SE Project-1 Report

Table of Contents

Team Members

1) Vineeth Bhat (2021101103)

2) Ishwar B Balappanawar (2021101023)

3) Swayam Agrawal (2021101068)

4) Mitansh Kayathwal (2021101026)

5) G.L.Vaishnavi (2023204009)

Task 1 : Mining the Repository

Introduction

Methodology

> *"In the labyrinth of code, do embark on an exploration guided by strategy, uncovering hidden chambers and winding passages. Each line reveals a twist in the intricate dance of JavaScript and Java, leading you closer to understanding. Amidst the complexity, the silent guardians—the classes—hold the key to enlightenment, guiding your journey through this digital maze."*
>
> **Sun Tzu**

The code is notably intricate, with layers of complexity. Given this intricate nature, it became imperative for us to devise a strategy. Our approach commenced with a deep dive into the repository, focusing on the functionality of the actual project. We executed the code, meticulously testing various components of the web application. This involved scrutinizing the corresponding JavaScript responsible for handling these components and the requests it transmitted to the Java Server via its API.

Subsequently, it became relatively straightforward to discern the classes implicated, enabling us to swiftly pinpoint the principal actors within these subsystems.

Additionally, we leveraged ChatGPT to efficiently generate PlantUML versions of the classes embedded in the code. However, we remained highly attentive to the interconnections between these classes, ultimately identifying each relationship through manual inspection.

Assumptions

- Every Class that is not an inbuilt Java Class and has been manually defined in the code is considered in the System and in the subsequent systems. - Including the seemingly low level ones.
  It is because we know how these classes interact with our main classes and this should be accurately reflected in the UML diagram. Plus some of these low level classes are fairly complex(hence non trivial)
- Only classes that directly affect the functionality of a class are included in that particular subsystem.
  This is because we have included only those classes that directly affect the functionality and have not shown other dependencies to keep the Subsystem clean. Although it is accurately reflected in the God UML for the entire system.

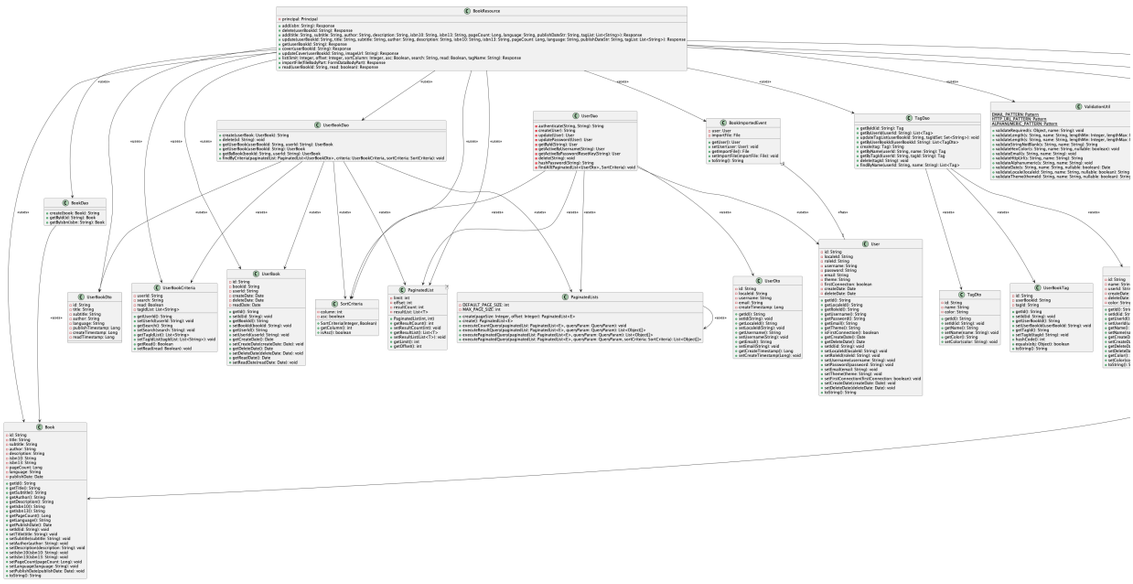
For example - Dependencies on UserBookDao on classes like UserBookDto, PaginatedList, PaginatedLists, SortCriteria etc have not been included in the UML diagram of the BookSelf Subsystem.

- We will be submitting different versions of UML Diagrams -
  - One, that will be an accurate reflection of the entire code-base which accurately models the class structures and their interactions
  - Two, an UML meant for understanding the subsystems - this models the major players in those particular subsystems who directly affect the functionality of that subsystem

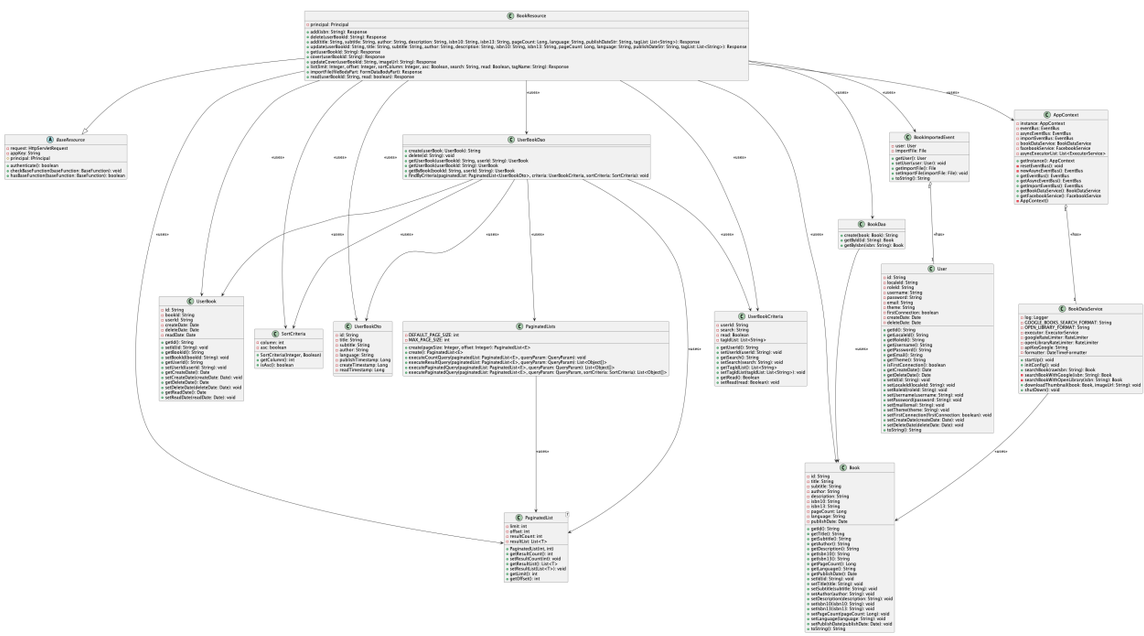
  The rationale behind different versions of UML is that the granularity of an UML is to be decided by the purpose for which it will be used,

Book Addition & Display Subsystem

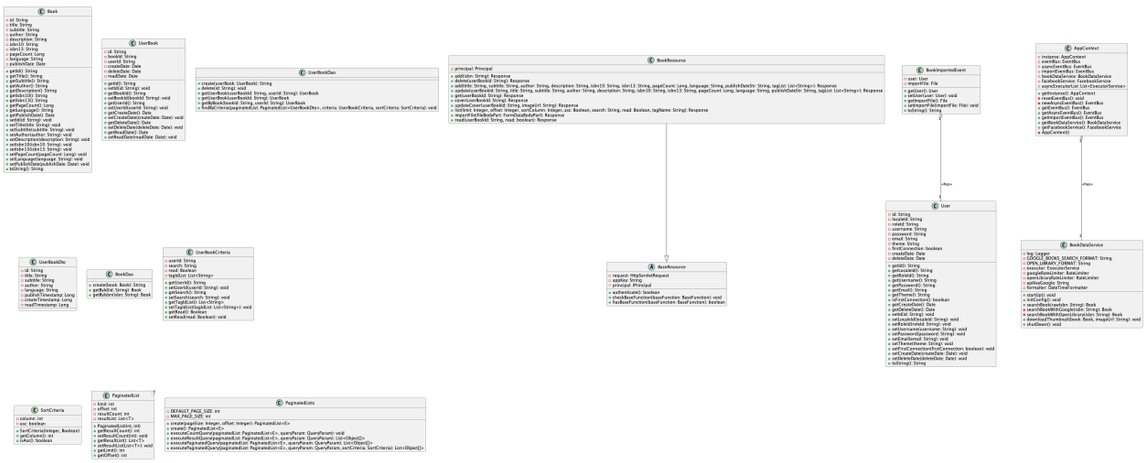
UML V1.1 (With all relations, a superset of classes of Subsystem)



UML V1.2 (With all relations, only major classes of Subsystem)



UML V1.3 (Without Dependencies, only major classes of Subsystem)



Classes Used

1. `class BookResource` - This class represents the RESTful resource for managing books. It includes methods for adding, updating, deleting, and retrieving books, as well as importing books from a file.
2. `class Book` - It represents books in the system, storing attributes such as title, subtitle, author, description, ISBN, page count, language, and publication date.
3. `class Tag` - It represents tags used in the system, storing attributes such as tag name, color, creation date, and user ID
4. `class User` - It contains user attributes such as username, password, email, and theme preferences.
5. `class UserBook` - This class facilitates the representation and manipulation of user-book associations
6. `class UserBookTag` - It represents relationship between a user book and a tag in the database. It contains fields for the IDs of the associated user book and tag, along with overridden methods for hash code generation, equality comparison, and string representation
7. `class BookDao` - It includes methods to create a new book, retrieve a book by its ID, and fetch a book by its ISBN number (10 or 13 digits)
8. `class UserDao` - It manages user-related operations, including authentication, creation, update, and retrieval by various criteria such as ID, username, or password recovery token. It also handles password hashing for security and supports pagination for listing users.
9. `class UserDto` - It's used for transferring user data such as ID, locale ID, username, email, and creation timestamp between different layers of the application.
10. `class UserBookDao` -It provides methods for managing user books in the database. It includes functionalities for creating, deleting, and searching user books based on various criteria.
11. `class UserBookDto` - It contains attributes such as the user book ID, title, subtitle, author, language, publication date, creation date, and read date. These attributes facilitate transferring user book data between different layers of the application.
12. `class TagDao` - It has methods to interact with tag-related data in the database. It includes functionalities such as retrieving tags by ID or name, updating tags on a user book, creating new tags, and deleting tags.
13. `class TagDto` - It is used for transferring tag-related data between different layers of the application with fields such as ID, name, and color

14. `class UserBookCriteria` - It defines criteria for searching user books based on specific conditions . It includes fields such as user ID, search query, read state, and a list of tag IDs.
15. `class TagResource` - It provides RESTful endpoints for managing tags, including listing , creating, updating , and deleting tags. It ensures authentication, handles input validation, and interacts with a `TagDao` to perform CRUD operations on tags, returning JSON responses for each operation.
16. `class BaseFunction` - It is a Java entity class representing core functions, identified by their unique `id`
17. `class BaseResource` - It serves as the foundation for API access in an Android application.
18. `class DirectoryUtil` - It facilitates access to various storage directories used by the application
19. `class BookImportedEvent` - It represents an event raised when a user requests to import books:
20. `class PaginatedList` - It manages paginated data with details such as page size, offset, total result count, and the list of records for the current page.
21. `class PaginatedLists` - It provides utilities for managing paginated lists, including methods for creating paginated lists with customizable parameters, executing count queries to determine the total number of results, and executing paginated queries to retrieve data for the current page.
22. `class SortCriteria` - It represents the sorting criteria of a query, including the index of the column to sort and whether the sorting order is increasing or decreasing.
23. `class BookDataService` - It provides functionality for fetching book information from various APIs like Google Books and Open Library.
24. `class AppContext` - It manages application-wide services and event buses, offering singleton access to `BookDataService` and `FacebookService`. It orchestrates synchronous and asynchronous event handling, including specialized buses for mass imports.
25. `class ValidationUtil` - It provides methods for validating parameters, including checks for null, string length, email format, URL format, alphanumeric characters, and date parsing. It also includes validation for locale and theme IDs, throwing client exception in case of validation failure.

Functionality and Behaviour

The class BaseResource, which serves as a foundational structure for handling authentication and base functionalities. This system operates with various entities, including Book, Tag, and UserBook, each encapsulating relevant attributes and methods for managing their data. User and UserDto classes represent user entities, with UserDao facilitating database interactions for user-related operations such as authentication, creation, and retrieval. Similarly, BookDao and TagDao fulfill roles for managing book and tag entities.

BookResource serves as a key interface for handling book-related operations, with methods for listing books, retrieving book details, and adding new entries. It acts as a bridge between user requests and the underlying data storage. The BookDataService class complements this functionality by providing services for searching books, downloading associated thumbnails, and managing the lifecycle of these operations.

Central to the system is the AppContext, representing the application's global context. It manages essential resources such as event buses, data services, and asynchronous executors, providing a centralized access point for these components throughout the application. Additionally, the ValidationUtil class offers utility methods for data validation, ensuring the integrity and consistency of user-provided information.

OOP Concepts

1. **Abstraction**: The abstract class BaseResource provides a template for other classes to implement common functionality related to handling HTTP requests.

2. **Inheritance**: Classes like BookResource and TagResource inherit from the abstract class BaseResource, inheriting its methods and properties.
3. **Encapsulation**: Classes encapsulate data and behavior, such as UserDao encapsulating methods related to user data access.
4. **Polymorphism**: Polymorphism is demonstrated with methods like authenticate() in BaseResource, which can behave differently depending on the class implementing it.
5. **Association**: Association represents the relationship between classes. For example, BookDao is associated with the Book class to perform operations on book data.
6. **Aggregation**: Aggregation is seen where UserDao aggregates multiple instances of the User class.
7. **Dependency**: Dependency signifies that a class relies on another class. For instance, BookResource has dependencies on classes like BookDao and UserBookDao to perform its operations.

Strengths

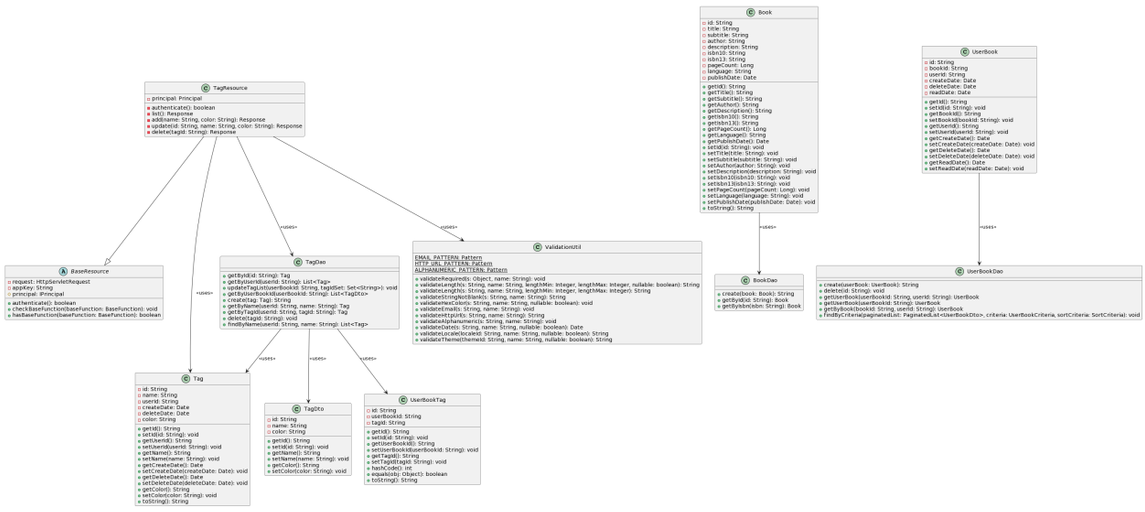
1. It has robust functionality for managing users, covering authentication, creation, updates, and password management.
2. Book-related operations, such as listing, retrieval by ID or ISBN, and adding new books, facilitate effective interaction with a library or catalog of books.
3. The Tagging System allows users to categorize and organize books based on preferences or characteristics, enhancing organization and retrieval.
4. It efficiently handles data access through data access objects (DAOs) and supports pagination for optimal performance, especially with large amounts of data.
5. Event-driven architecture enables the system to respond to events like book imports or system initialization, enhancing flexibility and adaptability.
6. Validation utilities ensure data integrity and validity, mitigating errors and maintaining consistency for reliable system operation.

Weaknesses

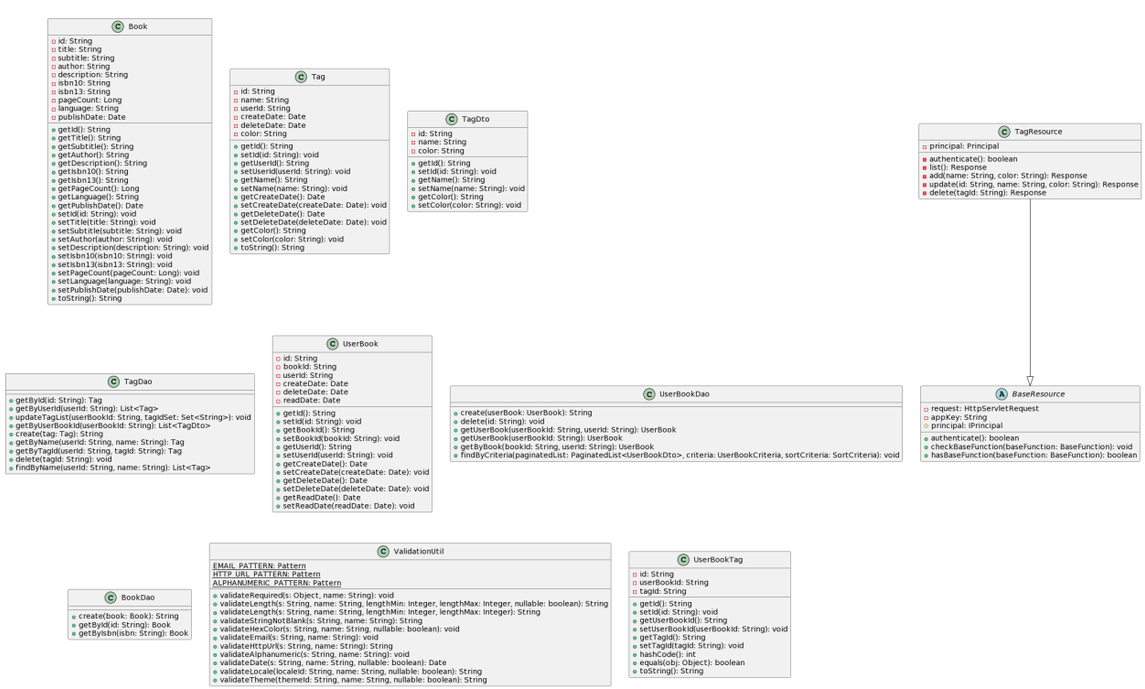
1. It is unclear how data validation and error handling are integrated into the system's functionality, without which the system may struggle to enforce data integrity and respond appropriately to erroneous user input.
2. Lack of features such as account customization, or social authentication restricts the system's ability to meet diverse user needs and preferences.

Bookshelf Management Subsystem

UML V2.1 (All Relations)



UML V2.2 (All relations except Dependencies)



Functionality and Behaviour:

The BookShelfManagement system provides users with a comprehensive platform for managing their digital book collections. Users can perform various book-related operations, such as adding new books to their shelves, retrieving book details based on ID or ISBN, and removing books from their collections. This ensures that users can effortlessly organize and maintain their virtual bookshelf according to their preferences.

Additionally, the tagging system allows users to categorize and organize books using customizable tags. Users can apply multiple tags to books, enabling them to create personalized collections or categorize books based on genre, author, or reading status. This tagging feature enhances the organization and retrieval of books, providing users with a flexible way to navigate their bookshelf.

Furthermore, the system tracks user interactions with books, capturing details such as when a book was added, read, or removed from the collection. This tracking functionality provides users with basic book information to the user allowing them to manage their reading activities effectively

Classes Used

1. `class Book` - It represents books in the system, storing attributes such as title, subtitle, author, description, ISBN, page count, language, and publication date.
2. `class Tag` - It represents tags used in the system, storing attributes such as tag name, color, creation date, and user ID
3. `class UserBook` - This class facilitates the representation and manipulation of user-book associations
4. `class BookDao` - It includes methods to create a new book, retrieve a book by its ID, and fetch a book by its ISBN (10 or 13 digits)
5. `class TagDao` - It has methods to interact with tag-related data in the database. It includes functionalities such as retrieving tags by ID or name, updating tags on a user book, creating new tags, and deleting tags.
6. `class UserBookDao` - It provides methods for managing user books in the database. It includes functionalities for creating, deleting, and searching user books based on various criteria.
7. `class TagDto` - It is used for transferring tag-related data between different layers of the application with fields such as ID, name, and color
8. `class TagResource` - It provides RESTful endpoints for managing tags, including listing, creating, updating, and deleting tags. It ensures authentication, handles input validation, and interacts with a `TagDao` to perform CRUD operations on tags, returning JSON responses for each operation.
9. `class BaseResource` - It serves as the foundation for API access in an Android application.
10. `class UserBookTag` - It represents the relationship between a userbook and a tag in the database. It contains fields for the IDs of the associated user book and tag, along with overridden methods for hash code generation, equality comparison, and string representation
11. `class ValidationUtil` - It provides methods for validating parameters, including checks for null, string length, email format, URL format, alphanumeric characters, and date parsing. It also includes validation for locale and theme IDs, throwing client exceptions in case of validation failure.

OOPs Concepts

1. **Inheritance**: Demonstrated by the inheritance relationship between TagResource and BaseResource. This allows TagResource to inherit attributes and methods from BaseResource.
2. **Encapsulation**: Exhibited by classes such as UserBook, Tag, and Book, which encapsulate their attributes and methods, providing access only through well-defined interfaces.
3. **Composition**: Represented by the composition relationship between BookDao and Book, indicating that a BookDao object contains Book objects, enabling BookDao to perform operations on Book instances.
4. **Abstraction**: Seen in the BaseResource class, which abstracts common functionalities for handling HTTP requests, allowing concrete resource classes to implement specific functionalities.
5. **Association**: Various classes in the diagram, such as TagDao, UserBookDao, and BookDao, exhibit associations with other classes, signifying relationships between objects.
6. **Dependency Injection**: Although not explicitly depicted, the concept of dependency injection might be utilized in the system, allowing dependencies to be injected into classes like TagResource, enhancing testability and flexibility.

Strengths:

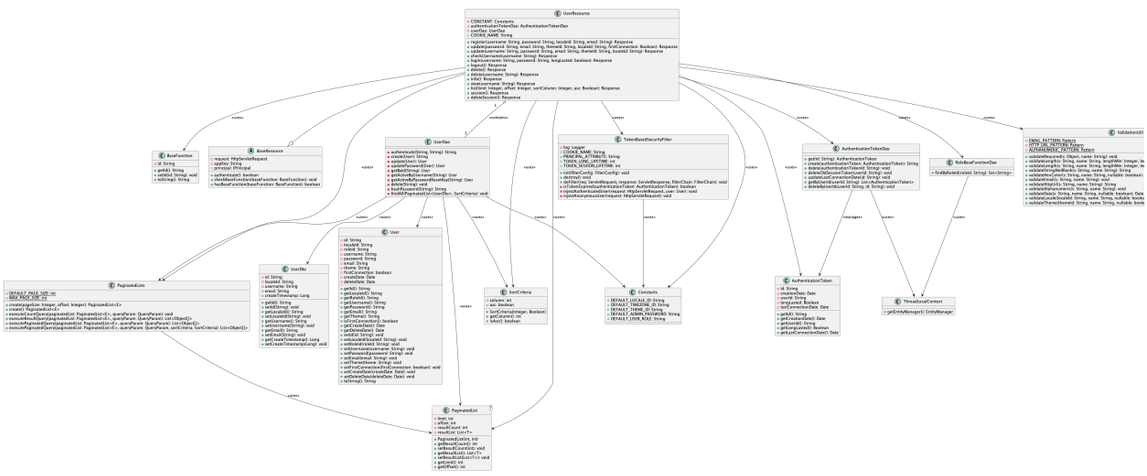
1. It offers robust functionality for managing books such as viewing books, adding new books, updating book information and deleting books.
2. Users can create and manage tags dynamically, allowing categorization based on individual preferences.

Weaknesses:

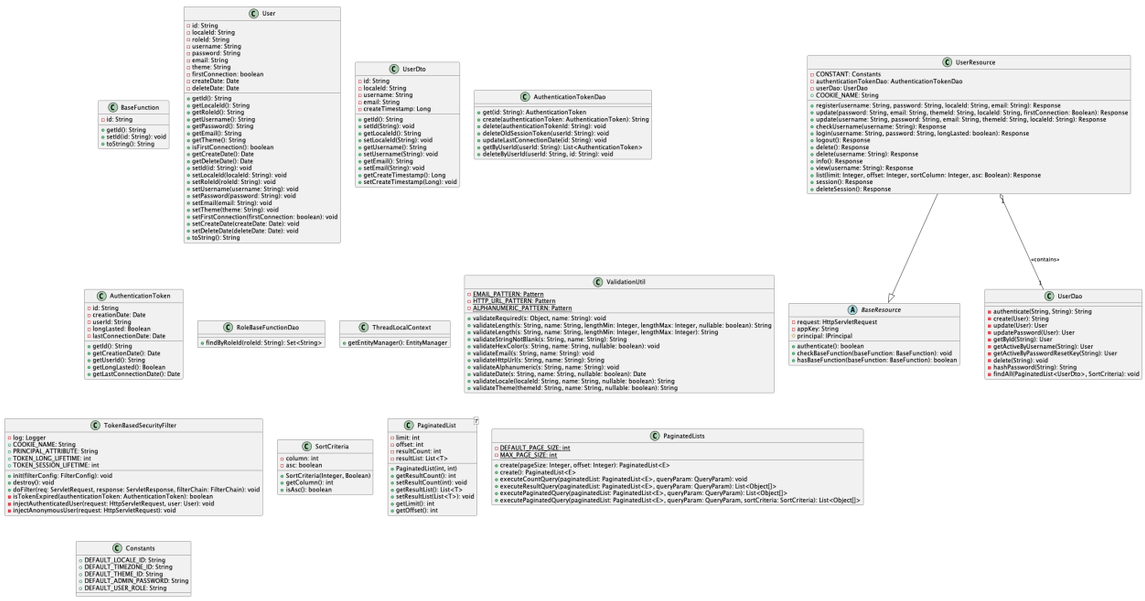
1. As the number of tagged books grows, scalability challenges may arise, particularly in terms of performance and usability, necessitating optimization strategies to maintain system efficiency.
2. Users may create an excessive number of tags, leading to clutter and confusion in the tagging system, especially if there are no mechanisms in place to manage or consolidate redundant tags.
3. It does not include collaborative features for sharing or collaborating on bookshelves with other users, such as sharing reading lists, recommendations, or annotations with friends or colleagues.

User Management Subsystem

UML V3.1 (All Relations)



UML V3.2 (All relations except Dependencies)



Functionality and Behaviour

It encompasses essential operations such as User registration, login, profile management, and session handling. Users can seamlessly register for accounts, log in securely using their credentials, and update their profiles with personal information such as email addresses and themes. The system also supports session management, ensuring that users remain authenticated during their active sessions and can log out when desired. Furthermore, the code incorporates mechanisms for error handling, effectively managing exceptions that may arise during user interactions, such as invalid login attempts or database errors.

Classes Used

1. `class UserResource` - It handles user-related REST endpoints, such as registration, updating, authentication, session management, and deletion. It utilizes input validation from `ValidationUtil`, token-based security filters, and provides endpoints for user information retrieval and session management.
2. `class BaseResource` - It serves as the foundation for API access in an Android application.
3. `class User` - It contains user attributes such as username, password, email, and theme preferences.
4. `class UserDao` - It manages user-related operations, including authentication, creation, update, and retrieval by various criteria such as ID, username, or password recovery token. It also handles password hashing for security and supports pagination for listing users.
5. `class UserDto` - It's used for transferring user data such as ID, locale ID, username, email, and creation timestamp between different layers of the application.
6. `class AuthenticationTokenDao` - It manages authentication tokens in the database, allowing operations such as creation, deletion, and retrieval based on user ID. It also includes methods to delete old short-lived tokens and update the last connection date
7. `class AuthenticationToken` -It represents an entity for authentication tokens in the database. It includes fields for the token ID, user ID, whether the session is long-lasting, and timestamps for the creation and last connection.
8. `class BaseFunction` -It represents an entity for base functions in the database
9. `class RoleBaseFunctionDao` -It provides methods to interact with role base functions in the database
10. `class ThreadLocalContext` - It manages context associated with a user request, stored in a ThreadLocal. It provides methods to retrieve and manipulate the entity manager within the context.

11. `class TokenBasedSecurityFilter` - It is a servlet filter that authenticates users based on authentication tokens stored in cookies. It verifies the token's validity, injects either an authenticated or anonymous user into the request attributes accordingly, and updates the token's last connection date.
12. `class SortCriteria` - It represents the sorting criteria of a query, including the index of the column to sort and whether the sorting order is increasing or decreasing.
13. `class PaginatedList` - It manages paginated data with details such as page size, offset, total result count, and the list of records for the current page
14. `class PaginatedLists` - It provides utilities for managing paginated lists, including methods for creating paginated lists with customizable parameters, executing count queries to determine the total number of results, and executing paginated queries to retrieve data for the current page.
15. `class Constants` - It hold application constants such as default locale, timezone, theme, administrator's default password, and default generic user role.
16. `class ValidationUtil` - It provides methods for validating parameters, including checks for null, string length, email format, URL format, alphanumeric characters, and date parsing. It also includes validation for locale and theme IDs, throwing client exceptions in case of validation failure.

OOP Concepts

1. **Abstraction**: Classes like BaseResource, User, UserDao encapsulate functionalities, hiding implementation details.
2. **Inheritance**: UserResource inherits from BaseResource, promoting code reuse and ensuring consistent behavior.
3. **Polymorphism**: Method overloading is seen in UserResource with multiple update() and delete() methods accepting different parameters.
4. **Encapsulation**: Data and behavior are encapsulated within classes, like private attributes and public getter/setter methods in User.
5. **Association**: Various classes have associations indicating relationships, like UserResource using AuthenticationToken and AuthenticationTokenDao.
6. **Dependency**: Classes rely on each other, such as UserDao depending on ThreadLocalContext and ValidationUtil for database operations and validation.
7. **Aggregation**: Aggregation relationships exist, like UserDao aggregating UserDto and PaginatedLists creating PaginatedList.

Strengths:

1. The inclusion of token-based security and password hashing mechanisms enhances the security of user authentication processes, mitigating risks associated with unauthorized access or data breaches.
2. The ValidationUtil class provides comprehensive validation utilities for ensuring the integrity and validity of user-provided data
3. The UserDao class leverages efficient data access strategies, including paginated queries and sorting capabilities. These features optimize data retrieval performance, by fetching only the necessary subset of records and sorting them based on specified criteria.

Weaknesses:

1. The implementation of role-based access control (RBAC) functionalities, while crucial for security, may introduce complexity in managing user roles and permissions, potentially leading to administrative overhead.
2. It lacks advanced features for user profile customization, such as avatar uploads, bio descriptions, or personalized settings.

Task 2 : Analysis

Methodology

We started our initial exploration of the code using Sonarqube, which flagged numerous issues. However, many of these were trivial or insignificant. Despite this, we identified some valuable insights on smells in the "blocker" and "critical" categories.

Subsequently, we employed Designite's IntelliJ plugin, which successfully pinpointed several design smells and provided essential metrics on the spot. Notably, it drew our attention to classes such as `BookResource.java`, `UserResource.java`, and `FaceBookService.java`, which exhibited unusually high code metrics, thereby uncovering a plethora of design issues.

Additionally, we utilized SonarLint's VS Code extension to receive live feedback while refactoring, ensuring a smoother development process. Designite employed static analysis techniques to scrutinize the source code, identifying potential design flaws efficiently.

We also leveraged ChatGPT cautiously, verifying all design smells manually, which proved to be quite helpful. Furthermore, we experimented with different prompts to enhance its performance - more on this later.

Delving deeper into the code, we pinpointed precise pain points and deviations from best practices. To complement our findings, we cross-referenced our manually filtered smells with various sources:

- Lecture slides
- *A Taxonomy of Software Smells*
- *Refactoring for Software Design Smells Managing Technical Debt* by Girish Suryanarayana, Ganesh Samarthyam, Tushar Sharma

Regarding code metrics, Sonarqube and Designite provided preliminary insights. Additionally, we employed tools such as CodeMR, CheckStyle, and PDM to obtain comprehensive metrics, as outlined in Task 2B below. Upon exploring Checkstyle's metrics, we found most detected flaws to be pedantic formatting issues, overlooking the broader context captured by PMD, which highlighted major smells more effectively.

Task 2A : Design Smells

Cyclic Dependent Modularization

This smell arises when two or more abstractions depend on each other directly or indirectly (creating a tight coupling between the abstractions)

Indication: Upon making the dependency graph of the abstractions, there is a cycle detected.

Causes:

- Improper responsibility realization - Often, when some of the members of an abstraction are wrongly misplaced in another abstraction, the members may refer to each other, resulting in a

cyclic dependency between the abstractions.

- Passing a self-reference - A method invocation from one abstraction to another often involves data transfer. If instead of explicitly passing only the required data, the abstraction passes its own reference (for instance, via “this”).

Impact:

- Makes it difficult to understand, make changes, and maintain software, and can lead to issues like tight coupling, hindered *testability*, and *scalability* challenges, often indicating a flawed design approach

Code Smell	Description	Examples from Code	Fix
Singleton Pattern Abuse	Misuse of the singleton design pattern - the class is restricted to having only one instance	AppContext initialization in `AppContext.java`	Create a new class that contains an instance of AppContext

Multi-faceted Abstraction

This smell arises when an abstraction has more than one responsibility assigned to it.

Indication: A single class or module has multiple responsibilities or concerns, rather than adhering to the principle of single responsibility - which says an abstraction should have a single well-defined responsibility and that responsibility should be entirely encapsulated within that abstraction.

Causes:

- General-purpose abstractions - When designers introduce an abstraction with a generic name (examples: Node, Component, Element, and Item), it often becomes a “placeholder” for providing all the functionality related (but not necessarily belonging) to that abstraction.
- Evolution without periodic refactoring - When a class undergoes extensive changes over a long period of time without refactoring, other responsibilities start getting introduced in these classes and design decay starts.
- The burden of processes - Sometimes the viscosity of the software and environment serves to discourage the adoption of good practice.
- Mixing up concerns - When designers don’t give sufficient attention to the separation of different concerns

Impact:

- Multifaceted Abstraction increases cognitive load, impacting *understandability*.
- *Changeability* and *extensibility* are hindered as modifications may affect unrelated responsibilities, leading to ripple effects.
- *Reusability* is reduced due to the necessity of using the entire abstraction, potentially causing costly overhead.
- Testing becomes challenging when responsibilities are entwined, impacting **testability**.
- Modifications to intertwined responsibilities may lead to unpredictable runtime problems, affecting *reliability*.

Code Smell	Description	Examples from Code	Fix
Separation of Concerns	Class/Module having too many in-cohesive responsibilities leading to tight coupling.	`Constants.java` - It violates SRP by having all constants used in the entire code base in one place.	Include Constants in their respective classes. If these were meant to be global variable, making a separate class is not a solution.
Large Class	A module having too many lines of code and has multiple responsibilities.	`BookResource.java` - This class is too large and performs a lot of responsibilities	Split into smaller classes with specific responsibilities.
		`UserResource.java` - This class is too large and performs a lot of responsibilities	Split into smaller classes with specific responsibilities.

Broken Modularization

This smell arises when data and/or methods that ideally should have been localized into a single abstraction are separated and spread across multiple abstractions.

Indication: The ‘data class’ code smell is an indication of Broken Modularization. There is a class with only data and no methods.

Causes:

- Procedural thinking in object-oriented languages : This is because procedural programming languages developers assume that data must be separated from the functions that process the data so that in OOP, developers break it into separate classes.
- Lack of knowledge of existing design : In some cases, especially in large companies, there are many classes that a developer should be able to work on apart from the part he is working on. He didn't know about this, so the developer placed the members/methods in the wrong location, which in turn caused the smell.

Impact:

- Leads to poor maintainability, readability and extensibility.

Code Smell	Description	Examples from Code	Fix
Data Class	Class containing only data or parameters, and no methods	`MimeType.java` - It has only data and no methods.	Merge into appropriate classes

Primitive Obsession (Missing Abstraction)

This smell arises when clumps of data or encoded strings are used instead of creating a class or an interface.

Indication: Due to the lack of abstraction, the associated data and behaviour is spread across other abstractions. Clumps of data items that occur together in lots of places.

Causes:

- Inadequate design analysis - When careful thought is not applied during design, it is easy to overlook creating abstractions and use primitive type values or strings to “get the work done.”
- Misguided focus on minor performance gains - like using arrays directly without abstraction

Impact:

- Understandability: Lack of abstractions and scattered logic make the design difficult to comprehend.
- Changeability and Extensibility: Missing abstractions lead to difficulty in identifying and implementing changes, requiring modifications in multiple places.
- Reusability and Testability: Absence of domain abstractions and scattered logic impairs both reusability and testability.
- Reliability: Without abstractions, data and behavior dispersion across the codebase compromises data integrity, affecting reliability.

Code Smell	Description	Examples from Code	Fix
Data Clumps	Clumps of data items that occur together in lots of places.	`BookResource.java` - This can be observed every time `Book` is created and populated or every time an API request is made	Do the required abstraction
		`UserResource.java` - This can be observed every time `User` is created and populated or every time an API request is made	Do the required abstraction

Unnecessary Abstraction

This smell occurs when an abstraction that is actually not needed is introduced in a software design.

Indication: The lazy class’ code smell is an indication of Unnecessary Abstraction. This smell occurs where a class is not doing enough i.e. it does not have a concrete responsibility

Causes:

- Procedural thinking in object-oriented languages : developers are not familiar with the OOP paradigm so they make mistakes when designing classes.

Impact:

- Over-engineering adds complexity : developers make designs that are not really needed/overkill, aka ambitious ideas

Code Smell	Description	Examples from Code	Fix
Lazy Class	Class is not doing enough i.e. it does not have a concrete responsibility.	MimeType class in `MineType.java`	Remove class/merge with existing class

Spaghetti Code (leading to Inconsistent Abstraction)

This smell occurs when Spaghetti code leads to inconsistent abstraction, resulting in reduced maintainability, increased bug proneness, and hindered collaboration among developers.

Indication: A telltale sign of this smell is the 'conditional complexity' code smell. It arises when code becomes overly complex and difficult to understand due to its lack of structure and organization.

Causes:

- When abstraction is not used consistently in the code, it can become difficult to understand how the different parts of the system interact with each other, leading to a tangled mess of code that resembles a plate of spaghetti.
- Impact: Can lead to maintenance problems and make it difficult to add new features or modify the code.

Impact:

- Makes it harder to understand and modify, diminishing the codebase's maintainability. It is difficult to isolate and address issues due to the lack of clear abstraction boundaries.
- The entanglement of abstraction levels in spaghetti code raises the likelihood of introducing bugs. Changes made to one part of the code may unknowingly impact unrelated sections, leading to unexpected and difficult-to-trace errors.

Code Smell	Description	Examples from Code	Fix
Conditional Complexity	Having too many conditional operations (if, else) makes it harder to understand, and high probability of breaking, testing also becomes difficult	`TokenBasedSecurityFilter Class` - function `doFilter()`	Remove class/merge with existing class
		`UserBookTag Class` - `equals()` function	Extract Methods to reduce conditional complexity
		`BookImportAsyncListener Class` - `run()` , `on()` functions	Extract Methods to reduce conditional complexity
		`BookResource Class` - `add()` and `update()` functions	Extract Methods to reduce conditional complexity
Low Cohesion	High Cyclomatic, NPath complexities	`BookDataService Class` - `searchBook()` , `searchBookWithGoogle()` , `searchBookWithOpenLibrary()` functions	Extract Methods to reduce conditional complexity

Deficient Encapsulation

This smell occurs when the declared accessibility of one or more members of an abstraction is more permissive than required.

Indication: One or more members is not having required protection (eg: public)

Causes:

- Easier testability, procedural thinking (expose data as global variables), quick fixes.

Impact:

- Affects code changeability, extensibility, and reliability.

Code Smell	Description	Examples from Code	Fix
Unhidden Public Constructor	Utility classes, which are collections of static members, are not meant to be instantiated	`Class Constants` in `Constants.java`	Add a private constructor
		`Class ConfigUtil` in `ConfigUtil.java`	Add a private constructor
		`Class DirectoryUtil` in `DirectoryUtil.java`	Add a private constructor

... And the Rest

After reviewing Snoarqube and others, we've pinpointed certain code issues that don't neatly categorize as typical design smells (per se) but still significantly hamper overall code quality, particularly in terms of their impact on metrics.

Code Smell	Description	Examples from Code	Fix
Duplicated code	Repetition of code that could’ve been reused. This violates the DRY (Don't Repeat Yourself) principle.	`UserAppDao.java` in Books Core: Repetition of code (the construction of the list part) `ValidationUtil.java` in Books Web-Common	- Make the code reusable to remove repetiition - Define a constant instead of duplicating this literal "{0} must be set" 3 times
Dead Code	Unused import, checks etc	`BookDao.java` in Books Core `AppResource.java` in Books Web `BaseResource.java` in Books Web	- Remove this unused import `javax.persistence.EntityTransaction` - Remove this unnecessary null check; "instanceof" returns false for nulls.
Encapsulation Violation	Public constructor	`PaginatedList.java` ValidationUtil class in `ValidationUtil.java`	- Add a private constructor to hide the implicit public one
Redundant Code	Useless assignment to local variables	`TagDao.java` , `UserBookDao.java`, `UserDao.java`	- Remove the useless assignments to local variables eg. “i”.
Unreachable Exception	Declaration of thrown exceptions which can not be thrown from method’s body	`UserBookDao.java`, `BaseResource.java`	- Remove the declaration of thrown exception `java.lang.Exception`, as it cannot be thrown from method's body. - Remove the declaration of thrown exception `org.codehaus.jettison.json.JSONException`, as it cannot be thrown from method's body
Naming convention not followed	Constant name not following regex `^[A-Z][A-Z0-9](_[A-Z0-9]+)$`	`ThreadLocalContext.java`	- Rename

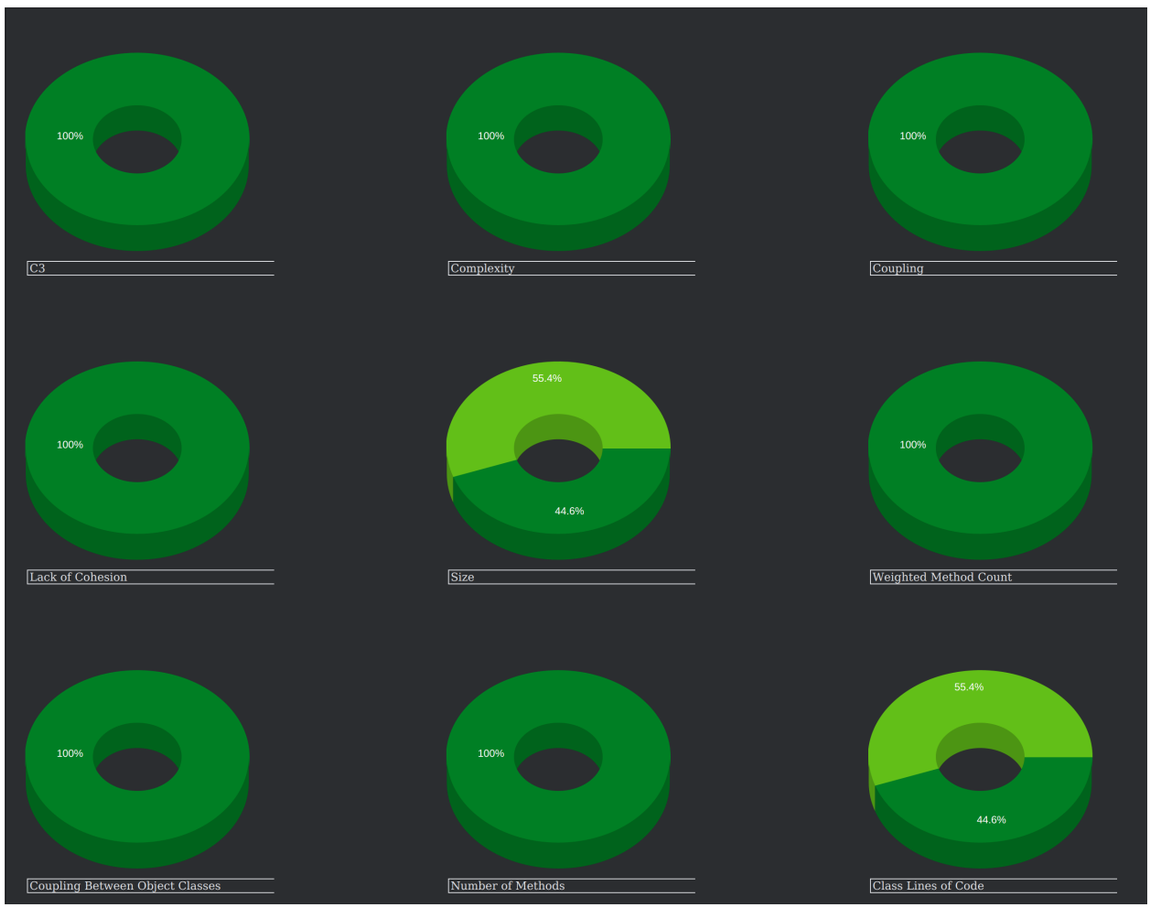
Code Smell	Description	Examples from Code	Fix
Long Method	Method is quite long and performs multiple tasks such as building queries, executing them, and assembling results. Long methods can be harder to understand, maintain, and test.	Method `findAll` starting at Line 207 in `UserDao.java` .	- Break down the method into smaller, more manageable chunks by extracting separate methods for query building, execution, and result assembly.

Task 2B : Code Metrics

CodeMR

We only run it over `books-core` and `books-web` since they encompass all of the functionalities that we've been asked to consider.

We first present the metrics graphs given by CodeMR:



Metrics Graph for Books-Core

Metrics Graph for Books-Web

Some other metrics given by CodeMR

Metric	Books-Core	Books-Web
Total Lines of Code	2523	1022
Number of classes (Not strictly a metric)	71	10

The Total Lines of Code is a good measure -

- it is the number of all nonempty, non-commented lines of code in the project.
- A higher LOC might indicate additional complexity.

Detailed metrics given by CodeMR are:

ID	CLASS	COUPLING	COMPLEXITY	LACK OF COHESION	SIZE	LOC	COMPLEXITY	COUPLING	LACK OF COHESION	SIZE
1	BookDataService					146	low	low	low	low-medium
2	UserAppDao					119	low	low	low	low-medium
3	DbOpenHelper					110	low	low	low	low-medium
4	FacebookService					105	low	low	low	low-medium
5	UserDao					98	low	low	low	low-medium
6	TagDao					84	low	low	low	low-medium
7	UserBookDao					81	low	low	low	low-medium
8	UserBook					75	low	low	low	low-medium
9	UserContactDao					74	low	low	low	low-medium
10	UserApp					71	low	low	low	low-medium
11	User					70	low	low	low	low-medium
12	Book					70	low	low	low	low-medium
13	BookImportAsyncLi...					68	low	low	low	low-medium
14	UserContact					65	low	low	low	low-medium
15	UserBookTag					58	low	low	low	low-medium

Detailed Metrics on Books-Core

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

Detailed Metrics on Books-Web

We've skipped metrics that were returned as "low", i.e., have no need to be considered since they are already good.

The metrics of concern in the above diagrams are

- Class Lines of Code (LOC):
  - The number of all nonempty, non-commented lines of the body of the class.
  - CLOC is a measure of size and also indirectly related to the class complexity.
- Complexity:
  - Implies being difficult to understand and describes the interactions between a number of entities.
  - Higher levels of complexity in software increase the risk of unintentionally interfering with interactions and so increases the chance of introducing defects when making changes.
- Lack of Cohesion:
  - Measure how well the methods of a class are related to each other.

    - High cohesion (low lack of cohesion) tend to be preferable because high cohesion is associated with several desirable traits of software including robustness, reliability, reusability, and understandability.
- Coupling:
    - A high value of coupling would indicate that a change in a class will force a ripple effect of changes in other classes.
    - Moreover, high coupling might make it harder to reuse a class because dependent classes must be included.

CodeMR has not revealed any issues apart from CLOC in Books-Web.

So, we should move towards more concrete - specific - ways of finding smells.

CheckStyle

Boolean Complexity

```
books-core/src/main/java/com/sismics/books/core/util/mime/MimeTypeUtil.jav
a:48:16:
         Boolean expression complexity is 7 (max allowed is 3).
[books-core/src/main/java/com/sismics/util/log4j/MemoryAppender.java:107:1
3:
        Boolean expression complexity is 5 (max allowed is 3).
```

- Restricts the number of boolean operators (&&, ||, &, | and ^) in an expression.
- Too many conditions leads to code that is difficult to read and hence debug and maintain so having a low boolean complexity is good.

Class Data Abstraction Coupling

```
books-core/src/main/java/com/sismics/books/core/listener/async/BookImportA
syncListener.java:37:1:
        Class Data Abstraction Coupling is 9 (max allowed is 7) classes [B
ookDao, CSVReader, Date, FileReader, Runnable, Tag, TagDao, UserBook, User
BookDao]. [ClassDataAbstractionCoupling]

books-core/src/main/java/com/sismics/books/core/service/BookDataService.ja
va:47:1:
        Class Data Abstraction Coupling is 8 (max allowed is 7) classes [B
ook, BufferedInputStream, Callable, DateTimeFormatterBuilder, FutureTask,
ObjectMapper, Runnable, URL]. [ClassDataAbstractionCoupling]
books-core/src/main/java/com/sismics/books/core/model/context/AppContext.j
ava:24:1:
        Class Data Abstraction Coupling is 9 (max allowed is 7) classes [A
syncEventBus, BookDataService, BookImportAsyncListener, DeadEventListener,
EventBus, FacebookService, LinkedBlockingQueue, ThreadPoolExecutor, UserAp
pCreatedAsyncListener]. [ClassDataAbstractionCoupling]


books-web/src/main/java/com/sismics/books/rest/resource/UserResource.java:
50:1:
        Class Data Abstraction Coupling is 13 (max allowed is 7) classes
```

```
[AuthenticationToken, AuthenticationTokenDao, ClientException, Date, Forbi
ddenClientException, JSONArray, JSONObject, NewCookie, RoleBaseFunctionDa
o, ServerException, SortCriteria, User, UserDao]. [ClassDataAbstractionCou
pling]

books-web/src/main/java/com/sismics/books/rest/resource/BookResource.java:
66:1:
        Class Data Abstraction Coupling is 18 (max allowed is 7) classes
[Book, BookDao, BookImportedEvent, ClientException, Date, File, FileInputS
tream, FileOutputStream, ForbiddenClientException, JSONObject, ServerExcep
tion, SimpleDateFormat, SortCriteria, TagDao, UserBook, UserBookCriteria,
UserBookDao, UserDao]. [ClassDataAbstractionCoupling]

books-web/src/main/java/com/sismics/books/rest/resource/ConnectResource.ja
va:44:1:
        Class Data Abstraction Coupling is 8 (max allowed is 7) classes [C
lientException, ForbiddenClientException, JSONObject, UserApp, UserAppCrea
tedEvent, UserAppDao, UserContactCriteria, UserContactDao]. [ClassDataAbst
ractionCoupling]
```

- Measures the number of instantiations of other classes within the given class or record.
- Generally speaking, any data type with other data types as members or local variable that is an instantiation (object) of another class has data abstraction coupling
- A higher DAC can indicate higher complexity

Class Fan Out Complexity

```
books-core/src/main/java/com/sismics/books/core/service/BookDataService.ja
va:47:1:
        Class Fan-Out Complexity is 21 (max allowed is 20). [ClassFanOutCo
mplexity]

books-core/src/main/java/com/sismics/books/core/service/FacebookService.ja
va:43:1:
        Class Fan-Out Complexity is 22 (max allowed is 20). [ClassFanOutCo
mplexity]


books-web/src/main/java/com/sismics/books/rest/resource/UserResource.java:
50:1:
        Class Fan-Out Complexity is 29 (max allowed is 20). [ClassFanOutCo
mplexity]

books-web/src/main/java/com/sismics/books/rest/resource/BookResource.java:
66:1:
        Class Fan-Out Complexity is 40 (max allowed is 20). [ClassFanOutCo
mplexity]

books-web/src/main/java/com/sismics/books/rest/resource/ConnectResource.ja
va:44:1:
        Class Fan-Out Complexity is 25 (max allowed is 20). [ClassFanOutCo
mplexity]
```

- Checks the number of other types a given class/record/interface/enum/annotation relies on
- Again, this can indicate how complex a class is.

Cyclomatic Complexity

```
books-core/src/main/java/com/sismics/books/core/listener/async/BookImportA
syncListener.java:57:13:
        Cyclomatic Complexity is 17 (max allowed is 10). [CyclomaticComple
xity]

books-core/src/main/java/com/sismics/books/core/service/BookDataService.ja
va:167:5:
        Cyclomatic Complexity is 11 (max allowed is 10). [CyclomaticComple
xity]

books-core/src/main/java/com/sismics/books/core/service/BookDataService.ja
va:227:5:
        Cyclomatic Complexity is 17 (max allowed is 10). [CyclomaticComple
xity]

books-core/src/main/java/com/sismics/books/core/dao/jpa/UserBookDao.java:1
31:5:
        Cyclomatic Complexity is 11 (max allowed is 10). [CyclomaticComple
xity]

books-core/src/main/java/com/sismics/books/core/util/mime/MimeTypeUtil.jav
a:39:5:
        Cyclomatic Complexity is 22 (max allowed is 10). [CyclomaticComple
xity]

books-core/src/main/java/com/sismics/books/core/util/TransactionUtil.java:
27:5:
        Cyclomatic Complexity is 12 (max allowed is 10). [CyclomaticComple
xity]

books-core/src/main/java/com/sismics/util/jpa/DbOpenHelper.java:60:5:
        Cyclomatic Complexity is 12 (max allowed is 10). [CyclomaticComple
xity]

books-core/src/main/java/com/sismics/util/ResourceUtil.java:33:5:
        Cyclomatic Complexity is 13 (max allowed is 10). [CyclomaticComple
xity]


books-web/src/main/java/com/sismics/books/rest/resource/BookResource.java:
159:5:
        Cyclomatic Complexity is 19 (max allowed is 10). [CyclomaticComple
xity]

books-web/src/main/java/com/sismics/books/rest/resource/BookResource.java:
273:5:
        Cyclomatic Complexity is 24 (max allowed is 10). [CyclomaticComple
xity]
```

- It is a measure of the minimum number of possible paths through the source and therefore the number of required tests
- The complexity is equal to the number of `decision points + 1`
- The higher the number of decision points, the more complex the code and the more difficult to refactor

JavaNCSS

```
books-core/src/main/java/com/sismics/books/core/listener/async/BookImportA
syncListener.java:49:5:
        NCSS for this method is 60 (max allowed is 50). [JavaNCSS]

books-core/src/main/java/com/sismics/books/core/listener/async/BookImportA
syncListener.java:57:13:
        NCSS for this method is 56 (max allowed is 50). [JavaNCSS]

books-core/src/main/java/com/sismics/util/jpa/DbOpenHelper.java:60:5:
        NCSS for this method is 52 (max allowed is 50). [JavaNCSS]


books-web/src/main/java/com/sismics/books/rest/resource/BookResource.java:
159:5:
        NCSS for this method is 60 (max allowed is 50). [JavaNCSS]

books-web/src/main/java/com/sismics/books/rest/resource/BookResource.java:
273:5:
        NCSS for this method is 58 (max allowed is 50). [JavaNCSS]

books-web/src/test/java/com/sismics/books/rest/TestBookResource.java:34:5:
        NCSS for this method is 150 (max allowed is 50). [JavaNCSS]

books-web/src/test/java/com/sismics/books/rest/TestTagResource.java:27:5:
        NCSS for this method is 85 (max allowed is 50). [JavaNCSS]

books-web/src/test/java/com/sismics/books/rest/TestUserResource.java:28:5:
        CSS for this method is 152 (max allowed is 50). [JavaNCSS]

books-web/src/test/java/com/sismics/books/rest/TestUserResource.java:235:
5:
        NCSS for this method is 62 (max allowed is 50). [JavaNCSS]

books-web/src/test/java/com/sismics/books/rest/TestConnectResource.java:5
7:5:
        NCSS for this method is 121 (max allowed is 50). [JavaNCSS]
```

- Determines complexity of methods, classes and files by counting the Non Commenting Source Statements (NCSS)
- A large NCSS number often means that a method or class has too many responsibilities and/or functionalities which should be decomposed into smaller units

NPathComplexity

```
books-core/src/main/java/com/sismics/books/core/listener/async/BookImportA
syncListener.java:57:13:
    NPath Complexity is 5,409 (max allowed is 200). [NPathComplexity]

books-core/src/main/java/com/sismics/books/core/service/BookDataService.ja
va:167:5:
    NPath Complexity is 384 (max allowed is 200). [NPathComplexity]

books-core/src/main/java/com/sismics/books/core/service/BookDataService.ja
va:227:5:
    NPath Complexity is 15,552 (max allowed is 200). [NPathComplexity]

books-core/src/main/java/com/sismics/books/core/service/FacebookService.ja
va:136:5:
    NPath Complexity is 272 (max allowed is 200). [NPathComplexity]

books-core/src/main/java/com/sismics/books/core/dao/jpa/UserBookDao.java:1
31:5:
    NPath Complexity is 240 (max allowed is 200). [NPathComplexity]

books-core/src/main/java/com/sismics/books/core/util/TransactionUtil.java:
27:5:
    NPath Complexity is 385 (max allowed is 200). [NPathComplexity]

books-core/src/main/java/com/sismics/util/jpa/DbOpenHelper.java:60:5:
    NPath Complexity is 1,093 (max allowed is 200). [NPathComplexity]

books-core/src/main/java/com/sismics/util/ResourceUtil.java:33:5:
    NPath Complexity is 486 (max allowed is 200). [NPathComplexity]


books-web/src/main/java/com/sismics/books/rest/resource/BookResource.java:
159:5:
    NPath Complexity is 64,512 (max allowed is 200). [NPathComplexity]

books-web/src/main/java/com/sismics/books/rest/resource/BookResource.java:
273:5:
    NPath Complexity is 358,400 (max allowed is 200). [NPathComplexity]
```

- The NPATH metric computes the number of possible execution paths through a function(method)
- The NPATH metric was designed base on Cyclomatic complexity to avoid problem of Cyclomatic complexity metric like nesting level within a function(method) and has the same use as cyclomatic complexity.

We also have pmd metric available - please check out the `code-metrics/before-refactoring/pmd` directopry in our GitHub Repo. We have chosen not to include them as there are no advantages compared to CheckStyle.

In conclusion, we've particularly presented the following metrics above:

- Lines of Code
  - and related - Class Lines of Code

- Complexity (as defined as CodeMR)
- Lack of Cohesion
- Coupling
- Boolean Complexity
- Class Data Abstraction Complexity
- Class Fan Out Complexity
- Cyclomatic Complexity
- JavaNCSS
- NPath Complexity

Tools Used

- CodeMR
- CheckStyle
- PMD

Implications Discussions

Has been included wherever we have introduced the metric.

Task 3 Refactoring

Methodology

A systematic approach was adopted within our team to address design flaws identified in our code base. One individual would initiate the process by raising an issue on GitHub, highlighting the specific problem area. They would then assign the task to another team member best suited to address it.

Effective communication was maintained throughout the process using GitHub comments, allowing for seamless coordination among team members. Once the necessary changes were implemented, the responsible individual would request a review from another team member. This ensured that each modification was thoroughly examined before proceeding further.

To maintain the integrity of our code base, rigorous testing procedures were implemented at each stage of the refactoring process. Following every commit or merge, comprehensive manual testing was conducted to ensure that the functionality remained intact and that the changes did not introduce any unintended consequences.

Task 3A : Design Smells

Issues on GitHub

Issue	Description	Fix
Poor separation of concerns in Constants.java	Class/Module having too many in-cohesive responsibilities leading to tight coupling. It violates SRP by having all constants used in the entire code base in one place.	Since the variables inside the **Constants.java** file were public and being used very sparsely, I took the design decision to include the variables as private static variables in the files itself, this ensures both

Issue	Description	Fix
		prevention of misuse of data and also follows SRP. Since there are no common Constants across files, this ensures a smooth code flow.
High Conditional Complexity in TokenBasedSecurityFilter.java	Having too many conditional operations (if, else) makes it harder to understand, and high probability of breaking, testing also becomes difficult due to this. Refactor function **doFilter()** . Remove class/merge with existing class	I refactored by reducing the depth of conditional statements. I added 3 new functions - `getAuthToken()` - Method to handle expired tokens, `handleExpiredToken()` - Optimized token extraction using direct access to cookies and `handleInvalidUser()` - Consolidated logic for handling invalid users to modularize `doFilter()`
Singleton Pattern Abuse in AppContext.java	There is a misuse of the singleton design pattern - the class is restricted to having only one instance. Fix AppContext initialization in AppContext.java .	
Large Class and Missing Abstraction in BookResource.java	Clumps of data items that occur together in lots of places. The code is very unorganized. In BookResource, this can be observed every time Book is created and populated or every time an API request is made. This class is also very large and performs a lot of responsibilities.	So, we have broken the BookResource class was violating SRP principle. It was acting as the main class for all Book Related functionalities. We created separate supporting classes and have removed BookResouce completely. The application functionality has also been tested out
Large Class and Missing Abstraction in UserResource.java	Clumps of data items that occur together in lots of places. The code is very unorganized. In UserResource, this can be observed every time User is created and populated or every time an API request is made. This class is also very large and performs a lot of responsibilities.	Fixed Singleton Pattern issue by creating a new Private class method
DataClass in MimeType.java	Class MimeType containing only data or parameters, and no methods.	This issue is automatically resolved with Issue#13 where I merged the MimeType class with MimeTypeUtil class for the reasons mentioned in that Issue.
LazyClass in MimeType.java	Class is not doing enough i.e. it does not have a concrete responsibility	I found that MimeType Class was always used along with MimeTypeUtil Class, hence I've Merged the Lazy MimeType Class into MimeTypeUtil and made the necessary changes in the rest of the repository. I've tested and it works.
High Conditional Complexity in UserBookTag.java	Having too many conditional operations (if, else) makes it	Issues: The code is very unorganized. In UserResource,

Issue	Description	Fix
	harder to understand, and high probability of breaking, testing also becomes difficul	this can be observed every time User is created and populated or every time an API request is made. This class is also very large and performs a lot of responsibilities. Changes: Mada a package userresource inside resource itself, to divide the long class into 7 functionalities. The imports in each java file are modified to include only those being actually used, so no redundant imports. Routes specified in each java file for each functionality in the package. All the methods utilizing the same routes defined in the same class and thus the same file
High Conditional Complexity in BookImportAsyncListener.java	Having too many conditional operations (if, else) makes it harder to understand, and high probability of breaking, testing also becomes difficult. Refactor functions run(), on() in class BookImportAsyncListener.	I have reduced conditional complexity (depth of conditional statements) by modularizing the function and giving single responsibilities to auxiliary functions.
High Conditional Complexity in BookResource.java	Having too many conditional operations (if, else) makes it harder to understand, and high probability of breaking, testing also becomes difficult. Refactor functions add() in class BookResouce.	BookResource.java has been refactored,
Low Cohesion in BookDataService.java	High Cyclomatic, NPath complexities in the class.BookDataService Class - searchBookWithOpenLibrary() functions need refactoring.	Multiple issues like redundant code for establishing connection, setting Book isbn's, Long method etc. Made reusable methods like establishConnection(), getRootNode(). Long method issue resolved by making methods for building books for Google, OpenLibrary; ISBN setting.
Unhidden Public Constructor in Constants.java, ConfigUtil.java and DirectoryUtil.java	Utility classes, which are collections of static members, are not meant to be instantiated.	

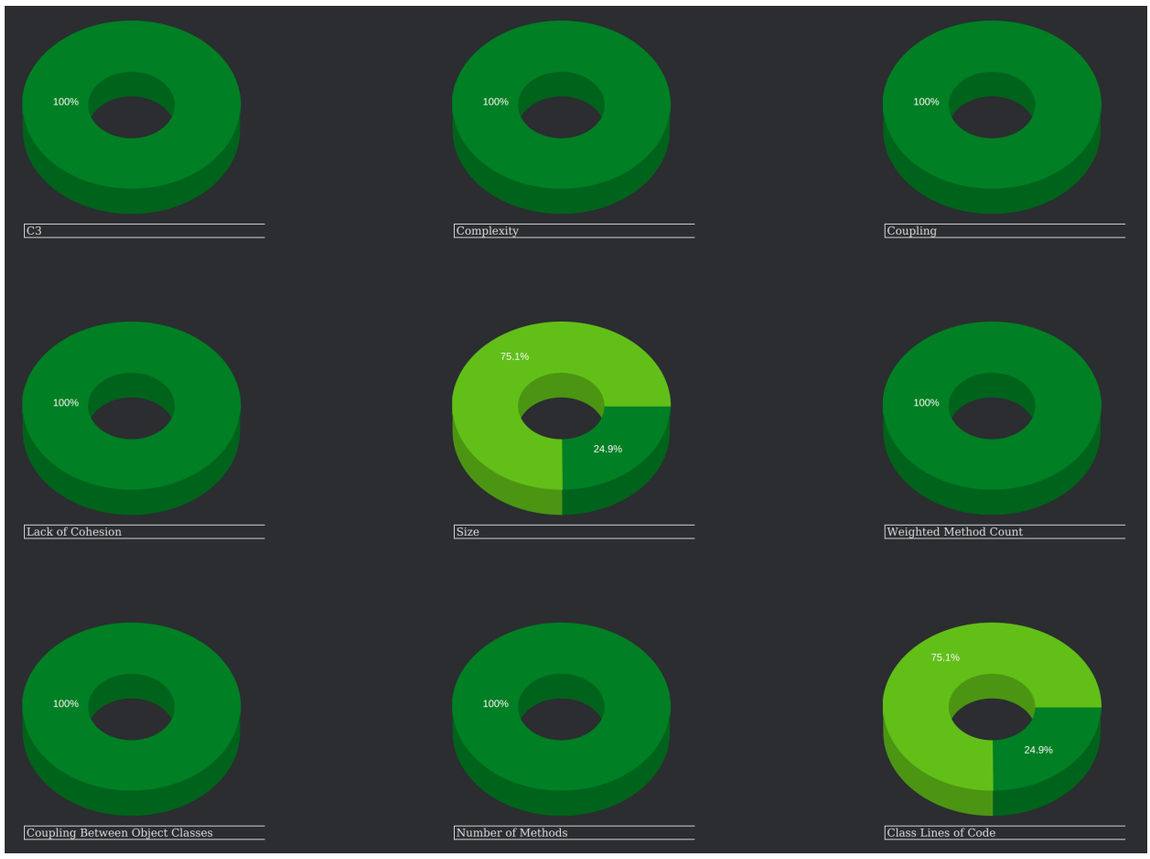
Task 3B : Code Metrics

We evaluated the code metrics on our refactored code. Here are the results:
We only run it over
`books-core` and `books-web` since they encompass all of the functionalities that we've been asked to consider.

CodeMR

Detailed metrics given by CodeMR are:

Graph view of books-core



Grapjs view of books-web

Some other metrics given by CodeMR

Metric	Books-Core	Books-Web
Total Lines of Code	2539 ~~2523~~	1074 ~~1022~~
Number of classes (Not strictly a metric)	70 ~~71~~	21 ~~10~~

The number of lines of code have increased as we broke down complex things during our refactoring. The same goes for the increase in number of classes as we refactored god classes.

Detailed metrics given by CodeMR are:

ID	CLASS	COUPLING	COMPLEXITY	LACK OF COHESION	SIZE	LOC	COMPLEXITY	COUPLING	LACK OF COHESION	SIZE
1	BookDataService					166	low	low	low	low-medium
2	UserAppDao					119	low	low	low	low-medium
3	DbOpenHelper					110	low	low	low	low-medium
4	FacebookService					102	low	low	low	low-medium
5	UserDao					99	low	low	low	low-medium
6	BookImportAsyncLi...					84	low	low	low	low-medium
7	TagDao					84	low	low	low	low-medium
8	UserBookDao					81	low	low	low	low-medium
9	UserBook					75	low	low	low	low-medium
10	UserContactDao					74	low	low	low	low-medium
11	UserApp					71	low	low	low	low-medium
12	User					70	low	low	low	low-medium
13	Book					70	low	low	low	low-medium
14	UserContact					65	low	low	low	low-medium
15	MemoryAppender					52	low	low	low	low-medium

Metrics on books-core

ID	CLASS	COUPLING	COMPLEXITY	LACK OF COHESION	SIZE	LOC	COMPLEXITY	COUPLING	LACK OF COHESION	SIZE
1	BookTagUpdate					183	low	low	low	low-medium
2	ConnectResource					131	low	low	low	low-medium
3	ManualBookAddition					109	low	low	low	low-medium
4	UserHandler					108	low	low	low	low-medium
5	TagResource					85	low	low	low	low-medium
6	UserUtils					70	low	low	low	low-medium
7	AppResource					66	low	low	low	low-medium
8	ListBooks					55	low	low	low	low-medium
9	UserSession					39	low	low	low	low
10	AddBook					30	low	low	low	low
11	ImportFile					29	low	low	low	low
12	UserList					28	low	low	low	low
13	UserLogout					25	low	low	low	low
14	UserLogin					24	low	low	low	low
15	TextPlainMessageB...					24	low	low	low	low

Metrics on books-web

There have been no significant shifts after refactoring in the metrics given by CodeMR. **However, we use CheckStyle to show changes in Metrics that matterx.**

Checkstyle

BooleanExpressionComplexity:

For `book-core`:

```
Starting audit...

se-project-1--_14/books-core/src/main/java/com/sismics/util/log4j/MemoryAppe
        Boolean expression complexity is 5 (max allowed is 3). [BooleanExpre

se-project-1--_14/books-core/src/main/java/com/sismics/books/core/util/mime/
        Boolean expression complexity is 7 (max allowed is 3). [BooleanExpre

Audit done.
Checkstyle ends.
```

For `books-web`:

```
Starting audit...
Audit done.
```

CyclomaticComplexity:

For `book-core`:

```
Starting audit...
se-project-1--_14/books-core/src/main/java/com/sismics/util/jpa/DbOpenHelper
        Cyclomatic Complexity is 12 (max allowed is 10). [CyclomaticComplexi

se-project-1--_14/books-core/src/main/java/com/sismics/util/ResourceUtil.jav
        Cyclomatic Complexity is 13 (max allowed is 10). [CyclomaticComplexi

se-project-1--_14/books-core/src/main/java/com/sismics/books/core/util/mime/
        Cyclomatic Complexity is 22 (max allowed is 10). [CyclomaticComplexi

se-project-1--_14/books-core/src/main/java/com/sismics/books/core/util/Trans
        Cyclomatic Complexity is 12 (max allowed is 10). [CyclomaticComplexi

se-project-1--_14/books-core/src/main/java/com/sismics/books/core/dao/jpa/Us
        Cyclomatic Complexity is 11 (max allowed is 10). [CyclomaticComplexi

Audit done.
Checkstyle ends with 5 errors.
```

For `books-web`:

```
Starting audit...
se-project-1--_14/books-web/src/main/java/com/sismics/books/rest/resource/bo
        Cyclomatic Complexity is 24 (max allowed is 10). [CyclomaticComplexi

Audit done.
Checkstyle ends with 1 errors.
```

ClassDataAbstractionCoupling:

For `books-core:`

```
Starting audit...
se-project-1--_14/books-core/src/main/java/com/sismics/books/core/listener/a
        Class Data Abstraction Coupling is 9 (max allowed is 7) classes [Boo

se-project-1--_14/books-core/src/main/java/com/sismics/books/core/model/cont
        Class Data Abstraction Coupling is 9 (max allowed is 7) classes [Asy

se-project-1--_14/books-core/src/main/java/com/sismics/books/core/service/Bo
        Class Data Abstraction Coupling is 8 (max allowed is 7) classes [Boo

Audit done.
Checkstyle ends.
```

For `books-web`:

```
Starting audit...
se-project-1--_14/books-web/src/main/java/com/sismics/books/rest/resource/Co
        Class Data Abstraction Coupling is 8 (max allowed is 7) classes [Cli

se-project-1--_14/books-web/src/main/java/com/sismics/books/rest/resource/us
        Class Data Abstraction Coupling is 8 (max allowed is 7) classes [Cli

se-project-1--_14/books-web/src/main/java/com/sismics/books/rest/resource/bo
        Class Data Abstraction Coupling is 11 (max allowed is 7) classes [Bo

se-project-1--_14/books-web/src/main/java/com/sismics/books/rest/resource/bo
        Class Data Abstraction Coupling is 10 (max allowed is 7) classes [Bo

Audit done.
Checkstyle ends.
```

ClassFanOutComplexity:

For `books-core`:

```
Starting audit...

se-project-1--_14/books-core/src/main/java/com/sismics/books/core/service/Bo
        Class Fan-Out Complexity is 21 (max allowed is 20). [ClassFanOutComp

se-project-1--_14/books-core/src/main/java/com/sismics/books/core/service/Fa
        Class Fan-Out Complexity is 21 (max allowed is 20). [ClassFanOutComp

Audit done.
Checkstyle ends.
```

For `books-web`:

```
Starting audit...
se-project-1--_14/books-web/src/main/java/com/sismics/books/rest/resource/Co
        Class Fan-Out Complexity is 25 (max allowed is 20). [ClassFanOutComp

se-project-1--_14/books-web/src/main/java/com/sismics/books/rest/resource/bo
        Class Fan-Out Complexity is 27 (max allowed is 20). [ClassFanOutComp

Audit done.
Checkstyle ends.
```

JavaNCSS:

For `books-core`:

```
Starting audit...

se-project-1--_14/books-core/src/main/java/com/sismics/util/jpa/DbOpenHelper
        NCSS for this method is 52 (max allowed is 50). [JavaNCSS]

Audit done.
Checkstyle ends.
```

For `books-web`:

```
Starting audit...

se-project-1--_14/books-web/src/main/java/com/sismics/books/rest/resource/bo
        NCSS for this method is 58 (max allowed is 50). [JavaNCSS]

Audit done.
Checkstyle ends.
```

NPathComplexity:

For `books-core`:

```
se-project-1--_14/books-core/src/main/java/com/sismics/util/jpa/DbOpenHelper
        NPath Complexity is 1,093 (max allowed is 200). [NPathComplexity]

se-project-1--_14/books-core/src/main/java/com/sismics/util/ResourceUtil.jav
        NPath Complexity is 486 (max allowed is 200). [NPathComplexity]

se-project-1--_14/books-core/src/main/java/com/sismics/books/core/util/Trans
        NPath Complexity is 385 (max allowed is 200). [NPathComplexity]

se-project-1--_14/books-core/src/main/java/com/sismics/books/core/dao/jpa/Us
        NPath Complexity is 240 (max allowed is 200). [NPathComplexity]

se-project-1--_14/books-core/src/main/java/com/sismics/books/core/service/Bo
        NPath Complexity is 288 (max allowed is 200). [NPathComplexity]

se-project-1--_14/books-core/src/main/java/com/sismics/books/core/service/Fa
```

```
        NPath Complexity is 272 (max allowed is 200). [NPathComplexity]

Audit done.
Checkstyle ends.
```

For `books-web`:

```
se-project-1--_14/books-web/src/main/java/com/sismics/books/rest/resource/bo
        NPath Complexity is 358,400 (max allowed is 200). [NPathComplexity]

se-project-1--_14/books-web/src/main/java/com/sismics/books/rest/resource/bo
        NPath Complexity is 512 (max allowed is 200). [NPathComplexity]

Audit done.
Checkstyle ends.
```

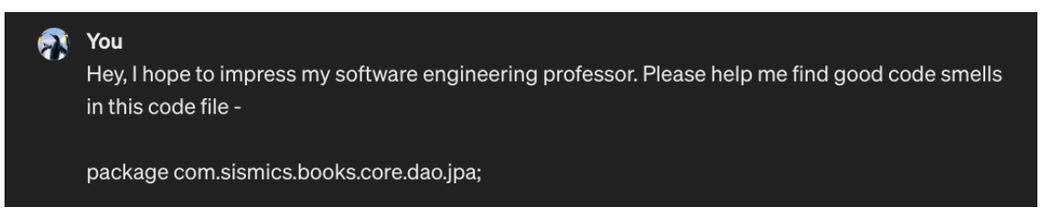
Observations in metrics after refactoring:

- Improved NCSS in `BookImportAsyncListener.java` to under threshold
- Improved cyclomatic complexity in the same file above to under threshold
- Improved cyclomatic complexity in `BookDataService.java` to under threshold
- Improved NPath complexity in the same file from 15K to 200 in once case and got it under bounds in another
- Broke up the class `UserResource` into multiple files - we don;' observe anymore coupling and fan out complexity metrics above threshold
- Broke up the class `BookResource` into multiple files
  - Class Data Abstraction Coupling to 10 from 13
  - Fan Out Complexity to 27 from 29
  - One instance of cyclomatic complexity from 19 to under threshold
  - NCSS to 58 from 60
- A case where we increased the metric - `UserHandler` now has Class Data Abstraction Coupling of 8 (one above threshold)

Task 3C : Leveraging Large Language Models for Refactoring

So we tested refactoring by GPT-3.5 on the following code snippet from `TokenBasedSecurityFilter.java`

This was the prompt used by us is research based on this paper - https://arxiv.org/pdf/2307.11760.pdf



This is how the original file looks like - https://pastebin.com/FS6kKFuU

After refactoring by us manually - https://pastebin.com/2Bvw66Ub - This was refactored and was frozen for the purpose of analysis

Refactored using GPT 3.5 - https://pastebin.com/j4fjf5T1. - The entire code was generated by GPT

Here are some differences -

	GPT	Our Code
Naming of Methods	Methods are extracted to handle specific concerns and named accordingly	Names are more focused on the actions they perform rather than the concerns they handle
Error Handling	Wraps potential exceptions in try-catch blocks within appropriate methods extensively	directly handles exceptions within the logic, potentially cluttering the code.
Use of Constants	default timezone ID is encapsulated within the `LocaleUtil` class	directly included as a constant in the filter class.
Handling of Null	Handled efficiently	There is repetition in checking for Null values which was done in order to reduce the conditional sentence depth

Overall, both versions address design smells and improve the readability and maintainability of the original code. GPT's version emphasizes structured error handling, clear separation of concerns, and dependency injection for improved testability. Our version also extracts methods for better readability but may have a slightly less structured approach to error handling and dependency management.

GPT Refactored Version:

Strengths:

- **Readability**: Clear separation of concerns and meaningful method names improve readability.
- **Maintainability**: Dependency injection enhances maintainability by promoting flexibility and easier testing.
- **Adherence to Best Practices**: Adheres closely to best practices such as dependency injection and structured error handling, enhancing maintainability and future scalability.
- **Testing**: More conducive to testing, especially unit testing, due to the use of dependency injection, which facilitates mocking and isolation of dependencies.

Weaknesses:

- **Dependency Injection Overhead**: Introducing dependency injection adds a level of complexity, which might be considered excessive for smaller projects.
- **Initial Setup Overhead**: Setting up dependency injection may require additional configuration and setup time, especially for developers unfamiliar with the concept.

Our Refactored Version:

Strengths:

- **Simplicity**: The approach is straightforward and may be easier to grasp for developers unfamiliar with dependency injection.
- **Direct Approach**: Direct instantiation of dependencies within methods simplifies the implementation, especially for smaller projects with fewer dependencies.

- **Efficiency**: Avoiding dependency injection overhead might lead to slightly better runtime efficiency in certain scenarios.

Weaknesses:

- **Readability**: Method names and organization may be less clear, leading to decreased readability compared to a more structured approach.
- **Maintainability**: Tight coupling between components and lack of dependency injection may hinder maintainability as the project grows or requirements change.
- **Testing**: Testing may be more challenging due to the lack of dependency injection, making it harder to isolate and mock dependencies for unit testing.
- **Scalability**: Lack of dependency injection makes it harder to integrate additional dependencies or modify existing ones, potentially limiting scalability.

While this may seem like GPT 3.5 beats human refactoring, let us not forget that GPT has the capability of generating absolute bullshit with style.

For instance, look at this -

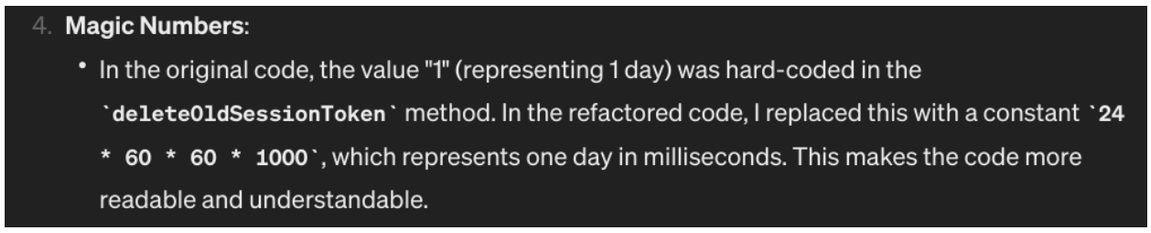
GPT is suggesting to replace this -

```
q.setParameter("minDate", DateTime.now().minusDays(1).toDate());
```

with this -

```
q.setParameter("minDate", new Date(System.currentTimeMillis() - (24 * 60 *
60 * 1000))); // One day ago
```

Understandably, GPT wants to remove the 1 as it might seem hard coded. But the changes it suggested replace something very readable with a very random number which takes the readability completely downhill.



Use your discretion before copying/following GPT blindly. In situations like these manual intervention is possible.

Another possible instance could be GPT flagging for “SQL Injection Vulnerability” while in reality, the code followed one of the best practices to prevent SQL injection

```
public void deleteOldSessionToken(String userId) {
    StringBuilder sb = new StringBuilder("delete from T_AUTHENTICATION_TOKEN AS ato ");
    sb.append(" where ato.AUT_IDUSER_C = :userId and ato.AUT_LONGLASTED_B = :longLasted");
    sb.append(" and ato.AUT_LASTCONNECTIONDATE_D < :minDate ");

    EntityManager em = ThreadLocalContext.get().getEntityManager();
    Query q = em.createNativeQuery(sb.toString());
    q.setParameter(name:"userId", userId);
    q.setParameter(name:"longLasted", value:false);
    q.setParameter(name:"minDate", DateTime.now().minusDays(days:1).toDate());
    q.executeUpdate();
}
```

GPT said, “It's safer to use parameterized queries to avoid this vulnerability.“.But in its refactoring, it ended up doing more or less the existing thing

Bibliography

[1] ChatGPT

[2] Sonarqube

[3] IntellilJ Designite

[4] Class Notes

[5] Refactoring for Software Design Smells Managing Technical Debt by Girish Suryanarayana Ganesh Samarthyam, Tushar Sharma

[6] List of all design smells

[7] CodeMR

[8] PMD

[9] Large Language Models Understand and Can Be Enhanced by Emotional Stimuli

[10] ChatGPT is a bullshit generator. But it can still be amazingly useful

Bonus Task

To be continued .....