1

Team 9

Members:

1. Anish Mathur - 2020102044
2. Pranav Kanan - 2019111033
3. Rishabh Agrawal - 2020102038
4. Sarthak Aggarwal - 2020101008
5. Souvik Karfa - 2020102051

Task 1

Book Addition & Display

1. **BookResource**: This is the primary class for managing books in the system. It interfaces directly with the frontend to facilitate book-related operations such as adding new books, updating book details, deleting books, retrieving book information, and listing books. It uses the `BookDao` for database operations, `UserBookDao` to manage user-book relationships, and interacts with `BookDataService` for external book information and thumbnails.
2. **BookDataService**: This service class is responsible for fetching book information from external sources (like Google Books and Open Library). It also manages downloading and storing book cover images. This class enhances the book addition feature by populating book details automatically based on ISBN numbers.
3. **BookDao** (implied usage): While not explicitly shown in the UML, it's assumed to be used by `BookResource` for direct database operations like creating, updating, and fetching book records.

4. **UserBookDao**: Manages the association between users and books, such as tracking which books are added by which users. It's crucial for personalized book displays and user-specific book management.

Bookshelf Management

1. **TagResource**: This class manages the bookshelf; tag is its alias name. As bookshelf is just a category of book it can be called a tag as well. This class is responsible of providing methods for creating, editing and deleting tags in the system.

User Management

1. **UserResource**: This class manages all user-related functionalities, including user registration, authentication (login and logout), updating user profiles, deleting users, and listing users. It serves as the bridge between the frontend and the backend for user management.
2. **UserDao**: Directly interacts with the database for user management operations such as creating new users, updating user information, authentication, and user deletion.
3. **AuthenticationTokenDao**: Manages session tokens for users, which are essential for maintaining user sessions and handling login states. It creates, deletes, and validates tokens for user authentication.
4. **User**: Represents the user entity in the system, containing attributes like username, password, email, and others necessary for user management.
5. **AuthenticationToken**: Represents the session token entity, including attributes for token management like token ID, user ID, and token expiry.
6. **AppResource**: This class handles all the server logs for the admin.

Detailed Functionality and Behavior

- **BookResource**:
 - Interfaces with `BookDao` and `UserBookDao` to perform CRUD operations on books.
 - Uses `BookDataService` to fetch external book data and images.

- Provides endpoints for book addition, deletion, update, and listing.
- **BookDataService:**
 - Fetches book data from external sources.
 - Downloads and stores book cover images.
 - Is used by `BookResource` to enrich book entries with external data.
- **UserResource:**
 - Manages user registration, login, logout, and profile updates.
 - Interacts with `UserDao` for direct user management tasks.
 - Uses `AuthenticationTokenDao` for handling user sessions and authentication tokens.
- **TagResource**
 - Manages addition and deletion of bookshelves in the system.
 - This `UserResource` interacts with these functions to add and delete the bookshelves.
 - This list of tags can be accessed by `BookResource` to be assigned to them.
- **UserDao and AuthenticationTokenDao:**
 - Perform database operations related to users and session tokens, respectively.
 - Essential for authenticating users and maintaining session states.

Task 2a

Design and Code Smells:

1. **God Class** `UserResource.java` Classes that have too many responsibilities and are overly complex. These classes are doing more than they should, which makes them difficult to maintain and understand.
2. **God Method** `add()` and `update()` in `BookResource.java` : These methods are overly complicated and perform too many functions inside them. Breaking the method into small functions is better to improve readability and maintainability.

3. **Too Few Branches for a Switch Statement** (`BookListActivity.java` and `BookListFragment.java`): Using a switch statement for fewer than three branches is less efficient than using an if statement.
4. **Avoid Reassigning Parameters** (`BooksAdapter.java` , `UserResource.java`): Reassigning parameters inside methods can lead to confusing behaviour and bugs. It's better to use local variables.
5. **Simplify Conditional** (`BaseResource.java`): Unnecessary complexity in conditionals should be avoided; for example, there's no need to check for null before an `instanceof` check.
6. **Close Resource** (`BookDataService.java`): Not closing resources after use can lead to memory leaks and other resource exhaustion issues.
7. **Magic Literals**: In the codebase, there are several instances of **magic literals**—hardcoded numeric and string values. These magic literals make the code harder to maintain, and prone to inconsistencies.

Task 2b

Metrics using CodeMR

List of all classes (#10)

ID	CLASS	COUPLING	COMPLEXITY	LACK OF COHESION	SIZE	LOC	COMPLEXITY	COUPLING	LACK OF COHESION	SIZE
1	BookResource					366	medium-high	high	low-medium	medium-high
2	UserResource					297	medium-high	high	low-medium	low-medium
3	ConnectResource					131	low-medium	high	low	low-medium
4	AppResource					66	low-medium	medium-high	low	low-medium
5	TagResource					85	low-medium	low-medium	low	low-medium
6	ThemeResource					15	low-medium	low	low	low
7	LocaleResource					15	low-medium	low	low	low
8	BaseFunction					2	low-medium	low	low	low
9	TextPlainMessageB...					24	low	low	low	low
10	BaseResource					21	low	low	low	low

The screenshot from the CodeMR tool provides an analysis of various classes within a codebase, assessing them based on key software metrics such as coupling, complexity, lack of cohesion, and size. Let's discuss each of these metrics, their implications, and how they might guide refactoring decisions.

1. *Coupling*: Measures the degree to which a class is dependent on other classes. High coupling indicates that a class may be difficult to modify in isolation and may be prone to bugs when changes are made. Classes like `BookResource` and `UserResource` with high coupling may need refactoring to reduce dependencies, possibly through interface abstraction or the use of design patterns such as Dependency Injection.
2. *Complexity*: Relates to how complex the class logic is. A high complexity score, as seen in `BookResource` and `UserResource`, suggests these classes have many branches or loops, making them harder to understand and maintain. Refactoring to simplify methods, such as breaking them into smaller, single-purpose methods, could be beneficial.
3. *Lack of Cohesion*: Indicates how closely related the methods and fields of a class are. Low cohesion can make a class hard to comprehend and may lead to code duplication. Classes with low-medium cohesion might need to be broken down into smaller, more focused classes that each have a single responsibility.
4. *Size*: Refers to the physical size of the class in terms of lines of code (LOC) or the number of methods and attributes. Large classes, such as `BookResource`, are typically harder to maintain and understand. They might be candidates for splitting into smaller classes or for separating concerns into different layers of the application (e.g., service layer, data access layer).
5. *LOC (Lines of Code)*: Directly indicates the length of the class. Longer classes, especially those with high complexity and coupling like `BookResource`, may be doing too much and could benefit from being broken up.
6. *Complexity (Duplicate Metric)*: This appears to be listed twice, which could be an error in the tool's reporting or represent different facets of complexity, such as cyclomatic complexity versus cognitive complexity.

Both measures of complexity inform about the risk of bugs and maintenance challenges.

Implications and Guiding Refactoring:

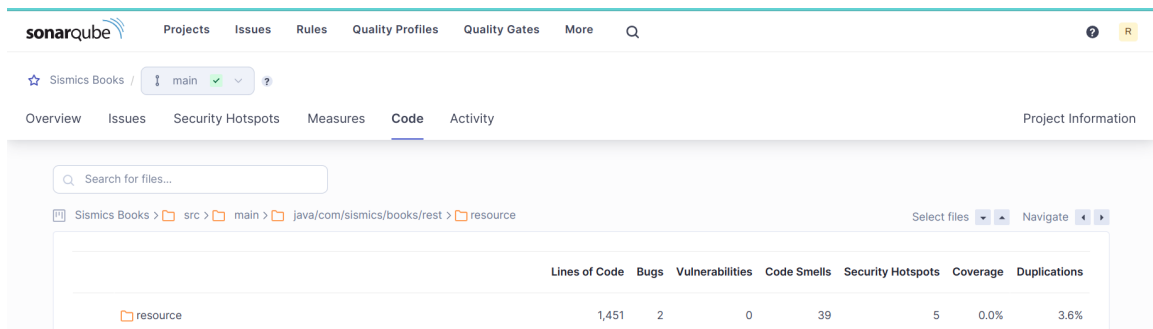
- *BookResource*: Given its high complexity of its methods and coupling, consider decomposing its responsibilities. For instance, give the responsibility of setting and updating book properties to the `Book` class.
- *UserResource*: Similar to *BookResource*, look for opportunities to simplify by dividing responsibilities and reducing dependencies on other classes. For instance, we can separate the responsibilities of admin and user, making a separate class for admin.
- **ConnectResource**: Despite its lower LOC, it has high coupling, suggesting it may be too intertwined with other parts of the system. Assess whether its role can be clarified or if its tasks can be distributed.
- **AppResource** and **TagResource**: They have medium-high complexity, indicating potential for simplifying their internal logic.
- **ThemeResource** and **LocaleResource**: These classes appear well-managed in terms of size and complexity. However, always be vigilant for creeping features that can increase complexity over time.
- **BaseFunction**, **TextPlainMessageBodyWriter**, and **BaseResource**: They have low complexity and coupling, suggesting they're well-designed, but continue to monitor as new features are added to ensure they don't become catch-all classes.

Refactoring Decisions:

The metrics suggest focusing refactoring efforts on *BookResource* and *UserResource* due to their higher complexity and coupling. By applying principles from SOLID design, such as the Single Responsibility Principle and Interface Segregation Principle, these classes can be made more manageable and maintainable. Additionally, refactoring should aim to increase cohesion within classes and reduce the overall size of classes where possible.

In summary, these metrics serve as indicators for potential code smells and provide a guide for where refactoring efforts will be most effective. They highlight areas of risk in the codebase and can help prioritize refactoring tasks to improve the quality and maintainability of the system.

Metrics of SonarQube analysis



SonarQube helped us in localizing the code snippets, which resulted in the above high complexity. It also notified about potential bugs that could lead to potential leakage of data.

Task 3b

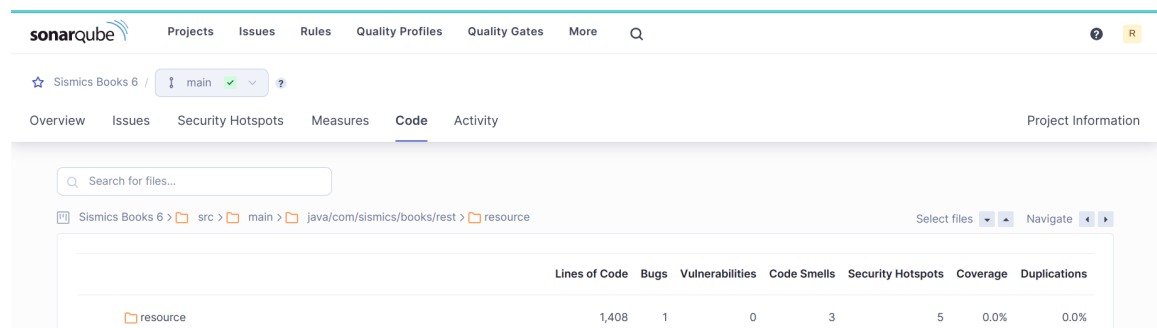
Metrics using CodeMR

List of all classes (#12)										
ID	CLASS	COUPLING	COMPLEXITY	LACK OF COHESION	SIZE	LOC	COMPLEXITY	COUPLING	LACK OF COHESION	SIZE
1	BookResource					332	medium-high	high	low-medium	medium-high
2	ConnectResource					131	low-medium	high	low	low-medium
3	UserResource					168	low-medium	medium-high	low	low-medium
4	AdminResource					131	low-medium	medium-high	low	low-medium
5	AppResource					66	low-medium	medium-high	low	low-medium
6	TagResource					85	low-medium	low-medium	low	low-medium
7	ThemeResource					15	low-medium	low	low	low
8	LocaleResource					15	low-medium	low	low	low
9	BaseFunction					2	low-medium	low	low	low
10	ConstantResource					31	low	low	low	low
11	TextPlainMessageB...					24	low	low	low	low
12	BaseResource					21	low	low	low	low

The code underwent a significant transformation! Both complexity and coupling have been dramatically reduced, moving from high levels to either low or medium. This indicates that the code is now easier to understand,

maintain, and modify. Additionally, cohesion has increased, signifying that the code is now better organized and focused, with improved relationships between its components. These positive changes contribute to a more robust and sustainable codebase, making it easier to develop and evolve in the future.

Metrics of SonarQube analysis



	Lines of Code	Bugs	Vulnerabilities	Code Smells	Security Hotspots	Coverage	Duplications
resource	1,408	1	0	3	5	0.0%	0.0%

This analysis on **SonarQube** shows that all the smells identified by the tool are now resolved.

Initially, in the following section of the code base, there were 40 smells which is now reduced to just 3 smells.

Task 3c

ChatGPT suggested the following design smells and refactored code:

1. **se DTOs (Data Transfer Objects):** Instead of passing numerous parameters, consider encapsulating related parameters into a DTO (Data Transfer Object) to make the method signature cleaner and more maintainable.
2. **Consolidate Validation Logic:** Consolidate validation logic into a separate method to improve readability and maintainability.
3. **Separate Business Logic:** Separate business logic from the controller method to adhere to the Single Responsibility Principle.
4. **Use Dependency Injection:** Instead of creating DAO instances within the method, consider injecting them through constructor injection or using a dependency injection framework.
5. **Error Handling Improvement:** Handle exceptions more gracefully and provide meaningful error messages.

6. **Use Optional Instead of Null Checks:** Use `Optional` or other mechanisms to avoid explicit null checks.

```
public class BookDTO {
    private String title;
    private String subtitle;
    private String author;
    private String description;
    private String isbn10;
    private String isbn13;
    private Long pageCount;
    private String language;
    private String publishDateStr;
    private List<String> tagList;
    // getters and setters
}
```

```
public class BookService {
    private final BookDao bookDao;
    private final UserBookDao userBookDao;
    private final TagDao tagDao;
```

```
    public BookService(BookDao bookDao, UserBookDao userBook
Dao, TagDao tagDao) {
        this.bookDao = bookDao;
        this.userBookDao = userBookDao;
        this.tagDao = tagDao;
    }
```

```
    public Response add(BookDTO bookDTO, Principal principa
l) throws JSONException {
        if (!authenticate()) {
            throw new ForbiddenClientException();
        }

        validateInput(bookDTO);
```

```

        String bookId = createBook(bookDTO);
        updateUserBook(principal.getId(), bookId);
        updateTags(bookDTO.getTagList(), principal.getId());

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

    private void validateInput(BookDTO bookDTO) {
        // Validation logic
    }

    private String createBook(BookDTO bookDTO) {
        // Creation logic
    }

    private void updateUserBook(String userId, String bookId) {
        // Update user book logic
    }

    private void updateTags(List<String> tagList, String userId) {
        // Update tags logic
    }
}

```

Gemini could not take in so much data.

books-

web/src/main/java/com/sismics/books/rest/resource/BookResource.java

Update function:

The above functions had the following design smells:

1. **Code Duplication:** There is repetition of code for input validation, book retrieval, and tag handling in both the `add` and `update` methods. This

violates the DRY (Don't Repeat Yourself) principle and can lead to maintenance issues if changes are needed in the future.

2. **Large Method:** Both the `add` and `update` methods are quite lengthy, containing multiple responsibilities such as input validation, database operations, and response creation. Large methods make code harder to understand, maintain, and test.
3. **Low Cohesion:** The methods are responsible for a mix of tasks, including input validation, database operations, and response generation. This violates the principle of high cohesion, which suggests that a method should have a single, well-defined purpose.
4. **Poor Error Handling:** Error handling is scattered throughout the methods and could be improved for better readability and maintainability. For instance, error messages are hardcoded in some cases, which makes it harder to maintain consistency and localization.
5. **Direct Interaction with DAOs:** The methods directly instantiate DAOs, violating the principle of Dependency Inversion. This makes the code tightly coupled to specific implementations and difficult to unit test.
6. **Violation of Single Responsibility Principle (SRP):** Both methods are doing more than one thing. They handle both business logic (such as validating input, updating database records) and request handling (such as parsing form parameters, generating responses). This makes the methods harder to understand and maintain.
7. **Lack of Abstraction:** There is a lack of abstraction in handling database operations and input validation. Extracting common functionalities into separate methods or classes could improve code readability and maintainability.

The following code is suggested to mitigate these issues:

```
public class BookService {
    private final UserBookDao userBookDao;
    private final BookDao bookDao;
    private final TagDao tagDao;

    public BookService(UserBookDao userBookDao, BookDao book
```

```

    Dao, TagDao tagDao) {
        this.userBookDao = userBookDao;
        this.bookDao = bookDao;
        this.tagDao = tagDao;
    }

    public Response update(String userBookId, BookDTO bookDT
0, Principal principal) throws JSONException {
        if (!authenticate()) {
            throw new ForbiddenClientException();
        }

        validateInput(bookDTO);

        UserBook userBook = getUserBook(userBookId, principa
l.getId());
        Book book = getBook(userBook.getBookId());

        validateISBN(bookDTO, book);

        updateBook(book, bookDTO);

        updateTags(bookDTO.getTagList(), userBookId, princip
al.getId());

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

    private void validateInput(BookDTO bookDTO) {
        // Validation logic
    }

    private UserBook getUserBook(String userBookId, String u
serId) {
        UserBook userBook = userBookDao.getUserBook(userBook
Id, userId);
    }

```

```

        if (userBook == null) {
            throw new ClientException("BookNotFound", "Book
not found with id " + userBookId);
        }
        return userBook;
    }

    private Book getBook(String bookId) {
        return bookDao.getById(bookId);
    }

    private void validateISBN(BookDTO bookDTO, Book book) {
        // Validation logic for ISBN
    }

    private void updateBook(Book book, BookDTO bookDTO) {
        // Update book logic
    }

    private void updateTags(List<String> tagList, String use
rBookId, String userId) {
        // Update tags logic
    }
}

```