

Software Engineering Project 1

Team Numer: 12

Team Members

Name	Roll Number
Bhav Beri	2021111013
Divij	2021101001
Harshit Aggarwal	2021111015
Jhalak Banzal	2021101079
Pranav Agrawal	2021101052

We have used various tools across the project, some of which are as follows. A few examples of the tools outputs are attached in the `docs` folder for reference.

- SonarQube / SonarLint
- Designite
- PMD
- CodeMR
- GPT-3.5-Turbo/4

Contributions to the Project

Below are the approx contrbution areas to the project by each team member. Please note that these are just approx, but the whole team used to sit together and help each other in the different portions of the project.

- UML for the Book Addition & Display Subsystem: Divij
- UML for the User Management Subsystem: Jhalak
- UML for the Bookshelf Management Subsystem: Pranav & Harshit
- Documentation for the SubSystems: Jhalak & Harshit
- Finding Design Smells: Bhav, Harshit & Pranav
- Resolving the minor Code smells: Bhav
- Refactoring the Design Smells
 - Major Part: Bhav & Pranav
 - Other Parts: Jhalak & Divij
- Code Metrics: Divij & Harshit
- Testing/DeBugging of Code: Bhav & Divij
- Merging PRs: Divij
- GPT comparison with the refactored code: Jhalak & Harshit
- Final Documentation: Jhalak & Bhav

Task 1

1. Book Addition & Display Subsystem:

BookResource

- Extends the BaseResource class which has basic stuff for handling HTTP requests, sockets, etc.
- Implements methods to add, remove, and edit books that exist for the user.
- Interacts with several DAO objects to perform the same

PaginatedList

- simple representation of a paginated list, commonly used in applications dealing with large datasets that need to be retrieved in smaller, manageable portions.
- defines a generic class `PaginatedList<T>`, which is used to handle paginated lists of objects of type `T`.
- Used for retrieving large data in chunk or pages. Four instance variables: `limit`, `offset`, `resultCount`, `resultList`.

`limit` - max number of records a page can contain

`offset` - starting point in a list of records for the page

`resultCount` - total number of records in the list

`resultList` - List of results

- Considering that pagination parameters are typically set once upon instantiation and remain constant, making them immutable (e.g., through constructor parameters without corresponding setters) could enhance the class's reliability and maintainability.

PaginatedLists

- Creates and handles `PaginatedList` object
- Provides utilities for working with paginated lists in the context of Java Persistence API (JPA).
- It allows the creation of paginated lists with specified page size and offset, ensuring efficient retrieval of data from large result sets.

BookImportedEvent

- Represents an event raised when a request is made to import books.
- Generates a string of the form `BookImportedEvent{user=value, importFile=value}` for pushing onto the Import bus

UserBookCriteria

- Represents criteria for filtering user book data.
- Stores `userId`, `search query`, `read state`, `tagIdList` for storing the criteria.
- Validation logic for ensuring the correctness of criteria values could be added to setter methods to prevent invalid criteria from being set.

SortCriteria

- Stores the sorting parameter (which column to sort, asc/desc)
- The class lacks mutator methods (setters) for its attributes, which promotes immutability. Once created, an instance of `SortCriteria` cannot be modified, ensuring consistency and predictability.

UserDAO

- Serves as a Data Access Object for managing user-related data in the application's database. It provides methods to interact with the `User` entity, including authentication, creation, retrieval, updating, and deletion of user records.
- **Methods:**
 1. `authenticate(String username, String password)`: Authenticates a user by verifying the provided username and password against the database records. It uses BCrypt for secure password hashing.
 2. `create(User user)`: Creates a new user record in the database. It generates a UUID for the user ID, checks for username uniqueness, hashes the password, sets default values for certain attributes, and persists the user object.
 3. `update(User user)`: Updates an existing user record in the database. It retrieves the user from the database, updates the relevant attributes, and returns the updated user object.
 4. `updatePassword(User user)`: Updates the password of an existing user in the database. It retrieves the user from the database, updates the password with a hashed version, and returns the updated user object.
 5. `findById(String id)`: Retrieves a user record from the database by its unique identifier (`id`).
 6. `getActiveByUsername(String username)`: Retrieves an active user record from the database by its username.
 7. `getActiveByPasswordResetKey(String passwordResetKey)`: Retrieves an active user record from the database by its password reset token.
 8. `delete(String username)`: Deletes a user record from the database by its username. It marks the user as deleted and deletes any associated data, such as authentication tokens.
 9. `hashPassword(String password)`: Hashes the provided password using the BCrypt hashing algorithm for secure storage in the database.
 10. `findAll(PaginatedList<UserDto> paginatedList, SortCriteria sortCriteria)`: Retrieves a paginated list of all user records from the database. It constructs and executes a query to fetch user data, assembles the results into DTOs (Data Transfer Objects), and populates the provided paginated list.
- **Dependencies:**
 - `EntityManager`: Manages the lifecycle of user entities within the persistence context.
 - `ThreadLocalContext`: Provides access to the `EntityManager` associated with the current thread.
 - `BCrypt`: Library for secure password hashing.
 - `Constants`: Contains application-wide constant values.
 - `PaginatedList`, `PaginatedLists`, `QueryParam`, `SortCriteria`: Utility classes for handling pagination, query parameters, and sorting in JPA queries.

UserDto

- Serves as a lightweight representation of user data, facilitating data transfer between different layers of the application. Here's a brief overview:
- Enables efficient transfer of user data between application layers.
- Encapsulates essential user information for specific operations.
- Typically employed for transferring user details to presentation layer components. Enhances maintainability and performance by controlling data granularity.

UserBookDAO

- A Data Access Object (DAO) responsible for interacting with the database to perform CRUD (Create, Read, Update, Delete) operations related to user books.
- Relies on the entity manager injected via the `ThreadLocalContext`, ensuring proper management of database transactions and resources.
- The method `findByCriteria` is quite complex due to the dynamic construction of the query based on multiple criteria. While this offers flexibility, it can make maintenance challenging, especially when modifying or extending the search logic.
- Pagination is implemented using the `PaginatedList` and `PaginatedLists` utility classes, enhancing performance and usability when dealing with large result sets.

BookDAO

- Responsible for CRUD operations related to books in the database.
- The class provides methods for creating a new book and retrieving books by their ID or ISBN number.
- The `create` method persists a new book entity to the database using the entity manager obtained from the thread-local context. It then returns the ID of the newly created book.
- The `getId` method retrieves a book by its ID using the `EntityManager.find` method, which is efficient for primary key lookups.
- The `getByIsbn` method retrieves a book by its ISBN number (either ISBN-10 or ISBN-13) using a JPQL query.
- Relies on the entity manager injected via the `ThreadLocalContext`, ensuring proper management of database transactions and resources.
- Proper error handling is implemented using try-catch blocks to handle potential `NoResultException` when retrieving books by ID or ISBN.

TagDAO

- The class provides methods for creating, reading, updating, and deleting tags.
- The `create` method persists a new tag entity to the database, generating a UUID for the tag ID and setting the creation date.
- Methods like `getId`, `getUserBookId`, `getName`, and `getTagId` retrieve tags based on different criteria such as tag ID, user ID, or tag name.
- The `delete` method marks a tag as deleted by setting the deletion date and also deletes any linked data (user book tag associations) to ensure data integrity.
- The class contains methods like `getUserBookId` that convert raw database query results into DTOs (Data Transfer Objects), making it easier to work with the data in the application layer.
- The `updateTagList` method allows updating the tags associated with a user book. It first deletes the existing tag links and then creates new tag links based on the provided tag IDs.

BookDataService

- Responsible for fetching book information from external APIs like Google Books and Open Library.
- The class uses `RateLimiter` from Guava to limit the rate of API requests made to Google Books and Open Library APIs. This prevents hitting API rate limits and potential service degradation.
- The `searchBook()` method searches for book information using the provided ISBN. It first attempts to search using the Google Books API and falls back to the Open Library API if no results are found.
- API requests are made asynchronously using `ExecutorService` and `FutureTask`. This allows for concurrent execution of API requests, improving performance and responsiveness.
- The class relies on external libraries such as Guava, Jackson, and Joda-Time for functionality related to rate limiting, JSON parsing, and date parsing, respectively.

DirectoryUtil

- Provides utility methods for accessing various directories used by the application.
- The `getBaseDataDirectory()` method returns the base data directory used by the application. It checks if the application is running in a webapp environment and if the `books.home` property is set. If not, it defaults to using a directory named "sismicsbooks" in the webapp root directory or falls back to the system's temporary directory.
- `EnvironmentUtil` is used to detect whether the application is running in a webapp environment and to retrieve the webapp root directory.
- For the theme directory, if the application is not running in a webapp environment, it attempts to load the directory from the classpath.

Book

- Represents the entity model for books in the application's database.
- It defines attributes such as `id`, `title`, `subtitle`, `author`, `description`, `isbn10`, `isbn13`, `pageCount`, `language`, and `publishDate`, representing various properties of a book.

User

- Represents a user entity in the JPA context.
- Contains fields representing user attributes.
- Has storage of user-specific settings, such as language and theme.

Tag

- Represents the entity model for tags in the application's database.
- It defines attributes such as `id`, `name`, `userId`, `createDate`, `deleteDate`, and `color`, representing various properties of a tag.
- Getter and setter methods are provided for each attribute to access and modify the state of the object.

AppContext

- Has various buses (`EventBus`, `asyncEventBus`, `importEventBus`, `bookDataService`, etc.)
- Has Singleton pattern (private constructor)

BookDao

- This Java class BookDao is a Data Access Object (DAO) responsible for handling database operations related to the Book entity in a Java Persistence API (JPA) environment.
- Some of the methods are as follows:
 1. create(Book book): This method creates a new book entry in the database. It persists the provided Book entity using the entity manager obtained from the thread-local context and returns the ID of the newly created book.
 2. getById(String id): This method retrieves a book from the database by its ID. It uses the entity manager to find the Book entity with the specified ID and returns it. If no book is found, it returns null.
 3. getByIsbn(String isbn): This method retrieves a book from the database by its ISBN number (either ISBN-10 or ISBN-13). It constructs a JPQL query to select the book with the specified ISBN number and executes it. If a book matching the ISBN is found, it returns it; otherwise, it returns null.

UserBook

- This class represents the association between a user and a book within the application's database.
- It stores metadata about the relationship, such as the creation date, deletion date, and read date.
- The main fields are id, bookId, userId, createDate, deleteDate, readDate.
- The hashCode() and equals() methods are overridden to provide custom equality comparison based on the bookId and userId fields. This ensures proper comparison and identification of UserBook instances.

BaseResource (abstract class)

- Implements authentication and other checks for the user so that he is not anonymous
- Can be extended by resource classes, supplying common functionalities and applying security measures
- Implements checks to verify if the user has the authority to perform tasks

UserBookTag

- This class represents a mapping between a user book and a tag within a database. It allows for the association of multiple tags with a user book and provides methods for accessing and manipulating these associations within a JPA-based application.
- The main fields are id, userBookId and tagId.
- The hashCode() and equals() methods are overridden to provide custom equality comparison based on the userBookId and tagId fields. This ensures proper comparison and identification of UserBookTag instances.

2. Bookshelf Management Subsystem:

BaseResource (abstract class)

- Implements authentication and other checks for the user so that he is not anonymous

- Can be extended by resource classes, supplying common functionalities and applying security measures
- Implements checks to verify if the user has the authority to perform tasks

Tag

- Represents tags in the system, storing information such as their name, associated user ID, creation date, deletion date (if applicable), and color.
- Ensures proper mapping of the class to the database schema, facilitating persistence operations.
- Attributes are encapsulated with private access modifiers and accessed through public getter and setter methods, adhering to the principle of encapsulation.
- The class does not include any validation logic for attributes like `id`, `name`, and `color`. Adding validation logic to ensure that these attributes meet certain criteria (e.g., length constraints) can enhance data integrity.

TagResource

- Provides endpoints for listing, adding, updating, and deleting tags.
- Implements standard RESTful endpoints for CRUD operations on tags, providing a clear and predictable interface for clients.
- Properly handles authentication to ensure only authenticated users can perform operations on tags.
- Utilizes data validation to ensure the integrity of tag attributes.
- Direct interaction with `TagDao` within the resource methods tightly couples the resource layer with the data access layer, potentially leading to issues like code duplication and difficulty in testing.
- **Code Duplication:** Some validation and error-checking logic is duplicated across methods. Refactoring common functionality into helper methods or aspects could improve maintainability and readability.

TagDao

- Responsible for interacting with the database to perform CRUD (Create, Read, Update, Delete) operations on tags.
- Utilizes parameterized queries to prevent SQL injection.
- Supports transactional operations for data consistency.
- Lack of input validation in some methods, potentially leading to unexpected behavior.
- Direct interaction with the database layer might result in tight coupling with the persistence framework.

TagDto

- Serves as a Data Transfer Object (DTO) for transferring tag information between different layers of the application.
- Encapsulates tag attributes within a single object, improving maintainability and readability.
- Lack of validation or data transformation logic, which might lead to inconsistencies if not handled properly when converting between DTOs and domain objects.

UserBookDao

- Provides data access methods for managing user books

- **Search:** A method for searching user books by criteria such as search text, tag IDs, and read status.
- **Pagination:** Supports paginated retrieval of user books to efficiently handle large datasets.
- Uses parameterized queries to prevent SQL injection attacks and ensure data integrity. Breaks down complex queries into manageable parts, enhancing readability and maintainability.
- **Lack of Abstraction:** Directly couples database-specific details with application logic, potentially reducing portability and making it harder to switch database providers.
- **Limited Reusability:** Methods are tightly coupled with the `UserBookDto` class, limiting their reusability in other parts of the application.

UserBookDto

- Serves as a data transfer object (DTO) representing user book information. It contains properties corresponding to various attributes of a user book, such as title, subtitle, author, language, publication date, creation date, and read date.
- Provides a straightforward representation of user book data with simple getter and setter methods for each attribute.
- Does not include validation logic for ensuring the correctness of data during setting of properties, potentially leading to inconsistencies or errors in the application.
- Uses `Long` data type for timestamps instead of `Date` or `LocalDateTime`, which may require additional conversion and handling in certain scenarios.

UserBookCriteria

- Represents criteria used for searching user books. It encapsulates various filtering options such as user ID, search query, read state, and tag IDs.
- Offers a simple interface with getter and setter methods for each criteria property, making it easy to understand and use.
- The absence of default values for criteria properties may require additional handling to prevent null pointer exceptions or unexpected behavior when certain criteria are not set.

BookResource

- This class acts as an intermediary between the Android application and a backend server that provides data about books. It encapsulates the logic for making HTTP requests to the server's API endpoints and handling the responses.
- The `list()` method retrieves a list of books from the server, specifying parameters for sorting and pagination.
- The `info()` method fetches detailed information about a specific book identified by its unique ID.
- The `add()` method allows the application to add a new book to the system by providing its ISBN.

UserBookTag

- This class represents a mapping between a user book and a tag within a database. It allows for the association of multiple tags with a user book and provides methods for accessing and manipulating these associations within a JPA-based application.
- The main fields are `id`, `userBookId` and `tagId`.
- The `hashCode()` and `equals()` methods are overridden to provide custom equality comparison based on the `userBookId` and `tagId` fields. This ensures proper comparison and identification of

UserBookTag instances.

BookImportedEvent

- This class serves as a container for data related to a book import event, allowing for the transfer of information about the import request within the application's event-driven architecture.
- The main fields are the user and the importFile which contains the csv file to be imported.
- This event class encapsulates information about a book import request, including the user initiating the request and the file containing the books to be imported.
- It facilitates passing relevant data about the import request between different components of the application.

UserBook

- This class represents the association between a user and a book within the application's database.
- It stores metadata about the relationship, such as the creation date, deletion date, and read date.
- The main fields are id, bookId, userId, createDate, deleteDate, readDate.
- The hashCode() and equals() methods are overridden to provide custom equality comparison based on the bookId and userId fields. This ensures proper comparison and identification of UserBook instances.

BookImportAsyncListener

- This Java class is a listener component responsible for handling events related to book import requests asynchronously.
- `@Subscribe` indicates that the method `on(BookImportedEvent bookImportedEvent)` subscribes to events of type `BookImportedEvent` published by the event bus.
- `on(BookImportedEvent bookImportedEvent)`: This method is invoked when a `BookImportedEvent` is published. It processes the event by importing books from a CSV file provided in the event.
- This adds the books to the database if they don't already exist and then adds the tags to the user corresponding to each book.

Book

- Represents the entity model for books in the application's database.
- It defines attributes such as `id`, `title`, `subtitle`, `author`, `description`, `isbn10`, `isbn13`, `pageCount`, `language`, and `publishDate`, representing various properties of a book.

PaginatedList

- Simple representation of a paginated list, commonly used in applications dealing with large datasets that need to be retrieved in smaller, manageable portions.
- Defines a generic class `PaginatedList<T>`, which is used to handle paginated lists of objects of type `T`.
- Used for retrieving large data in chunk or pages. Four instance variables: `limit`, `offset`, `resultCount`, `resultList`.
- Considering that pagination parameters are typically set once upon instantiation and remain constant, making them immutable (e.g., through constructor parameters without corresponding

setters) could enhance the class's reliability and maintainability.

PaginatedLists

- Creates and handles PaginatedList object
 - Provides utilities for working with paginated lists in the context of Java Persistence API (JPA).
 - It allows the creation of paginated lists with specified page size and offset, ensuring efficient retrieval of data from large result sets.
-

3. User Management Subsystem:

BaseFunction

- Defines the fundamental functionalities that can be performed in the associated base function
- Defines a role (eg: admin) and the corresponding base function
- Controls access based on the role of a user

BaseResource (abstract class)

- Implements authentication and other checks
- Can be extended by resource classes, supplying common functionalities and applying security measures

RoleBaseFunctionDao

- Data Access Object for base functions that have a role associated with them

Constants

- Has constants that are used all over the application, increasing efficiency and eliminating the need to hardcode them
- Stores the default locale, timezone, theme, admin password, and user role

AuthenticationTokenDao

- Data Access Object for Authentication Tokens
- Has methods to get, retrieve, create, and delete tokens

TokenBasedSecurityFilter

- Manages user authentication via authentication tokens in cookies
- Checks token validity and expiration status to determine user authentication status.
- Injects either authenticated or anonymous user principal into request attributes
- Utilizes DAO classes (AuthenticationTokenDao, UserDao, RoleBaseFunctionDao) for database interactions

ThreadLocalContext

- Manages context associated with a user request and stores it in a ThreadLocal.

- Utilizes a ThreadLocal variable named threadLocalContext for storage.
- Includes an EntityManager field for managing database transactions.
- Follows a singleton pattern to ensure one instance per thread.
- Provides methods to initialize and clean up the thread context (get() and cleanup()).
- Offers a isInTransactionalContext() method to detect if the entity manager is defined, indicating a transactional scope.

ValidationUtil

- A utility class for validating parameters in a RESTful context.
- Includes methods for checking if parameters are required and validating their lengths.
- Provides validation for strings, including not blank, hex color, email, HTTP URL, and alphanumeric patterns.
- Validates and parses dates.
- Validates locale and theme IDs against database entries.
- Throws `ClientException` with appropriate error messages for validation failures.
- Utilizes DAO classes (`ThemeDao`, `LocaleDao`) to validate theme and locale IDs against database entries.

SortCriteria

- Represents sort criteria for a query.
- Includes an index of the column to sort (starting from 0) and a flag indicating ascending or descending order.
- Provides a constructor to initialize the sort criteria with optional parameters for column index and sort order.
- Contains getters to retrieve the column index and sort order.

PaginatedList

- Represents a paginated list of records.
- Includes properties for page size, offset, total result count, and a list of records for the current page.
- Provides constructors to initialize page size and offset.
- Contains getters and setters for accessing and modifying the properties of the paginated list.

PaginatedLists

- Provides utilities for handling paginated lists in JPA contexts.
- Can create paginated lists with configurable page size and offset.
- Supports executing count queries to determine the total number of results.
- Executes queries to retrieve data for the current page based on pagination parameters.
- Offers methods for executing paginated queries with sorting criteria.
- Utilizes `QueryUtil` and `QueryParam` classes for query execution and parameter management.
- Implements default page size and maximum page size constants for consistency and limiting purposes.

User

- Represents a user entity in the JPA context.

- Contains fields representing user attributes.
- Has storage of user-specific settings, such as language and theme.

UserDao

- Serves as a Data Access Object for managing user-related data in the application's database. It provides methods to interact with the `User` entity, including authentication, creation, retrieval, updating, and deletion of user records.
- **Methods:**
 1. `authenticate(String username, String password)`: Authenticates a user by verifying the provided username and password against the database records. It uses BCrypt for secure password hashing.
 2. `create(User user)`: Creates a new user record in the database. It generates a UUID for the user ID, checks for username uniqueness, hashes the password, sets default values for certain attributes, and persists the user object.
 3. `update(User user)`: Updates an existing user record in the database. It retrieves the user from the database, updates the relevant attributes, and returns the updated user object.
 4. `updatePassword(User user)`: Updates the password of an existing user in the database. It retrieves the user from the database, updates the password with a hashed version, and returns the updated user object.
 5. `findById(String id)`: Retrieves a user record from the database by its unique identifier (`id`).
 6. `getActiveByUsername(String username)`: Retrieves an active user record from the database by its username.
 7. `getActiveByPasswordResetKey(String passwordResetKey)`: Retrieves an active user record from the database by its password reset token.
 8. `delete(String username)`: Deletes a user record from the database by its username. It marks the user as deleted and deletes any associated data, such as authentication tokens.
 9. `hashPassword(String password)`: Hashes the provided password using the BCrypt hashing algorithm for secure storage in the database.
 10. `findAll(PaginatedList<UserDto> paginatedList, SortCriteria sortCriteria)`: Retrieves a paginated list of all user records from the database. It constructs and executes a query to fetch user data, assembles the results into DTOs (Data Transfer Objects), and populates the provided paginated list.
- **Dependencies:**
 - `EntityManager`: Manages the lifecycle of user entities within the persistence context.
 - `ThreadLocalContext`: Provides access to the `EntityManager` associated with the current thread.
 - `BCrypt`: Library for secure password hashing.

- **Constants**: Contains application-wide constant values.
- **PaginatedList, PaginatedLists, QueryParam, SortCriteria**: Utility classes for handling pagination, query parameters, and sorting in JPA queries.

Overall, the **UserDao** class encapsulates the logic for interacting with user data in the database, providing essential functionalities for user management within the application.

UserDto

- Serves as a lightweight representation of user data, facilitating data transfer between different layers of the application. Here's a brief overview:
- Enables efficient transfer of user data between application layers.
- Encapsulates essential user information for specific operations.
- Typically employed for transferring user details to presentation layer components. Enhances maintainability and performance by controlling data granularity.

UserResource

- Extends **BaseResource** and gives resource implementation for managing user-related operations such as registration, login, logout, profile updates, and user deletion.
- It includes functionality for managing user sessions, including login, logout, and session deletion.
- Input data is validated before processing, ensuring data integrity and preventing common security vulnerabilities like injection attacks.
- Some code segments, particularly validation logic, appear to be duplicated across different methods. Refactoring common logic into utility methods or shared components can improve maintainability.
- The class may benefit from further modularization, breaking down complex methods into smaller, more focused functions to improve readability and maintainability.

Iprincipal

- The **IPrincipal** interface defines the contract for principal objects in the application. It extends the **java.security.Principal** interface and provides additional methods to retrieve user-related information such as ID, locale, timezone, and email.
- Tight coupling with **DateTimeZone** from Joda-Time library, which may limit flexibility in using other date/time libraries.

AnonymousPrincipal

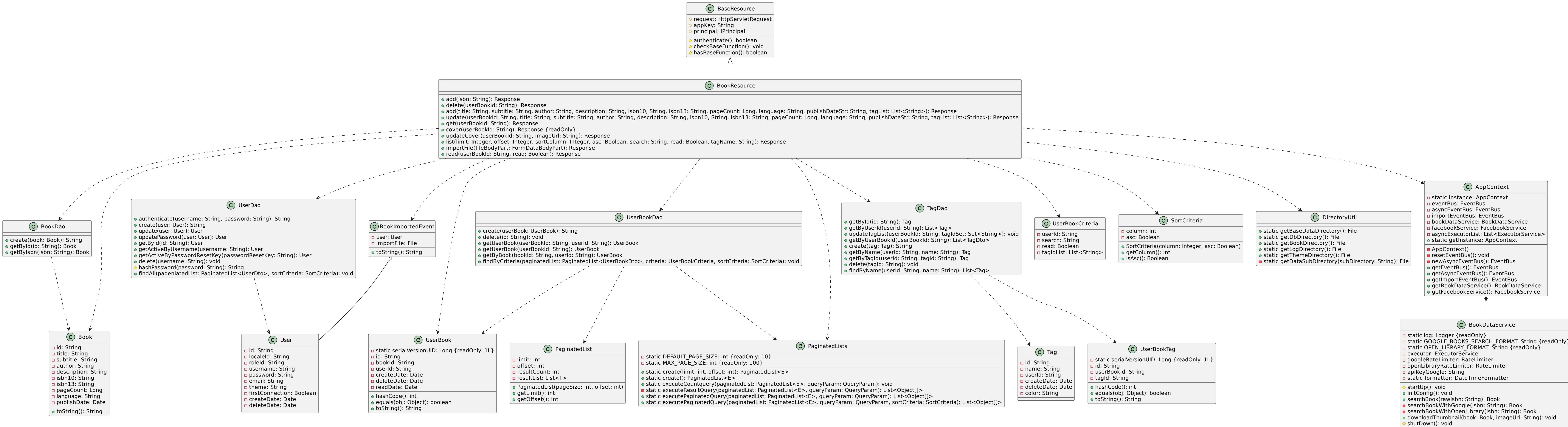
- The **AnonymousPrincipal** class represents an anonymous user within the application.
- It implements the **IPrincipal** interface and provides methods to retrieve information about the anonymous user such as locale and timezone.
- This class is typically used to represent unauthenticated users or users with limited access.
- Lack of flexibility in representing different types of anonymous users with varying characteristics.

UserPrincipal

- The **UserPrincipal** class represents an authenticated user within the application.
- It implements the **IPrincipal** interface and provides methods to retrieve information about the authenticated user such as ID, username, locale, timezone, email, and base functions.

- This class is typically used to represent users who have successfully authenticated and have access to various functionalities within the application.
- Provides a comprehensive representation of authenticated users within the application, including essential user information and base functions.

The UML Diagrams are attached in the same order as the above subsystems.



Bookshelf





Task 2A

There are various design smells listed on the [GitHub issues page](#). The exact information about the same can be found in the respective issues themselves.

Listing out some of the major names of the design smells, as per the issues and found in the codebase:

1. Missing Modularization
2. Brain Method
3. God Class
4. Missing Hierarchy
5. Missing Abstraction
6. Missing Encapsulation
7. Broken hierarchy
8. Duplicated Abstraction

Task 2B

Written along with the task 3B.

Task 3

Task 3A

All of the Design Smells are listed in the [GitHub issues page](#). The exact information about the same can be found in the respective issues themselves.

The refactoring PRs are also listed in the [GitHub PRs page](#). All of the PRs are linked with at least one issue, and marked with the corresponding label for the design smell title.

Task 3B (Along with Task 2B)

Task 2b & 3b

Metrics

Type filter text																			
Element	LOC	Coupling	Complexity	Lack of Cohesion	RFC	WMC	LOC	CMLOC	NOF	NOSF	NOM	NOSM	NORM	LCOM	LCAM	LTCC	ATFD	SI	
> AuthenticationException	4	low	low-medium	low	1	1	4	2	0	1	1	0	0	0.0	0.0	0.0	0	0.0	
> PermissionException	4	low	low-medium	low	1	1	4	2	0	1	1	0	0	0.0	0.0	0.0	0	0.0	
> com.sismics.books.core.util	105	low	low	low		37	105												
> ConfigUtil	15	low	low	low	14	5	15	14	0	0	0	4	0	0.0	0.125	0.0	1	0.0	
> DirectoryUtil	37	low	low	low	15	15	37	36	0	0	0	6	0	0.0	0.417	0.0	1	0.0	
> EntityManagerUtil	3	low	low	low	6	1	3	2	0	0	0	1	0	0.0	0.0	0.0	1	0.0	
> StreamUtil	10	low	low	low	3	3	10	9	0	0	0	1	0	0.0	0.0	0.0	0	0.0	
> TransactionUtil	40	low	low	low	19	13	40	38	0	1	0	2	0	0.0	0.25	1.0	1	0.0	
> com.sismics.books.core.util.jpa	87	low	low	low		27	87												
> PaginatedList	20	low	low	low	7	7	20	15	4	0	7	0	0	0.833	0.524	0.81	0	0.0	
> PaginatedLists	38	low	low	low	31	10	38	35	0	2	0	6	0	0.0	0.467	1.0	3	0.0	
> QueryParam	10	low	low	low	3	3	10	7	2	0	3	0	0	0.5	0.444	0.333	0	0.0	
> QueryUtil	7	low	low	low	12	2	7	6	0	0	0	1	0	0.0	0.0	0.0	2	0.0	
> SortCriteria	12	low	low	low	3	5	12	9	2	0	3	0	0	0.5	0.444	0.333	0	0.0	
> com.sismics.books.core.util.math	19	low	low	low		7	19												
> MathUtil	19	low	low	low	9	7	19	18	0	0	0	3	0	0.0	0.167	0.0	0	0.0	
> com.sismics.books.core.util.mime	32	low	low	low		24	32												
> MimeType	7	low	low	low	0	0	7	0	0	6	0	0	0	0.0	0.0	0.0	0	0.0	
> MimeTypeUtil	25	low	low-medium	low	6	24	25	24	0	0	0	2	0	0.0	0.333	0.0	0	0.0	
> com.sismics.util	133	low	low	low		43	133												
> EnvironmentUtil	26	low	low	low	11	9	26	19	0	6	0	9	0	0.917	0.444	0.833	0	0.0	
> HttpUtil	46	low	low	low	15	12	46	44	0	1	0	2	0	0.0	0.167	1.0	0	0.0	
> LocaleUtil	26	low	low	low	13	8	26	25	0	0	0	2	0	0.0	0.0	0.0	1	0.0	
> ResourceUtil	35	low	low	low	29	14	35	34	0	0	0	2	0	0.0	0.125	0.0	0	0.0	
> com.sismics.util.context	18	low	low	low		7	18												
> ThreadLocalContext	18	low	low	low	8	7	18	15	1	1	4	2	0	0.625	0.417	0.6	0	0.0	
> com.sismics.util.jpa	233	low	low	low		60	233												
> DbOpenHelper	110	medium-high	low-medium	low-medium	74	31	110	102	6	1	9	0	0	0.786	0.667	0.857	0	0.0	
> EMF	51	low-medium	low	low	21	5	51	29	0	2	1	2	0	1.0	0.0	0.0	0	0.0	
> ManagedProviderConnectionI	41	low-medium	low	low	19	11	41	37	3	0	6	1	0	0.583	0.524	0.667	0	0.0	
> SessionUtil	5	low	low	low	5	2	5	4	0	0	1	1	0	0.0	0.0	0.0	0	0.0	
> SuppliedConnectionProviderC	26	low	low	low	10	8	26	22	3	0	4	0	0	0.444	0.5	0.167	0	0.0	
> com.sismics.util.log4j	86	low	low	low		31	86												
> MemoryAppender	52	low-medium	low-medium	low-medium	39	20	52	49	2	0	7	0	0	0.625	0.657	0.667	3	0.0	
> LogCriteria	16	low	low	low	7	6	16	12	3	0	6	0	0	0.8	0.25	0.8	0	0.0	
> LogEntry	18	low	low	low	5	5	18	13	4	0	5	0	0	0.75	0.533	0.6	0	0.0	

Overview Metric Distribution Package Structure Sunburst Package Dependency Treemap Project Outline

Type filter text																			
Element	LOC	Coupling	Complexity	Lack of Cohesion	RFC	WMC	LOC	CMLOC	NOF	NOSF	NOM	NOSM	NORM	LCOM	LCAM	LTCC	ATFD	SI	
> AuthenticationException	4	low	low-medium	low	1	1	4	2	0	1	1	0	0	0.0	0.0	0.0	0	0.0	
> PermissionException	4	low	low-medium	low	1	1	4	2	0	1	1	0	0	0.0	0.0	0.0	0	0.0	
> com.sismics.books.core.util	105	low	low	low		37	105												
> ConfigUtil	15	low	low	low	14	5	15	14	0	0	0	4	0	0.0	0.125	0.0	1	0.0	
> DirectoryUtil	37	low	low	low	15	15	37	36	0	0	0	6	0	0.0	0.417	0.0	1	0.0	
> EntityManagerUtil	3	low	low	low	6	1	3	2	0	0	0	1	0	0.0	0.0	0.0	1	0.0	
> StreamUtil	10	low	low	low	3	3	10	9	0	0	0	1	0	0.0	0.0	0.0	0	0.0	
> TransactionUtil	40	low	low	low	19	13	40	38	0	1	0	2	0	0.0	0.25	1.0	1	0.0	
> com.sismics.books.core.util.jpa	87	low	low	low		27	87												
> PaginatedList	20	low	low	low	7	7	20	15	4	0	7	0	0	0.833	0.524	0.81	0	0.0	
> PaginatedLists	38	low	low	low	31	10	38	35	0	2	0	6	0	0.0	0.467	1.0	3	0.0	
> QueryParam	10	low	low	low	3	3	10	7	2	0	3	0	0	0.5	0.444	0.333	0	0.0	
> QueryUtil	7	low	low	low	12	2	7	6	0	0	0	1	0	0.0	0.0	0.0	2	0.0	
> SortCriteria	12	low	low	low	3	5	12	9	2	0	3	0	0	0.5	0.444	0.333	0	0.0	
> com.sismics.books.core.util.math	19	low	low	low		7	19												
> MathUtil	19	low	low	low	9	7	19	18	0	0	0	3	0	0.0	0.167	0.0	0	0.0	
> com.sismics.books.core.util.mime	32	low	low	low		24	32												
> MimeType	7	low	low	low	0	0	7	0	0	6	0	0	0	0.0	0.0	0.0	0	0.0	
> MimeTypeUtil	25	low	low-medium	low	6	24	25	24	0	0	0	2	0	0.0	0.333	0.0	0	0.0	
> com.sismics.util	133	low	low	low		43	133												
> EnvironmentUtil	26	low	low	low	11	9	26	19	0	6	0	9	0	0.917	0.444	0.833	0	0.0	
> HttpUtil	46	low	low	low	15	12	46	44	0	1	0	2	0	0.0	0.167	1.0	0	0.0	
> LocaleUtil	26	low	low	low	13	8	26	25	0	0	0	2	0	0.0	0.0	0.0	1	0.0	
> ResourceUtil	35	low	low	low	29	14	35	34	0	0	0	2	0	0.0	0.125	0.0	0	0.0	
> com.sismics.util.context	18	low	low	low		7	18												
> ThreadLocalContext	18	low	low	low	8	7	18	15	1	1	4	2	0	0.625	0.417	0.6	0	0.0	
> com.sismics.util.jpa	233	low	low	low		60	233												
> DbOpenHelper	110	medium-high	low-medium	low-medium	74	31	110	102	6	1	9	0	0	0.786	0.667	0.857	0	0.0	
> EMF	51	low-medium	low	low	21	5	51	29	0	2	1	2	0	1.0	0.0	0.0	0	0.0	
> ManagedProviderConnectionI	41	low-medium	low	low	19	11	41	37	3	0	6	1	0	0.583	0.524	0.667	0	0.0	
> SessionUtil	5	low	low	low	5	2	5	4	0	0	1	1	0	0.0	0.0	0.0	0	0.0	
> SuppliedConnectionProviderC	26	low	low	low	10	8	26	22	3	0	4	0	0	0.444	0.5	0.167	0	0.0	
> com.sismics.util.log4j	86	low	low	low		31	86												
> MemoryAppender	52	low-medium	low-medium	low-medium	39	20	52	49	2	0	7	0	0	0.625	0.657	0.667	3	0.0	
> LogCriteria	16	low	low	low	7	6	16	12	3	0	6	0	0	0.8	0.25	0.8	0	0.0	
> LogEntry	18	low	low	low	5	5	18	13	4	0	5	0	0	0.75	0.533	0.6	0	0.0	

Overview Metric Distribution Package Structure Sunburst Package Dependency Treemap Project Outline

Element	LOC	Coupling	Complexity	Lack of Cohesion	RFC	WMC	LOC	CMLOC	NOF	NOSF	NOM	NOSM	NORM	LCOM	LCAM	LTCC	ATFD	SI
com.sismics.books.core.dao.jpac	37	low	low	low		14	37											
UserBookCriteria	21	low	low	low-medium	8	8	21	16	4	0	8	0	0	0.857	0.525	0.857	0	0.0
UserContactCriteria	16	low	low	low	6	6	16	12	3	0	6	0	0	0.8	0.25	0.8	0	0.0
com.sismics.books.core.dao.jpac	132	low	low	low		50	132											
TagDto	17	low	low	low	6	6	17	12	3	0	6	0	0	0.8	0.25	0.8	0	0.0
UserAppDto	31	low	low	low	12	12	31	24	6	0	12	0	0	0.909	0.5	0.909	0	0.0
UserBookDto	42	low	low	low	16	16	42	32	8	0	16	0	0	0.933	0.5	0.933	0	0.0
UserContactDto	16	low	low	low	6	6	16	12	3	0	6	0	0	0.8	0.25	0.8	0	0.0
UserDto	26	low	low	low	10	10	26	20	5	0	10	0	0	0.889	0.5	0.889	0	0.0
com.sismics.books.core.event	28	low	low	low		8	28											
BookImportedEvent	17	low	low	low	8	5	17	14	2	0	5	0	0	0.5	0.533	0.4	0	0.0
UserAppCreatedEvent	11	low	low	low	6	3	11	9	1	0	3	0	0	0.0	0.333	0.0	0	0.0
com.sismics.books.core.listener	90	low	low	low			90											
BookImportAsyncListener	68	low	low	low	26	2	68	66	0	1	1	0	0	0.0	0.0	0.0	0	0.0
UserAppCreatedAsyncListener	22	low	low	low	26	2	22	20	0	1	1	0	0	0.0	0.0	0.0	1	0.0
com.sismics.books.core.listener	5	low	low	low		1	5											
DeadEventListener	5	low	low	low	3	1	5	3	0	1	1	0	0	0.0	0.0	0.0	0	0.0
com.sismics.books.core.model	45	low	low	low		11	45											
AppContext	45	low-medium	low	low	15	11	45	37	6	1	8	1	0	0.893	0.0	1.0	0	0.0
com.sismics.books.core.service	8	low	low	low		2	8											
AuthenticationException	4	low	low-medium	low	1	1	4	2	0	1	1	0	0	0.0	0.0	0.0	0	0.0
PermissionException	4	low	low-medium	low	1	1	4	2	0	1	1	0	0	0.0	0.0	0.0	0	0.0
com.sismics.books.core.util	105	low	low	low		37	105											
ConfigUtil	15	low	low	low	14	5	15	14	0	0	0	4	0	0.0	0.125	0.0	1	0.0
DirectoryUtil	37	low	low	low	15	15	37	36	0	0	0	6	0	0.0	0.417	0.0	1	0.0
EntityManagerUtil	3	low	low	low	6	1	3	2	0	0	0	1	0	0.0	0.0	0.0	1	0.0
StreamUtil	10	low	low	low	3	3	10	9	0	0	0	1	0	0.0	0.0	0.0	0	0.0
TransactionUtil	40	low	low	low	19	13	40	38	0	1	0	2	0	0.0	0.25	1.0	1	0.0
com.sismics.books.core.util	87	low	low	low		27	87											
PaginatedList	20	low	low	low	7	7	20	15	4	0	7	0	0	0.833	0.524	0.81	0	0.0
PaginatedLists	38	low	low	low	31	10	38	35	0	2	0	6	0	0.0	0.467	1.0	3	0.0
QueryParam	10	low	low	low	3	3	10	7	2	0	3	0	0	0.5	0.444	0.333	0	0.0
QueryUtil	7	low	low	low	12	2	7	6	0	0	1	0	0	0.0	0.0	0.0	2	0.0
SortCriteria	12	low	low	low	3	5	12	9	2	0	3	0	0	0.5	0.444	0.333	0	0.0
com.sismics.books.core.util.math	19	low	low	low		7	19											
MathUtil	19	low	low	low	9	7	19	18	0	0	0	3	0	0.0	0.167	0.0	0	0.0

Element	LOC	Coupling	Complexity	Lack of Cohesion	RFC	WMC	NOSM	NORM	LCOM	LCAM	LTCC	ATFD	SI
books-web													
com.sismics.books.rest.constant	2	low	low	low		0							
BaseFunction	2	low	low-medium	low	0	0	0	0	0.0	0.0	0.0	0	0.0
com.sismics.books.rest.resource	1020	low-medium	low-medium	low		202							
AppResource	66	medium-high	low-medium	low	46	9	0	0	0.0	0.333	0.0	0	0.0
BaseResource	21	low	low	low	8	8	0	0	1.0	0.167	0.0	0	0.0
BookResource	366	high	medium-high	low-medium	120	79	0	0	0.0	0.662	0.0	0	0.0
ConnectResource	131	high	low-medium	low	56	19	0	0	0.0	0.458	0.0	0	0.0
LocaleResource	15	low	low-medium	low	9	2	0	0	0.0	0.0	0.0	0	0.0
TagResource	85	low-medium	low-medium	low	31	18	0	0	0.0	0.125	0.0	0	0.0
TextPlainMessageBodyWriter	24	low	low	low	6	4	0	0	0.0	0.208	0.0	0	0.0
ThemeResource	15	low	low-medium	low	8	2	0	0	0.0	0.0	0.0	0	0.0
UserResource	297	high	medium-high	low-medium	90	61	0	0	0.0	0.631	0.0	0	0.0

Code Metrics

We used Codemr for code analysis. It provided a variety of code metrics, but we specifically focused on six: Lines of Code (LOC), coupling, complexity, cohesion, Response from Class (RFC), and Weighted Method Count (WMC).

Before refactoring

1. Lines of Code: The bookresource and userresource files contain a high number of lines of code, which can impact maintainability. This could be an

early indication that these files have too many responsibilities, making them prime candidates for refactoring.

2. Coupling: The resource classes, particularly BookResource and UserResource, exhibit a high degree of coupling. This is somewhat expected as they manage HTTP requests and relay them to other systems. Similarly, the BookDataService and FacebookService classes are highly coupled, suggesting that these might be god classes. Consequently, modifications in these classes could lead to unforeseen consequences and potentially introduce bugs into a significant portion of the codebase. This could negatively impact maintainability and needs to be addressed.
3. Complexity: The BookResource and UserResource classes have high complexity, indicating that their functionality is quite complicated. Combined with the other issues present in these classes, this complexity poses a significant challenge to their understandability and maintainability.
4. Cohesion: We determined that the codebase has high cohesion, suggesting that there are no unnecessary variables declared and all declared attributes are used appropriately by their respective classes.
5. Response from Class (RFC): RFC scores are high for both the resource classes and the service classes. This is anticipated as these are the key classes handling a substantial part of the project's functionality. However, this also implies that tracking down and fixing any bugs would be challenging.
6. Weighted Method Count (WMC): The resource classes tend to have a higher WMC than other classes, as do the service classes, though to a lesser extent. This supports our hypothesis that these classes contain complex functionality, making them difficult to understand and maintain.

We've identified several problematic classes within the code base. These classes handle multiple functions, resulting in high complexity. Additionally, numerous methods call upon these classes, making them bug-prone and hard to read.

Analysis of books-core

General Information

Total lines of code: 2523

Number of classes: 71

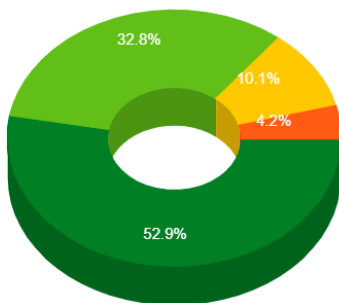
Number of packages: 20

Number of external packages: 48

Number of external classes: 210

Number of problematic classes: 1

Number of highly problematic classes: 0



Analysis of books-web

General Information

Total lines of code: 1022

Number of classes: 10

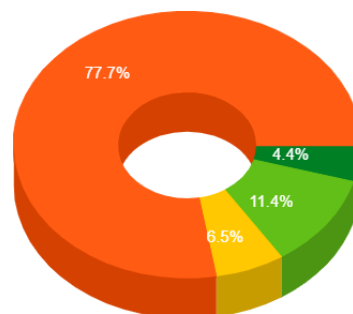
Number of packages: 2

Number of external packages: 43

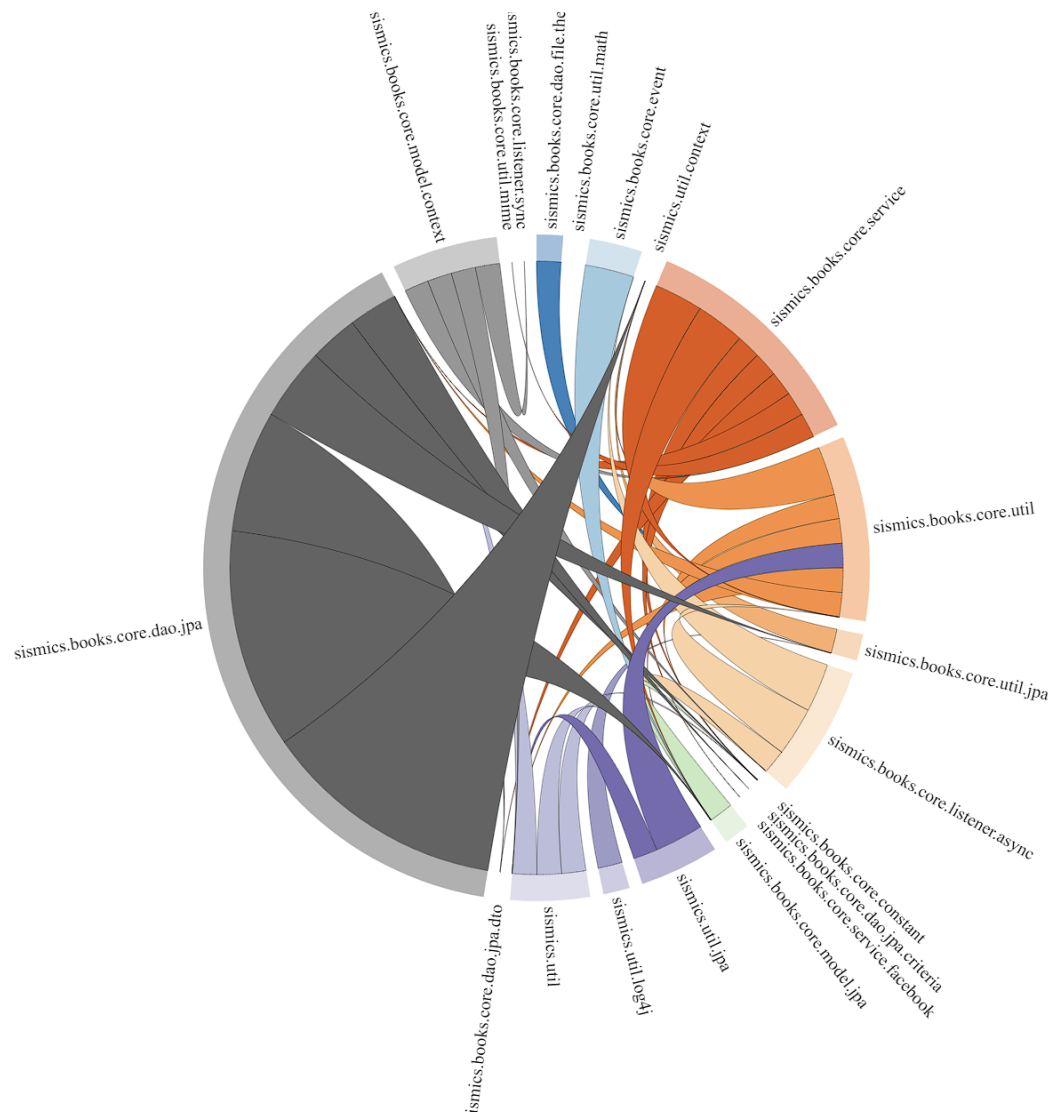
Number of external classes: 128

Number of problematic classes: 3

Number of highly problematic classes: 0



Package Dependencies:



After Refactoring

After manually refactoring the code, we observed the following changes in the metrics:

1. Lines of code: The number of lines in almost all files was reduced, with none exceeding 130 lines in any single file.
2. Coupling: Most of the problematic classes now have low to medium coupling, except for the Facebook service. This improvement makes the code more modular and maintainable. However, some of the helper classes for the newly introduced resource classes have relatively high coupling. This suggests that the functionality of some methods is inherently coupled to the rest of the system, a problem that might be solved by a major architectural redesign.

3. Complexity: The complexity of the resource classes remained unchanged, but it was reduced for the BookData and Facebook service classes. However, the classes to which complexity was abstracted still have high complexity levels. Their other metrics are low, so this high complexity should not produce other issues.
4. Cohesion: High cohesion was maintained throughout the code base.
5. Response from class (RFC): With the introduction of significant abstraction, it was expected that RFC would increase for most classes, although the increase was not significant. However, it decreased for the BookDataService, as we refactored it to call methods based on certain preconditions, thus reducing the number of potentially called methods.
6. Weighted Method Count (WMC): The WMC has drastically decreased for all classes. This is a positive development as it signifies a significant reduction in the complexity of each method, thereby improving the readability and maintainability of the functions.

After refactoring, we found that the overall complexity of the codebase has not significantly decreased. This implies that a substantial architectural redesign of the system is necessary, and it cannot be simply addressed by removing code and design smells. However, we did notice a reduction in coupling and complexity in problem areas. These issues have either been largely eliminated or shifted to classes with singular responsibilities, specifically designed as helper classes. This enhances readability and makes the code easier to maintain.

Analysis of books-core_new

General Information

Total lines of code: 2575

Number of classes: 74

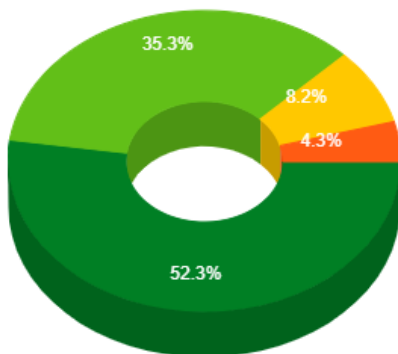
Number of packages: 20

Number of external packages: 48

Number of external classes: 210

Number of problematic classes: 1

Number of highly problematic classes: 0



Analysis of books-web_new

General Information

Total lines of code: 1108

Number of classes: 20

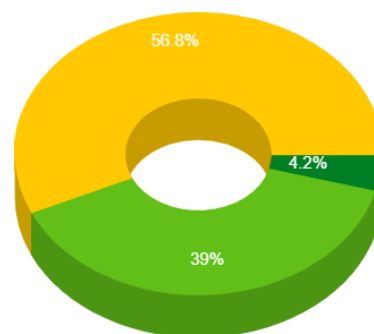
Number of packages: 3

Number of external packages: 43

Number of external classes: 127

Number of problematic classes: 0

Number of highly problematic classes: 0



Books Web:

Detailed metric tables

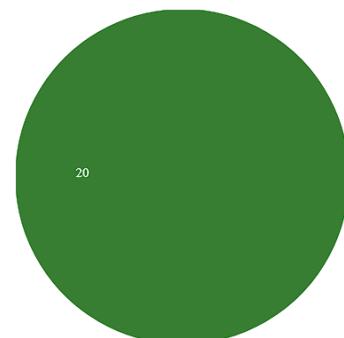
Classes with high coupling, high complexity, low cohesion (#0)

Classes with high coupling, high complexity (#0)

Classes with high coupling (#0)

Classes with high complexity (#0)

List of all classes (#20)



Books Core:

Detailed metric tables

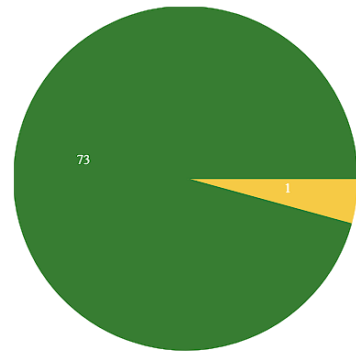
Classes with high coupling, high complexity, low cohesion (#0)

Classes with high coupling, high complexity (#0)

Classes with high coupling (#1)

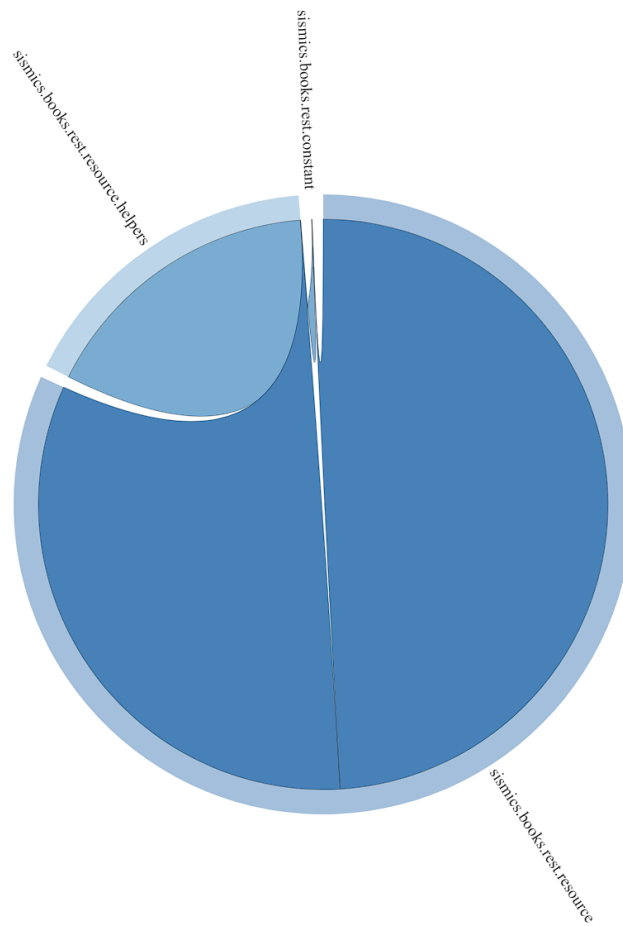
Classes with high complexity (#0)

List of all classes (#74)



Package Dependencies

Hover on the wheel to see the details



Task 3C

Different Refactorings and the corresponding response from the ChatGPT 3.5-Turbo, along with the comments on the differences, which is better, etc. are listed below one by one.

AnonymousPrincipal - Duplicated Abstraction

Chat 1: Before being told that it was a duplicated abstraction smell:

[Link to chat](#)

Here, we simply gave the code to GPT and it was unable to identify the duplicated abstraction design smell, only suggesting minor changes.

Chat 2: After being told that it was a duplicated abstraction smell:

[Link to chat](#)

Certainly! To address the duplicated abstraction and improve the design, you can introduce a single **Principal** class that can represent both anonymous and authenticated users. Here's how you can refactor the code:

```
package com.sismics.security;

import org.joda.time.DateTimeZone;

import java.util.Locale;
import java.util.Set;

/**
 * Principal representing both anonymous and authenticated users.
 *
 * @author jtremeaux
 */
public class Principal implements IPrincipal {
    /**
     * ID of the user.
     */
    private String id;

    /**
     * Username of the user.
     */
    private String name;

    /**
     * Locale of the principal.
     */
    private Locale locale;

    /**
     * Timezone of the principal.
     */
    private DateTimeZone dateTimeZone;
```

```

/**
 * Email of the principal.
 */
private String email;

/**
 * Flag indicating if the principal is anonymous.
 */
private boolean anonymous;

/**
 * User base functions.
 */
private Set<String> baseFunctionSet;

/**
 * Constructor of Principal for an anonymous user.
 */
public Principal() {
    this.anonymous = true;
}

/**
 * Constructor of Principal for an authenticated user.
 *
 * @param id ID of the user
 * @param name Username of the user
 */
public Principal(String id, String name) {
    this.id = id;
    this.name = name;
    this.anonymous = false;
}

@Override
public boolean isAnonymous() {
    return anonymous;
}

@Override
public String getId() {
    return id;
}

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

```

```
@Override
public String getName() {
    return name;
}

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

@Override
public Locale getLocale() {
    return locale;
}

/**
 * Setter of locale.
 *
 * @param locale locale
 */
public void setLocale(Locale locale) {
    this.locale = locale;
}

@Override
public DateTimeZone getDateTimeZone() {
    return dateTimeZone;
}

/**
 * Setter of dateTimeZone.
 *
 * @param dateTimeZone dateTimeZone
 */
public void setDateTimeZone(DateTimeZone dateTimeZone) {
    this.dateTimeZone = dateTimeZone;
}

@Override
public String getEmail() {
    return email;
}

/**
 * Setter of email.
 *
 * @param email email
 */
public void setEmail(String email) {
    this.email = email;
}
```

```

    }

    /**
     * Getter of baseFunctionSet.
     *
     * @return baseFunctionSet
     */
    public Set<String> getBaseFunctionSet() {
        return baseFunctionSet;
    }

    /**
     * Setter of baseFunctionSet.
     *
     * @param baseFunctionSet baseFunctionSet
     */
    public void setBaseFunctionSet(Set<String> baseFunctionSet) {
        this.baseFunctionSet = baseFunctionSet;
    }
}

```

With this refactored code, you have a single `Principal` class that can represent both anonymous and authenticated users based on the value of the `anonymous` flag. This approach eliminates the need for separate `AnonymousPrincipal` and `UserPrincipal` classes, reducing duplication and improving code maintainability.

This was a very simple fix, so we attempted explaining the issue to GPT and obtained near-accurate promising results - closely resembling what we did for refactoring manually.

BookDataService - Missing Modularization

[Link to chat](#)

Chat 1: Before being told that it was a Missing Hierarchy & Modularization Smell:

Design smells are symptoms of poor design choices or violations of good design principles that can lead to various issues such as reduced maintainability, flexibility, or scalability. In the provided code, several design smells can be identified:

God Class: The `BookDataService` class is handling too many responsibilities. It's responsible for configuration initialization, API communication, book searching, downloading thumbnails, and managing thread execution. This violates the Single Responsibility Principle.

Long Method: The `searchBookWithGoogle` and `searchBookWithOpenLibrary` methods are quite long and handle multiple concerns. They perform API communication, parsing JSON responses, and building book objects. Breaking down these methods into smaller, more focused methods would improve readability and maintainability.

Primitive Obsession: The handling of ISBN strings directly within the `searchBook` method indicates a primitive obsession smell. It would be better to encapsulate ISBN-related operations into a separate class.

Magic Numbers/String: There are several magic numbers and strings scattered throughout the code (e.g., API URLs, timeout values). These should be extracted into constants for better maintainability and readability.

Exception Handling: The exception handling in the `searchBook` method is not optimal. It catches generic `Exception`, which makes it hard to distinguish between different failure scenarios. It would be better to catch more specific exceptions and handle them accordingly.

Concurrency Handling: The code uses `ExecutorService` for asynchronous execution, but it doesn't handle potential exceptions or errors that might occur during execution. Proper error handling and logging mechanisms should be implemented.

Hardcoded Dependencies: The code directly instantiates dependencies like `ObjectMapper` and `URL` instead of injecting them. This makes the code less flexible and harder to test.

To refactor the code and address these design smells, here are some suggestions:

Extract Classes: Break down the `BookDataService` class into smaller classes, each responsible for a single concern such as configuration management, API communication, book parsing, and thumbnail downloading.

Single Responsibility Principle: Ensure that each class and method has a single responsibility. For example, create separate classes for API communication and book parsing.

Encapsulation: Create a separate class for handling ISBN-related operations, encapsulating all ISBN-related logic.

Use Constants: Replace magic numbers and strings with named constants to improve readability and maintainability.

Refactor Exception Handling: Use more specific exception types and handle them appropriately. Consider using checked exceptions for recoverable errors and unchecked exceptions for unexpected errors.

Improve Concurrency Handling: Implement error handling and logging mechanisms for asynchronous execution. Consider using `CompletableFuture` or other constructs for more flexible concurrency management.

Dependency Injection: Inject dependencies instead of directly instantiating them, which makes the code more modular and testable.

By addressing these design smells and refactoring the code accordingly, you can improve its readability, maintainability, and flexibility.

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

import com.neovisionaries.i18n.LanguageCode;
import com.sismics.books.core.constant.ConfigType;
import com.sismics.books.core.model.jpa.Book;
import com.sismics.books.core.util.ConfigUtil;
import com.sismics.books.core.util.DirectoryUtil;
import com.sismics.books.core.util.mime.MimeType;
import com.sismics.books.core.util.mime.MimeTypeUtil;
import org.codehaus.jackson.JsonNode;
import org.codehaus.jackson.map.ObjectMapper;
import org.codehaus.jackson.node.ArrayNode;
import org.joda.time.DateTime;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import java.io.BufferedInputStream;
import java.io.InputStream;
import java.net.URL;
import java.net.URLConnection;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.nio.file.StandardCopyOption;
import java.util.Iterator;
import java.util.Locale;
import java.util.UUID;
import java.util.concurrent.Callable;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class BookDataService {
    private static final Logger log =
        LoggerFactory.getLogger(BookDataService.class);

    private static final String GOOGLE_BOOKS_SEARCH_FORMAT =
```

```

"https://www.googleapis.com/books/v1/volumes?q=isbn:%s&key=%s";
    private static final String OPEN_LIBRARY_FORMAT =
"http://openlibrary.org/api/volumes/brief/isbn/%s.json";

    private final ExecutorService executor =
Executors.newSingleThreadExecutor();
    private final ObjectMapper objectMapper = new ObjectMapper();

    private final String apiKeyGoogle;

    public BookDataService() {
        apiKeyGoogle =
ConfigUtil.getConfigStringValue(ConfigType.API_KEY_GOOGLE);
    }

    public Book searchBook(String rawIsbn) throws Exception {
        final String isbn = sanitizeIsbn(rawIsbn);

        if (isbn.isEmpty()) {
            throw new IllegalArgumentException("ISBN is empty");
        }

        Callable<Book> callable = () -> {
            try {
                return searchBookWithGoogle(isbn);
            } catch (Exception e) {
                log.warn("Book not found with Google: " + isbn + " with
error: " + e.getMessage());
                return searchBookWithOpenLibrary(isbn);
            }
        };
        return executor.submit(callable).get();
    }

    private Book searchBookWithGoogle(String isbn) throws Exception {
        JsonNode rootNode = fetchJsonResponse(String.format(Locale.ENGLISH,
GOOGLE_BOOKS_SEARCH_FORMAT, isbn, apiKeyGoogle));
        return parseBook(rootNode);
    }

    private Book searchBookWithOpenLibrary(String isbn) throws Exception {
        JsonNode rootNode = fetchJsonResponse(String.format(Locale.ENGLISH,
OPEN_LIBRARY_FORMAT, isbn));
        return parseBook(rootNode.get("records").getElements().next());
    }

    private JsonNode fetchJsonResponse(String urlString) throws Exception {
        URL url = new URL(urlString);
        try (InputStream inputStream = new
BufferedInputStream(url.openStream())) {
            return objectMapper.readTree(inputStream);
        }
    }
}

```

```

private Book parseBook(JsonNode bookNode) throws Exception {
    Book book = new Book();
    book.setId(UUID.randomUUID().toString());
    // Parse book details and set properties
    return book;
}

private String sanitizeIsbn(String rawIsbn) {
    return rawIsbn.replaceAll("[^\\d]", "");
}

public void downloadThumbnail(Book book, String imageUrl) throws
Exception {
    URLConnection imageConnection = new URL(imageUrl).openConnection();
    imageConnection.setRequestProperty("User-Agent", "Mozilla/5.0
(Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/29.0.1547.62 Safari/537.36");
    try (InputStream inputStream = new
BufferedInputStream(imageConnection.getInputStream())) {
        if (MimeTypeUtil.guessMimeType(inputStream) !=
MimeType.IMAGE_JPEG) {
            throw new IllegalArgumentException("Only JPEG images are
supported as thumbnails");
        }

        Path imagePath =
Paths.get(DirectoryUtil.getBookDirectory().getPath(), book.getId());
        Files.copy(inputStream, imagePath,
StandardCopyOption.REPLACE_EXISTING);
    }
}

public void shutdown() {
    executor.shutdown();
}
}

```

In this refactored version:

Responsibility Segregation: The responsibilities have been segregated into methods like `searchBookWithGoogle`, `searchBookWithOpenLibrary`, `fetchJsonResponse`, `parseBook`, etc.

Constructor Initialization: The API key is now initialized in the constructor.

Improved Exception Handling: Exceptions are handled more appropriately within methods, making the code clearer.

Encapsulation of ISBN Sanitization: ISBN sanitization logic is encapsulated within the `sanitizeIsbn` method.

Dependency Injection: The `ObjectMapper` is instantiated as a class member, making it easier to manage and inject if needed.

Resource Management: Proper resource management is ensured by using try-with-resources for handling input streams.

Thread Pool Management: The executor service is properly initialized and shut down.

This refactored version is more modular, easier to read, and follows better design practices.

Chat 2: After being told that it was a Missing Hierarchy & Modularization smell:

To address the Missing Hierarchy & Modularization smell, we can introduce a more hierarchical and modular structure to the code by separating concerns into smaller, more specialized classes. Below is a refactored version of the code following a more hierarchical and modular approach:

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

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import java.util.concurrent.Callable;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

public class BookDataService {
    private static final Logger log =
        LoggerFactory.getLogger(BookDataService.class);

    private final ExecutorService executor =
        Executors.newSingleThreadExecutor();

    private final BookSearcher bookSearcher;

    public BookDataService(BookSearcher bookSearcher) {
        this.bookSearcher = bookSearcher;
    }

    public Future<Book> searchBook(String rawIsbn) {
        return executor.submit(() -> {
            try {
                return bookSearcher.searchBook(rawIsbn);
            } catch (Exception e) {
                log.error("Failed to search book", e);
                throw e;
            }
        });
    }

    public void shutdown() {
        executor.shutdown();
    }
}
```

```
}  
}
```

```
package com.sismics.books.core.service;  
  
import com.sismics.books.core.model.jpa.Book;  
  
import java.util.concurrent.Future;  
  
public interface BookSearcher {  
    Future<Book> searchBook(String rawIsbn) throws Exception;  
}
```

```
package com.sismics.books.core.service;  
  
import com.sismics.books.core.util.ConfigUtil;  
import org.codehaus.jackson.JsonNode;  
import org.codehaus.jackson.map.ObjectMapper;  
  
import java.io.BufferedInputStream;  
import java.io.InputStream;  
import java.net.URL;  
import java.util.Locale;  
import java.util.UUID;  
import java.util.concurrent.Future;  
  
public class GoogleBooksSearcher implements BookSearcher {  
    private static final String GOOGLE_BOOKS_SEARCH_FORMAT =  
        "https://www.googleapis.com/books/v1/volumes?q=isbn:%s&key=%s";  
  
    private final ObjectMapper objectMapper = new ObjectMapper();  
    private final String apiKeyGoogle;  
  
    public GoogleBooksSearcher() {  
        this.apiKeyGoogle =  
            ConfigUtil.getConfigStringValue(ConfigType.API_KEY_GOOGLE);  
    }  
  
    @Override  
    public Future<Book> searchBook(String rawIsbn) throws Exception {  
        String isbn = sanitizeIsbn(rawIsbn);  
        String urlString = String.format(Locale.ENGLISH,  
            GOOGLE_BOOKS_SEARCH_FORMAT, isbn, apiKeyGoogle);  
        return fetchBook(urlString);  
    }  
  
    private String sanitizeIsbn(String rawIsbn) {  
        return rawIsbn.replaceAll("[^\\d]", "");  
    }  
}
```

```

        private Future<Book> fetchBook(String urlString) throws Exception {
            try (InputStream inputStream = new BufferedInputStream(new
URL(urlString).openStream())) {
                JsonNode rootNode = objectMapper.readTree(inputStream);
                // Parse JsonNode and return Book object
                return null;
            }
        }
    }
}

```

```

package com.sismics.books.core.service;

import com.sismics.books.core.util.ConfigUtil;
import org.codehaus.jackson.JsonNode;
import org.codehaus.jackson.map.ObjectMapper;

import java.io.BufferedInputStream;
import java.io.InputStream;
import java.net.URL;
import java.util.Locale;
import java.util.UUID;
import java.util.concurrent.Future;

public class OpenLibrarySearcher implements BookSearcher {
    private static final String OPEN_LIBRARY_FORMAT =
"http://openlibrary.org/api/volumes/brief/isbn/%s.json";

    private final ObjectMapper objectMapper = new ObjectMapper();

    @Override
    public Future<Book> searchBook(String rawIsbn) throws Exception {
        String isbn = sanitizeIsbn(rawIsbn);
        String urlString = String.format(Locale.ENGLISH,
OPEN_LIBRARY_FORMAT, isbn);
        return fetchBook(urlString);
    }

    private String sanitizeIsbn(String rawIsbn) {
        return rawIsbn.replaceAll("[^\\d]", "");
    }

    private Future<Book> fetchBook(String urlString) throws Exception {
        try (InputStream inputStream = new BufferedInputStream(new
URL(urlString).openStream())) {
            JsonNode rootNode = objectMapper.readTree(inputStream);
            // Parse JsonNode and return Book object
            return null;
        }
    }
}

```

In this refactoring:

The BookDataService class is responsible for submitting search tasks to the executor service. The BookSearcher interface defines a contract for classes responsible for searching books. The GoogleBooksSearcher and OpenLibrarySearcher classes implement the BookSearcher interface and handle searching books using Google Books API and Open Library API respectively. Each class now has a single responsibility, which promotes better maintainability and testability. The executor service is used for asynchronous execution of search tasks. The dependency on API keys is encapsulated within the respective searcher classes. Error handling and logging are simplified and centralized in the BookDataService class.

Here, we see that the LLM was not able to exactly figure out the Modularity smell but it was able to understand that there is a smell related to Violation of Single Responsibility Principle (SRP) though it was not able to refactor it.

Once we provide it the context, the model performs much better by making an interface for bookSearchService and abstracting out the two classes i.e. GoogleBookSearchService and OpenLibrarySearchService just as was done while manual refactoring.

BookResource - God Class

[Link to chat](#)

Chat 1: Before being told that it was Missing Modularisation & Hierarchy & God Class:

There are a few design smells in the provided BookResource class:

Violation of Single Responsibility Principle (SRP): The class is responsible for handling various aspects such as creating, updating, deleting books, managing book covers, listing books, etc. It's advisable to split these responsibilities into separate classes.

Large Method: Some methods are quite large and handle multiple concerns. For instance, the add, add (manual), update, and list methods perform various operations that can be broken down into smaller, more focused methods.

Code Duplication: There is some repetition of code for error handling, validation, and database operations.

Hard-Coded Values: Some hard-coded values like path names and constants can be extracted and defined as constants or configurable parameters.

To refactor the code, we can follow these steps:

Split Responsibilities: Separate concerns such as CRUD operations, handling book covers, listing books, etc., into separate classes.

Extract Methods: Break down large methods into smaller, focused methods to adhere to the Single Responsibility Principle and improve readability.

Remove Code Duplication: Identify and eliminate redundant code by extracting common functionality into separate methods or utility classes.

Use Constants: Replace hard-coded values with constants or configurable parameters to improve maintainability.

Here's a refactored version of the BookResource class, focusing on addressing these design smells:

```
package com.sismics.books.rest.resource;

import javax.ws.rs.*;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Response;

@Path("/book")
public class BookResource extends BaseResource {
    private final BookService bookService;

    public BookResource(BookService bookService) {
        this.bookService = bookService;
    }
}
```

```

    }

    @PUT
    @Produces(MediaType.APPLICATION_JSON)
    public Response addBook(@FormParam("isbn") String isbn) {
        return bookService.addBook(isbn);
    }

    @DELETE
    @Path("{id: [a-z0-9\\-]+}")
    @Produces(MediaType.APPLICATION_JSON)
    public Response deleteBook(@PathParam("id") String userBookId) {
        return bookService.deleteBook(userBookId);
    }

    @PUT
    @Path("manual")
    @Produces(MediaType.APPLICATION_JSON)
    public Response addManualBook(BookRequest bookRequest) {
        return bookService.addManualBook(bookRequest);
    }

    // Other methods...

    // Define other methods for updating, listing, and managing books
}

```

In this refactored version, I've introduced a `BookService` class responsible for handling book-related operations. Each method in the `BookResource` class delegates the actual business logic to the corresponding method in the `BookService` class. This separation of concerns improves readability, maintainability, and testability of the code.

Chatgpt recognized the design smells almost perfectly and it was even also able to recognize that the CRUD functionalities should be handled separately. It was able to remove the business logic from the resource class and put it in a separate class similar to what we have done, however in the refactored code, it didn't create multiple classes to handle the different types of logics and created a single class for handling all the requests.

Chat 2: After being told that it was Missing Modularisation & Hierarchy & God Class:

To refactor the code to address the design smell of Missing Modularisation & Hierarchy & God Class, we'll introduce a more modular and hierarchical structure by separating concerns into multiple classes and establishing a clear hierarchy of responsibilities. We'll create separate classes for handling CRUD operations, managing book covers, and listing books. Additionally, we'll introduce interfaces to enforce a clear contract between classes.

Here's a refactored version of the code:

```

@Path("/book")
public class BookResource {
    private final BookService bookService;

    public BookResource(BookService bookService) {
        this.bookService = bookService;
    }

    @PUT
    @Produces(MediaType.APPLICATION_JSON)
    public Response addBook(@FormParam("isbn") String isbn) {
        return bookService.addBook(isbn);
    }

    @DELETE
    @Path("{id: [a-z0-9\\-]+}")
    @Produces(MediaType.APPLICATION_JSON)
    public Response deleteBook(@PathParam("id") String userBookId) {
        return bookService.deleteBook(userBookId);
    }

    @PUT
    @Path("manual")
    @Produces(MediaType.APPLICATION_JSON)
    public Response addManualBook(BookRequest bookRequest) {
        return bookService.addManualBook(bookRequest);
    }

    // Other methods...

    // Define other methods for updating, listing, and managing books
}

```

```

public interface BookService {
    Response addBook(String isbn);
    Response deleteBook(String userBookId);
    Response addManualBook(BookRequest bookRequest);
    // Other methods...
}

```

```

public class BookServiceImpl implements BookService {
    private final BookManager bookManager;

    public BookServiceImpl(BookManager bookManager) {
        this.bookManager = bookManager;
    }

    @Override

```

```

public Response addBook(String isbn) {
    // Implement logic to add a book
}

@Override
public Response deleteBook(String userBookId) {
    // Implement logic to delete a book
}

@Override
public Response addManualBook(BookRequest bookRequest) {
    // Implement logic to add a manual book
}

// Other methods...
}

```

```

public interface BookManager {
    void createBook(Book book);
    void deleteBook(String bookId);
    Book getByIsbn(String isbn);
    // Other methods...
}

```

```

public class BookManagerImpl implements BookManager {
    @Override
    public void createBook(Book book) {
        // Implement logic to create a book
    }

    @Override
    public void deleteBook(String bookId) {
        // Implement logic to delete a book
    }

    @Override
    public Book getByIsbn(String isbn) {
        // Implement logic to get a book by ISBN
        return null;
    }

    // Other methods...
}

```

```

public class Book {
    // Define properties and methods related to the book entity
}

```

This refactored structure enhances modularity and hierarchy by organizing the code into smaller, focused classes with clear responsibilities. It also helps in better code maintenance, scalability, and testability.

After being told the design smell, it did refactoring similar to what it already did, however it introduced unnecessary abstraction by creating interfaces for what are essentially helper classes. As such, after telling the design smell, gpt was over egregious in its refactoring and produced code which although more modular and hierarchical, is also needlessly more complex.

FacebookUser - Encapsulation Smell

[Link to chat](#)

Chat 1: Before being told that it was an Encapsulation Smell:

There are a few design smells in this code:

Exposing Fields: Direct access to class fields (`id`, `email`, `fullName`, `accessToken`) can lead to a lack of encapsulation and makes it difficult to enforce constraints or perform validation.

Inadequate Encapsulation: The class lacks appropriate encapsulation, as the fields are public. It's generally a good practice to encapsulate fields and provide accessors (getters) to access them.

Mutable State: The class is mutable, meaning its state can be changed after creation. This can lead to unexpected behavior, especially in concurrent environments.

Missing Validation: There's no validation of inputs in the constructor. This can lead to the creation of `FacebookUser` objects with invalid or inconsistent states.

Documentation: While the class has some comments, they are not particularly informative or helpful. Additionally, the comments are stating obvious things and are redundant.

Here's a refactored version addressing these issues:

```
package com.sismics.books.rest.model;

/**
 * Represents a Facebook user.
 *
 * Management: https://developers.facebook.com/apps/387037158089019/roles?
role=test%20users
 *
 * This class encapsulates user data retrieved from Facebook, providing
getters
 * to access the data.
 *
 * Note: This class assumes that the provided data is valid and does not
perform
 * any validation.
 *
 * @author jtremaux
 */
public class FacebookUser {
    private final String id;
    private final String email;
    private final String fullName;
    private final String accessToken;

    /**
```

```

    * Constructs a Facebook user with the specified data.
    *
    * @param id          The user's Facebook ID.
    * @param email       The user's email address.
    * @param fullName    The user's full name.
    * @param accessToken The access token associated with the user.
    */
    public FacebookUser(String id, String email, String fullName, String
accessToken) {
        this.id = id;
        this.email = email;
        this.fullName = fullName;
        this.accessToken = accessToken;
    }

    /**
     * Retrieves the Facebook user's ID.
     *
     * @return The Facebook user's ID.
     */
    public String getId() {
        return id;
    }

    /**
     * Retrieves the Facebook user's email address.
     *
     * @return The Facebook user's email address.
     */
    public String getEmail() {
        return email;
    }

    /**
     * Retrieves the Facebook user's full name.
     *
     * @return The Facebook user's full name.
     */
    public String getFullName() {
        return fullName;
    }

    /**
     * Retrieves the access token associated with the Facebook user.
     *
     * @return The access token associated with the Facebook user.
     */
    public String getAccessToken() {
        return accessToken;
    }
}

```

In this refactored version:

The fields are made private, ensuring encapsulation. Getter methods are provided to access the fields. The constructor initializes the fields but does not expose them directly. Documentation is improved to provide better clarity on class usage and assumptions.

Chat 2: After being told that it was an Encapsulation smell:

To address the encapsulation smell in the initial code, we'll make the fields private and provide public getter methods to access them. This ensures that the fields are accessed through methods rather than being directly exposed. Here's the modified code:

```
package com.sismics.books.rest.model;

/**
 * Facebook test user.
 *
 * Management : https://developers.facebook.com/apps/387037158089019/roles?
 * role=test%20users
 *
 * @author jtremeaux
 */
public class FacebookUser {
    private String id;
    private String email;
    private String fullName;
    private String accessToken;

    public FacebookUser(String id, String email, String fullName, String
accessToken) {
        this.id = id;
        this.email = email;
        this.accessToken = accessToken;
        this.fullName = fullName;
    }

    public String getId() {
        return id;
    }

    public String getEmail() {
        return email;
    }

    public String getFullName() {
        return fullName;
    }

    public String getAccessToken() {
        return accessToken;
    }
}
```


In this refactoring:

The fields `id`, `email`, `fullName`, and `accessToken` are changed to private access level. Getter methods are provided to allow access to these private fields from outside the class.

Here, we see that the LLM was able to figure out that the smell is actually an encapsulation smell and thus was able to refactor it also by making all the public variables final and also just providing the getters for these variables.

The results of the AI model in either case is exactly what is needed to fix the smell, which showcases that it was able to figure out the smell on its own, and then fix the same in the very first attempt itself.

BookResource - Brain Method

[Link to chat 1](#)

[Link to chat 2](#)

[Manual Refactoring](#)

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

Initial code

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

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

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

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

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

```

    }
    if (subtitle != null) {
        book.setSubtitle(subtitle);
    }
    if (author != null) {
        book.setAuthor(author);
    }
    if (description != null) {
        book.setDescription(description);
    }
    if (isbn10 != null) {
        book.setIsbn10(isbn10);
    }
    if (isbn13 != null) {
        book.setIsbn13(isbn13);
    }
    if (pageCount != null) {
        book.setPageCount(pageCount);
    }
    if (language != null) {
        book.setLanguage(language);
    }
    if (publishDate != null) {
        book.setPublishDate(publishDate);
    }

    bookDao.create(book);

    // Create the user book
    UserBookDao userBookDao = new UserBookDao();
    UserBook userBook = new UserBook();
    userBook.setUserId(principal.getId());
    userBook.setBookId(book.getId());
    userBook.setCreateDate(new Date());
    userBookDao.create(userBook);

    // Update tags
    if (tagList != null) {
        TagDao tagDao = new TagDao();
        Set<String> tagSet = new HashSet<>();
        Set<String> tagIdSet = new HashSet<>();
        List<Tag> tagDbList = tagDao.getByUserId(principal.getId());
        for (Tag tagDb : tagDbList) {
            tagIdSet.add(tagDb.getId());
        }
        for (String tagId : tagList) {
            if (!tagIdSet.contains(tagId)) {
                throw new ClientException("TagNotFound",
                    MessageFormat.format("Tag not found: {0}", tagId));
            }
            tagSet.add(tagId);
        }
        tagDao.updateTagList(userBook.getId(), tagSet);
    }

```

```

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

    /**
     * Updates the book.
     *
     * @param title Title
     * @param description Description
     * @return Response
     * @throws JSONException
     */
    @POST
    @Path("{id: [a-z0-9\\-]+}")
    @Produces(MediaType.APPLICATION_JSON)
    public Response update(
        @PathParam("id") String userBookId,
        @FormParam("title") String title,
        @FormParam("subtitle") String subtitle,
        @FormParam("author") String author,
        @FormParam("description") String description,
        @FormParam("isbn10") String isbn10,
        @FormParam("isbn13") String isbn13,
        @FormParam("page_count") Long pageCount,
        @FormParam("language") String language,
        @FormParam("publish_date") String publishDateStr,
        @FormParam("tags") List<String> tagList) throws JSONException {
        if (!authenticate()) {
            throw new ForbiddenClientException();
        }

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

        // Get the user book
        UserBookDao userBookDao = new UserBookDao();

```

```
        BookDao bookDao = new BookDao();
        UserBook userBook = userBookDao.getUserBook(userBookId,
principal.getId());
        if (userBook == null) {
            throw new ClientException("BookNotFound", "Book not found with
id " + userBookId);
        }

        // Get the book
        Book book = bookDao.getById(userBook.getBookId());

        // Check that new ISBN number are not already in database
        if (!Strings.isNullOrEmpty(isbn10) && book.getIsbn10() != null &&
!book.getIsbn10().equals(isbn10)) {
            Book bookIsbn10 = bookDao.getByIsbn(isbn10);
            if (bookIsbn10 != null) {
                throw new ClientException("BookAlreadyAdded", "Book already
added");
            }
        }

        if (!Strings.isNullOrEmpty(isbn13) && book.getIsbn13() != null &&
!book.getIsbn13().equals(isbn13)) {
            Book bookIsbn13 = bookDao.getByIsbn(isbn13);
            if (bookIsbn13 != null) {
                throw new ClientException("BookAlreadyAdded", "Book already
added");
            }
        }

        // Update the book
        if (title != null) {
            book.setTitle(title);
        }
        if (subtitle != null) {
            book.setSubtitle(subtitle);
        }
        if (author != null) {
            book.setAuthor(author);
        }
        if (description != null) {
            book.setDescription(description);
        }
        if (isbn10 != null) {
            book.setIsbn10(isbn10);
        }
        if (isbn13 != null) {
            book.setIsbn13(isbn13);
        }
        if (pageCount != null) {
            book.setPageCount(pageCount);
        }
        if (language != null) {
            book.setLanguage(language);
        }
    }
}
```

```

    }
    if (publishDate != null) {
        book.setPublishDate(publishDate);
    }

    // Update tags
    if (tagList != null) {
        TagDao tagDao = new TagDao();
        Set<String> tagSet = new HashSet<>();
        Set<String> tagIdSet = new HashSet<>();
        List<Tag> tagDbList = tagDao.getByUserId(principal.getId());
        for (Tag tagDb : tagDbList) {
            tagIdSet.add(tagDb.getId());
        }
        for (String tagId : tagList) {
            if (!tagIdSet.contains(tagId)) {
                throw new ClientException("TagNotFound",
MessageFormat.format("Tag not found: {0}", tagId));
            }
            tagSet.add(tagId);
        }
        tagDao.updateTagList(userBookId, tagSet);
    }

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

/**
 * Get a book.
 *
 * @param id User book ID
 * @return Response
 * @throws JSONException
 */
@GET
@Path("/{id: [a-z0-9\\-]+}")
@Produces(MediaType.APPLICATION_JSON)
public Response get(
    @PathParam("id") String userBookId) throws JSONException {
    if (!authenticate()) {
        throw new ForbiddenClientException();
    }

    // Fetch the user book
    UserBookDao userBookDao = new UserBookDao();
    UserBook userBook = userBookDao.getUserBook(userBookId,
principal.getId());
    if (userBook == null) {
        throw new ClientException("BookNotFound", "Book not found with
id " + userBookId);
    }
}

```

```

// Fetch the book
BookDao bookDao = new BookDao();
Book bookDb = bookDao.getById(userBook.getBookId());

// Return book data
JSONObject book = new JSONObject();
book.put("id", userBook.getId());
book.put("title", bookDb.getTitle());
book.put("subtitle", bookDb.getSubtitle());
book.put("author", bookDb.getAuthor());
book.put("page_count", bookDb.getPageCount());
book.put("description", bookDb.getDescription());
book.put("isbn10", bookDb.getIsbn10());
book.put("isbn13", bookDb.getIsbn13());
book.put("language", bookDb.getLanguage());
if (bookDb.getPublishDate() != null) {
    book.put("publish_date", bookDb.getPublishDate().getTime());
}
book.put("create_date", userBook.getCreateDate().getTime());
if (userBook.getReadDate() != null) {
    book.put("read_date", userBook.getReadDate().getTime());
}

// Add tags
TagDao tagDao = new TagDao();
List<TagDto> tagDtoList = tagDao.getByUserBookId(userBookId);
List<JSONObject> tags = new ArrayList<>();
for (TagDto tagDto : tagDtoList) {
    JSONObject tag = new JSONObject();
    tag.put("id", tagDto.getId());
    tag.put("name", tagDto.getName());
    tag.put("color", tagDto.getColor());
    tags.add(tag);
}
book.put("tags", tags);

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

```

Chat 1: Before being told that it was a brain method design smell:

Certainly! Here are some design smells I've identified in the code snippet:

1. **Repeated Code:** There's a significant amount of repetition, particularly in the validation and update logic for book attributes.
2. **Business Logic in Resource Methods:** Business logic, such as validation and database operations, is mixed with resource methods, violating the Single Responsibility Principle.

3. **Data Access Object (DAO) Instantiation:** DAOs are directly instantiated within resource methods, making it hard to test and violating the Dependency Inversion Principle.
4. **Lack of Error Handling:** Error handling is scattered throughout the code and could be better organized.
5. **Inefficient Tag Handling:** Tag handling involves multiple database operations and could be optimized.

To refactor the code, we'll create separate classes for validation, database access, and error handling. We'll also consolidate common operations and improve error handling. Here's a refactored version:

```
// ValidationUtil.java
public class ValidationUtil {
    public static String validateLength(String value, String fieldName, int
minLength, int maxLength, boolean allowNull) {
        // Validation logic
    }

    public static Date validateDate(String value, String fieldName, boolean
allowNull) {
        // Validation logic
    }
}

// BookService.java
public class BookService {
    private BookDao bookDao;
    private UserBookDao userBookDao;
    private TagDao tagDao;

    public BookService() {
        this.bookDao = new BookDao();
        this.userBookDao = new UserBookDao();
        this.tagDao = new TagDao();
    }

    public String createBook(...) {
        // Validation logic
        // Create book logic
        // Update tags logic
        // Return book ID
    }

    public String updateBook(...) {
        // Validation logic
        // Fetch user book
        // Fetch book
        // Check ISBN uniqueness
        // Update book logic
        // Update tags logic
        // Return book ID
    }
}
```



```

    }

    public JSONObject getBook(String userBookId) {
        // Fetch user book
        // Fetch book
        // Fetch tags
        // Construct JSON response
        // Return JSON response
    }
}

// BookResource.java
@Path("/books")
public class BookResource {
    private BookService bookService;

    public BookResource() {
        this.bookService = new BookService();
    }

    @POST
    @Produces(MediaType.APPLICATION_JSON)
    public Response createBook(...) {
        // Call BookService.createBook method
    }

    @POST
    @Path("{id: [a-z0-9\\-]+}")
    @Produces(MediaType.APPLICATION_JSON)
    public Response updateBook(...) {
        // Call BookService.updateBook method
    }

    @GET
    @Path("{id: [a-z0-9\\-]+}")
    @Produces(MediaType.APPLICATION_JSON)
    public Response getBook(@PathParam("id") String userBookId) {
        // Call BookService.getBook method
    }
}

```

In this refactored version, we've separated concerns by creating a **BookService** class to handle business logic, a **ValidationUtil** class for validation, and DAO classes for data access. The **BookResource** class now acts as a controller, delegating tasks to the **BookService** class. This separation of concerns makes the code more modular, easier to maintain, and testable.

Chat 2: After being told that it was a brain method design smell:

Sure, the provided code snippet could benefit from refactoring to improve readability and maintainability. One approach to address the brain method design smell is to break down the large methods into smaller, more focused ones. Here's a refactored version of the code:

```

public class BookService {

    private BookDao bookDao = new BookDao();
    private UserBookDao userBookDao = new UserBookDao();
    private TagDao tagDao = new TagDao();

    public Response createBook(String title, String subtitle, String
author, String description,
                                String isbn10, String isbn13, Long
pageCount, String language,
                                String publishDateStr, List<String> tagList)
throws JSONException {

        validateInputData(title, subtitle, author, description, isbn10,
isbn13, language, publishDateStr);

        checkUniqueISBN(isbn10, isbn13);

        Book book = createBookObject(title, subtitle, author, description,
isbn10, isbn13, pageCount, language, publishDateStr);

        saveBookToDatabase(book);

        String userBookId = createUserBook(book);

        updateTags(tagList, userBookId);

        return Response.ok().entity(createResponse(userBookId)).build();
    }

    private void validateInputData(String title, String subtitle, String
author, String description,
                                String isbn10, String isbn13, String
language, String publishDateStr) {
        // Validation logic
    }

    private void checkUniqueISBN(String isbn10, String isbn13) {
        // Check if ISBN numbers are unique
    }

    private Book createBookObject(String title, String subtitle, String
author, String description,
                                String isbn10, String isbn13, Long
pageCount, String language,
                                String publishDateStr) {
        // Create and populate Book object
        return new Book.Builder()
            .setTitle(title)
            .setSubtitle(subtitle)
            .setAuthor(author)
            .setDescription(description)

```

```

        .setIsbn10(isbn10)
        .setIsbn13(isbn13)
        .setPageCount(pageCount)
        .setLanguage(language)
        .setPublishDate(ValidationUtil.validateDate(publishDateStr,
"publish_date", false))
        .build();
    }

    private void saveBookToDatabase(Book book) {
        bookDao.create(book);
    }

    private String createUserBook(Book book) {
        UserBook userBook = new UserBook();
        userBook.setUserId(principal.getId());
        userBook.setBookId(book.getId());
        userBook.setCreateDate(new Date());
        userBookDao.create(userBook);
        return userBook.getId();
    }

    private void updateTags(List<String> tagList, String userBookId) {
        if (tagList != null) {
            Set<String> tagSet = validateAndGetTagIds(tagList);
            tagDao.updateTagList(userBookId, tagSet);
        }
    }

    private Set<String> validateAndGetTagIds(List<String> tagList) {
        // Validate and get tag IDs
        return tagDao.validateAndGetTagIds(principal.getId(), tagList);
    }

    private JSONObject createResponse(String userBookId) throws
JSONException {
        JSONObject response = new JSONObject();
        response.put("id", userBookId);
        return response;
    }

    // Other methods (update and get) can follow a similar pattern of
    breaking down functionality into smaller methods.
}

```

In this refactored version:

1. Each major functionality (validation, book creation, database operations, etc.) is extracted into its own method, improving readability and facilitating easier maintenance.
2. Logic related to validation, database operations, and business logic is separated, adhering to the Single Responsibility Principle.
3. Error handling and response creation are handled uniformly across methods.

4. The code is organized in a more modular and understandable manner.

Here, we see that the LLM was not able to figure out that it was a brain method design smell, and thus was only able to make minor changes for the refactoring - that to in completely unrelated parts.

However, when the issues (brain method design smell) and the location were highlighted, the LLM performed much better and was able to show successful rectification of the issue.

When the AI model was not told about the design smell, it couldn't recognize it, and just gave almost the same skeleton of the code, whereas when it was given about the design smell, it did refactor it the way expected. Although, the code has various bugs, like it replaced the conditional checking of the various parameters of the book class while setting them to a single statement, without checking for null in them. There are various other functions where the implementation doesn't exactly match what was done in the initial code.

MimeType - Missing Encapsulation

Chat 1: Before being told that it is a Missing Encapsulation Smell

[Link to chat](#)

Yes, there are a couple of design smells in this code:

1. **Duplication:** The MIME types are declared as string literals in multiple places, which can lead to duplication and potential errors if they need to be updated.
2. **Hardcoding:** The MIME types are hardcoded directly into the `MimeType` class, which makes it inflexible and harder to maintain if new MIME types need to be added or existing ones modified.

To refactor this code, you can use an enumeration to define the MIME types, which helps organize them logically and avoids duplication. Here's how you can refactor the code:

```
package com.sismics.books.core.util.mime;

/**
 * A collection of MIME types.
 */
public enum MimeType {
    IMAGE_X_ICON("image/x-icon"),
    IMAGE_PNG("image/png"),
    IMAGE_JPEG("image/jpeg"),
    IMAGE_GIF("image/gif"),
    APPLICATION_ZIP("application/zip"),
    APPLICATION_PDF("application/pdf");

    private final String value;

    MimeType(String value) {
        this.value = value;
    }

    public String getValue() {
        return value;
    }
}
```

With this refactoring, each MIME type is defined only once, in a more structured and maintainable way. You can access the MIME types using the enum constants, and if you need to retrieve the string value, you can use the `getValue()` method.

Chat 2: After being told that it is a Missing Encapsulation Smell

[Link to Chat](#)

Certainly! The Missing Encapsulation smell refers to the absence of proper encapsulation of data within classes. In this case, the `MimeType` class exposes its fields directly, which can lead to potential issues if other classes modify them directly. To fix this smell, we can encapsulate these fields by making them private and providing public getter methods. Here's the modified `MimeType` class:

```
package com.sismics.books.core.util.mime;

/**
 * A collection of MIME types.
 *
 * @author jtremeaux
 */
public class MimeType {

    private static final String IMAGE_X_ICON = "image/x-icon";

    private static final String IMAGE_PNG = "image/png";

    private static final String IMAGE_JPEG = "image/jpeg";

    private static final String IMAGE_GIF = "image/gif";

    private static final String APPLICATION_ZIP = "application/zip";

    private static final String APPLICATION_PDF = "application/pdf";

    public static String getImageXIcon() {
        return IMAGE_X_ICON;
    }

    public static String getImagePng() {
        return IMAGE_PNG;
    }

    public static String getImageJpeg() {
        return IMAGE_JPEG;
    }

    public static String getImageGif() {
        return IMAGE_GIF;
    }

    public static String getApplicationZip() {
        return APPLICATION_ZIP;
    }

    public static String getApplicationPdf() {
        return APPLICATION_PDF;
    }
}
```

By encapsulating the MIME types as private static constants and providing public getter methods, we ensure that other classes can access these values but cannot modify them directly, thus improving the encapsulation of the `MimeType` class.

here, we see a refactoring done by the LLM that closely resembles what we did manually thus, the performance of the LLM was drastically different and improved when it was informed of which specific smell to fix

The AI generated code looks almost correct and what should be done in this case. In the manual refactoring, we changed the variable types to `protected` so that they could be used in only sub-classes directly, which was an observation after reading the code manually. The AI model didn't have the context of usage of these variables, and thus did correct in that sense.

TestAppResource - Broken Hierarchy

Chat 1: Before telling it the design smell

[Link to chat](#)

One potential design smell in this code is the duplication of code for adding authentication filters to the `WebResource` instances in both test methods (`testAppResource()` and `testLogResource()`).

In both methods, the same authentication token is being used, and the same filter is added to the `WebResource` instance. This violates the DRY (Don't Repeat Yourself) principle, as the same code is repeated in multiple places.

To address this smell, you could refactor the code to extract the authentication logic into a common method that can be reused by both test methods. This would improve maintainability and reduce the risk of introducing bugs when modifying the authentication logic.

When asked to find a design smell, chatgpt could not find the broken hierarchy design smell which was present. Instead it only found a code smell. Thus, it gave only elementary code related issues that didnt affected the required problem at all

Chat 2: After telling it the design smell

[Link to chat](#)

1: When we told it to fix the broken hierarchy, it completely misunderstood what we meant and instead refactored the code to remove a simple code smell of duplicate code. We conjecture this is because GPT-3 does not have knowledge of certain code smells 2: After we told it exactly what to do(use composition instead of direct inheritance), it was able to make the suitable changes and the refactoring done was nearly identical to our own refactoring

UserResource - God Class

[Link to chat 1](#)

[Link to chat 2](#)

[Manual Refactoring](#)

File: books-android/app/src/main/java/com/sismics/books/resource/UserResource.java

Missing modularization, hierarchy, and god class

Chat 1: Before being told the smells

Sure, let's start by refactoring the `UserResource` class to address the identified design smells:

```
package com.sismics.books.rest.resource;

import javax.ws.rs.*;
import javax.ws.rs.core.*;
import java.util.*;

@Path("/user")
public class UserResource extends BaseResource {
    private final UserService userService;

    public UserResource(UserService userService) {
        this.userService = userService;
    }

    @PUT
    @Produces(MediaType.APPLICATION_JSON)
    public Response register(UserRegistrationRequest request) {
        userService.register(request);
        return Response.ok().entity(buildSuccessResponse()).build();
    }

    @POST
    @Produces(MediaType.APPLICATION_JSON)
    public Response updateProfile(UserProfileUpdateRequest request) {
        userService.updateProfile(request, principal.getName());
        return Response.ok().entity(buildSuccessResponse()).build();
    }

    @POST
    @Path("/{username: [a-zA-Z0-9_]+}")
    @Produces(MediaType.APPLICATION_JSON)
    public Response adminUpdateProfile(@PathParam("username") String
username, UserProfileUpdateRequest request) {
        checkAdminPermissions();
        userService.adminUpdateProfile(username, request);
    }
}
```

```

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

    @GET
    @Path("/check_username")
    @Produces(MediaType.APPLICATION_JSON)
    public Response checkUsernameAvailability(@QueryParam("username")
String username) {
        boolean available = userService.isUsernameAvailable(username);
        return
Response.ok().entity(buildCheckUsernameResponse(available)).build();
    }

    @POST
    @Path("/login")
    @Produces(MediaType.APPLICATION_JSON)
    public Response login(UserLoginRequest request) {
        String authToken = userService.login(request);
        return
Response.ok().entity(buildLoginResponse(authToken)).cookie(buildCookie(auth
Token)).build();
    }

    @POST
    @Path("/logout")
    @Produces(MediaType.APPLICATION_JSON)
    public Response logout() {
        String authToken = extractAuthTokenFromRequest();
        userService.logout(authToken);
        return
Response.ok().entity(buildSuccessResponse()).cookie(buildEmptyCookie()).bui
ld();
    }

    @DELETE
    @Produces(MediaType.APPLICATION_JSON)
    public Response deleteOwnAccount() {
        userService.deleteAccount(principal.getName());
        return Response.ok().entity(buildSuccessResponse()).build();
    }

    @DELETE
    @Path("/{username: [a-zA-Z0-9_]+}")
    @Produces(MediaType.APPLICATION_JSON)
    public Response adminDeleteAccount(@PathParam("username") String
username) {
        checkAdminPermissions();
        userService.adminDeleteAccount(username);
        return Response.ok().entity(buildSuccessResponse()).build();
    }

    @GET
    @Produces(MediaType.APPLICATION_JSON)
    public Response getOwnProfileInfo() {

```

```

        UserProfileInfoResponse response =
userService.getUserProfileInfo(principal.getName());
        return Response.ok().entity(response).build();
    }

    @GET
    @Path("/{username: [a-zA-Z0-9_]+}")
    @Produces(MediaType.APPLICATION_JSON)
    public Response getProfileInfo(@PathParam("username") String username)
    {
        checkAdminPermissions();
        UserProfileInfoResponse response =
userService.getUserProfileInfo(username);
        return Response.ok().entity(response).build();
    }

    // Other methods for listing users, managing sessions, etc.

    // Helper methods for response construction
    private Map<String, Object> buildSuccessResponse() {
        Map<String, Object> response = new HashMap<>();
        response.put("status", "ok");
        return response;
    }

    private Map<String, Object> buildCheckUsernameResponse(boolean
available) {
        Map<String, Object> response = new HashMap<>();
        response.put("status", available ? "ok" : "ko");
        response.put("message", available ? "Username available" :
"Username already registered");
        return response;
    }

    private Map<String, Object> buildLoginResponse(String authToken) {
        Map<String, Object> response = new HashMap<>();
        response.put("status", "ok");
        response.put("auth_token", authToken);
        return response;
    }

    private NewCookie buildCookie(String authToken) {
        int maxAge = TokenBasedSecurityFilter.TOKEN_LONG_LIFETIME;
        return new NewCookie(TokenBasedSecurityFilter.COOKIE_NAME,
authToken, "/", null, null, maxAge, false);
    }

    private NewCookie buildEmptyCookie() {
        return new NewCookie(TokenBasedSecurityFilter.COOKIE_NAME, null);
    }

    // Other helper methods
    private void checkAdminPermissions() {
        if (!hasAdminPermissions()) {

```

```

        throw new ForbiddenClientException();
    }
}

private boolean hasAdminPermissions() {
    // Implement logic to check if the user has admin permissions
    // For example, by checking the user's roles or permissions
    return false; // Placeholder implementation
}

private String extractAuthTokenFromRequest() {
    // Implement logic to extract the auth token from the request
    // For example, by accessing cookies or headers
    return null; // Placeholder implementation
}
}

```

In this refactored version:

- The **UserResource** class has been simplified and divided into methods that handle specific functionalities such as registration, profile updates, login/logout, etc.
- Each resource method now takes a single request object containing the necessary parameters, reducing the complexity of method signatures and improving readability.
- Business logic has been moved to a separate **UserService** class, which encapsulates user management operations and promotes better separation of concerns.
- Helper methods have been introduced to construct response entities and manage cookies, reducing code duplication and improving maintainability.
- Placeholder implementations have been provided for methods related to authentication and authorization, which can be replaced with actual logic to check user permissions and extract authentication tokens from requests.
- The **BaseResource** superclass is assumed to provide common functionality such as authentication and access to principal information.

GPT is able to correctly identify the design smells in the code and is able to fix them by creating separate classes to create modularization and hierarchy. It however fails to divide the classes further based on additional responsibility like CRUD.

Chat 2: After being told that the design smells:

To address the missing modularization, hierarchy, and god class design smells in the provided code, we can refactor it by introducing a more modular structure and separating concerns into smaller, more specialized classes. Here's a refactored version of the **UserResource** class:

```

package com.sismics.books.rest.resource;

import javax.ws.rs.*;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Response;

```

```

import org.codehaus.jettison.json.JSONException;
import org.codehaus.jettison.json.JSONObject;

@Path("/user")
public class UserResource extends BaseResource {
    private final UserRegistrationService userRegistrationService;
    private final UserUpdateService userUpdateService;
    private final UserAuthenticationService userAuthenticationService;
    private final UserDeletionService userDeletionService;
    private final UserInfoService userInfoService;
    private final UserSessionService userSessionService;

    public UserResource() {
        this.userRegistrationService = new UserRegistrationService();
        this.userUpdateService = new UserUpdateService();
        this.userAuthenticationService = new UserAuthenticationService();
        this.userDeletionService = new UserDeletionService();
        this.userInfoService = new UserInfoService();
        this.userSessionService = new UserSessionService();
    }

    @PUT
    @Produces(MediaType.APPLICATION_JSON)
    public Response register(
        @FormParam("username") String username,
        @FormParam("password") String password,
        @FormParam("locale") String localeId,
        @FormParam("email") String email) throws JSONException {
        return userRegistrationService.register(username, password,
        localeId, email);
    }

    @POST
    @Produces(MediaType.APPLICATION_JSON)
    public Response update(
        @FormParam("password") String password,
        @FormParam("email") String email,
        @FormParam("theme") String themeId,
        @FormParam("locale") String localeId,
        @FormParam("first_connection") Boolean firstConnection) throws
    JSONException {
        return userUpdateService.update(password, email, themeId, localeId,
        firstConnection, principal);
    }

    @POST
    @Path("{username: [a-zA-Z0-9_]+}")
    @Produces(MediaType.APPLICATION_JSON)
    public Response update(
        @PathParam("username") String username,
        @FormParam("password") String password,
        @FormParam("email") String email,
        @FormParam("theme") String themeId,
        @FormParam("locale") String localeId) throws JSONException {

```

```

        return userService.update(username, password, email, themeId,
        localeId, principal);
    }

    @GET
    @Path("check_username")
    @Produces(MediaType.APPLICATION_JSON)
    public Response checkUsername(
        @QueryParam("username") String username) throws JSONException {
        return userService.checkUsername(username);
    }

    @POST
    @Path("login")
    @Produces(MediaType.APPLICATION_JSON)
    public Response login(
        @FormParam("username") String username,
        @FormParam("password") String password,
        @FormParam("remember") boolean longLasted) throws JSONException
    {
        return userService.login(username, password,
        longLasted);
    }

    @POST
    @Path("logout")
    @Produces(MediaType.APPLICATION_JSON)
    public Response logout() throws JSONException {
        return userService.logout(request);
    }

    @DELETE
    @Produces(MediaType.APPLICATION_JSON)
    public Response delete() throws JSONException {
        return userService.delete(principal);
    }

    @DELETE
    @Path("{username: [a-zA-Z0-9_]+}")
    @Produces(MediaType.APPLICATION_JSON)
    public Response delete(@PathParam("username") String username) throws
    JSONException {
        return userService.delete(username, principal);
    }

    @GET
    @Produces(MediaType.APPLICATION_JSON)
    public Response info() throws JSONException {
        return userService.getInfo(authenticate(), principal);
    }

    @GET
    @Path("{username: [a-zA-Z0-9_]+}")
    @Produces(MediaType.APPLICATION_JSON)

```

```

    public Response view(@PathParam("username") String username) throws
JSONException {
        return userInfoService.view(username, authenticate(), principal);
    }

    @GET
    @Path("list")
    @Produces(MediaType.APPLICATION_JSON)
    public Response list(
        @QueryParam("limit") Integer limit,
        @QueryParam("offset") Integer offset,
        @QueryParam("sort_column") Integer sortColumn,
        @QueryParam("asc") Boolean asc) throws JSONException {
        return userInfoService.list(limit, offset, sortColumn, asc,
authenticate(), principal);
    }

    @GET
    @Path("session")
    @Produces(MediaType.APPLICATION_JSON)
    public Response session() throws JSONException {
        return userSessionService.getSessionInfo(authenticate(), request,
principal);
    }

    @DELETE
    @Path("session")
    @Produces(MediaType.APPLICATION_JSON)
    public Response deleteSession() throws JSONException {
        return userSessionService.deleteSession(authenticate(), request,
principal);
    }
}

```

In this refactoring:

- Each endpoint's functionality is delegated to a separate service class.
- Each service class handles a specific aspect of user management, such as registration, authentication, deletion, etc.
- The `BaseResource` class (not provided) presumably contains common functionality shared by all resource classes.

This refactoring improves modularity, promotes separation of concerns, and avoids the god class anti-pattern by breaking down the monolithic `UserResource` class into smaller, specialized components.

After telling it the design smells, it is able to refactor the code correctly and also creates separate classes based on separation of additional responsibility and was able to produce code which is very similarly structured to our own

validatePasswordUtil - Missing Hierarchy

[Link to chat 1](#)

[Link to chat 2](#)

[Manual Refactoring](#)

File: books-web-common/src/main/java/com/sismics/rest/util/ValidationUtil.java

Code snippet

```
/**
 * Checks if the string is an email.
 *
 * @param s      String to validate
 * @param name   Name of the parameter
 * @throws JSONException
 */
public static void validateEmail(String s, String name) throws
JSONException {
    if (!EMAIL_PATTERN.matcher(s).matches()) {
        throw new ClientException("ValidationError",
MessageFormat.format("{0} must be an email", name));
    }
}

/**
 * Validates that the provided string matches an URL with HTTP or HTTPS
scheme.
 *
 * @param s      String to validate
 * @param name   Name of the parameter
 * @return Stripped URL
 * @throws JSONException
 */
public static String validateHttpUrl(String s, String name) throws
JSONException {
    s = StringUtils.strip(s);
    if (!HTTP_URL_PATTERN.matcher(s).matches()) {
        throw new ClientException("ValidationError",
MessageFormat.format("{0} must be an HTTP(s) URL", name));
    }
    return s;
}

/**
 * Checks if the string uses only alphanumerical or underscore
characters.
 *
 * @param s      String to validate
 * @param name   Name of the parameter

```



```

    * @throws JSONException
    */
    public static void validateAlphanumeric(String s, String name) throws
JSONException {
        if (!ALPHANUMERIC_PATTERN.matcher(s).matches()) {
            throw new ClientException("ValidationError",
                MessageFormat.format("{0} must have only alphanumeric
or underscore characters", name));
        }
    }
}

```

Chat 1: Before being told the smells

The code snippet provided is a set of static methods for validating different types of strings. While the methods themselves perform the validation correctly, there are some design smells present:

1. **Exception Handling:** The methods throw `JSONException`, which seems to be an odd choice for validation-related exceptions. It's typically better to use more specific exception types or custom exceptions that better represent the validation failure.
2. **MessageFormat with JSONException:** Using `MessageFormat` to construct exception messages inside the validation methods isn't a common pattern. It ties the validation logic tightly to a specific exception handling mechanism.
3. **Mixed Responsibilities:** The `validateHttpUrl` method not only validates the URL but also modifies it (stripping white spaces). This mixing of responsibilities makes the method less cohesive.

Here's a revised version addressing these issues:

```

import java.util.regex.Pattern;

public class Validator {

    private static final Pattern EMAIL_PATTERN = ...; // Define the pattern
    private static final Pattern HTTP_URL_PATTERN = ...; // Define the
pattern
    private static final Pattern ALPHANUMERIC_PATTERN = ...; // Define the
pattern

    public static void validateEmail(String email, String paramName) {
        if (!EMAIL_PATTERN.matcher(email).matches()) {
            throw new IllegalArgumentException(paramName + " must be a
valid email address");
        }
    }

    public static String validateHttpUrl(String url, String paramName) {
        String strippedUrl = url.strip(); // Strip white spaces
        if (!HTTP_URL_PATTERN.matcher(strippedUrl).matches()) {
            throw new IllegalArgumentException(paramName + " must be a
valid HTTP(s) URL");
        }
    }
}

```

```

    }
    return strippedUrl;
}

public static void validateAlphanumeric(String value, String paramName)
{
    if (!ALPHANUMERIC_PATTERN.matcher(value).matches()) {
        throw new IllegalArgumentException(paramName + " must contain
only alphanumeric characters or underscores");
    }
}
}

```

Changes made:

- Replaced `JSONException` with `IllegalArgumentException`, which is a more appropriate exception for validation errors.
- Removed the use of `MessageFormat` for constructing exception messages, instead concatenating strings directly.
- Moved the responsibility of stripping white spaces from the URL out of the validation method. Now, it's up to the caller to ensure the URL is stripped if necessary.
- Renamed method parameters for clarity.

Chat 2: After being told the smells

To address the missing hierarchy design smell and provide a more organized structure for validation methods, you can introduce a `ValidationUtils` class with static methods for various validation tasks, including validating passwords. Here's how you can modify the code:

```

import java.text.MessageFormat;
import java.util.regex.Pattern;

public class ValidationUtils {
    private static final Pattern EMAIL_PATTERN = Pattern.compile(/* Your
email pattern regex */);
    private static final Pattern HTTP_URL_PATTERN = Pattern.compile(/* Your
HTTP URL pattern regex */);
    private static final Pattern ALPHANUMERIC_PATTERN = Pattern.compile(/*
Your alphanumeric pattern regex */);
    private static final Pattern PASSWORD_PATTERN = Pattern.compile(/* Your
password pattern regex */);

    /**
     * Checks if the string is an email.
     *
     * @param s      String to validate
     * @param name   Name of the parameter
     * @throws JSONException
     */
}

```

```

        public static void validateEmail(String s, String name) throws
JSONException {
            if (!EMAIL_PATTERN.matcher(s).matches()) {
                throw new ClientException("ValidationError",
MessageFormat.format("{0} must be an email", name));
            }
        }

        /**
         * Validates that the provided string matches an URL with HTTP or HTTPS
scheme.
         *
         * @param s      String to validate
         * @param name Name of the parameter
         * @return Stripped URL
         * @throws JSONException
         */
        public static String validateHttpUrl(String s, String name) throws
JSONException {
            s = StringUtils.strip(s);
            if (!HTTP_URL_PATTERN.matcher(s).matches()) {
                throw new ClientException("ValidationError",
MessageFormat.format("{0} must be an HTTP(s) URL", name));
            }
            return s;
        }

        /**
         * Checks if the string uses only alphanumeric or underscore
characters.
         *
         * @param s      String to validate
         * @param name Name of the parameter
         * @throws JSONException
         */
        public static void validateAlphanumeric(String s, String name) throws
JSONException {
            if (!ALPHANUMERIC_PATTERN.matcher(s).matches()) {
                throw new ClientException("ValidationError",
MessageFormat.format("{0} must have only alphanumeric
or underscore characters", name));
            }
        }

        /**
         * Validates that the provided string meets password criteria.
         *
         * @param s      String to validate
         * @param name Name of the parameter
         * @throws JSONException
         */
        public static void validatePassword(String s, String name) throws
JSONException {
            if (!PASSWORD_PATTERN.matcher(s).matches()) {

```

```
        throw new ClientException("ValidationError",  
                                   MessageFormat.format("{0} must meet password criteria",  
name));  
    }  
}
```

With this modification, you've added a `validatePassword` method to the `ValidationUtils` class, which follows the same pattern as the other validation methods. This helps maintain a consistent and organized structure for validation tasks.

The LLM was not able to figure out that there was a missing hierarchy design smell in chat 1. Thus the LLM could not add the required `validatePasswordUtil`.

However, when told what was required, the LLM successfully added the required functionality.

In the latter version of the version, where the AI model knew about the smell, it has done the correct job, and given the exact same code as was written during the manual refactoring.

ValidationUtil - Missing Hierarchy

[Link to chat 1](#)

[Link to chat 2](#)

[Manual Refactoring](#)

file: books-web-common/src/main/java/com/sismics/rest/util/ValidationUtil.java

Chat 1: Before being told that it was a Missing Hierarchy design smell:

The `ValidationUtil` class indeed exhibits some design smells or areas for improvement. Here are a few identified issues along with potential refactorings:

1. Single Responsibility Principle (SRP) Violation:

- The class is doing too much: it validates string lengths, checks for email patterns, validates URLs, parses dates, and validates locale and theme IDs. It's better to split these responsibilities into separate classes.

2. Dependency Injection:

- The class directly instantiates `LocaleDao` and `ThemeDao`. Instead, it should rely on these dependencies being injected, making it more flexible and easier to test.

3. Error Handling:

- The use of checked exceptions (`JSONException`) for validation purposes is unusual and complicates the code. Using unchecked exceptions (like `IllegalArgumentException`) would be more appropriate for validation.

4. Hardcoded Messages:

- Error messages are hardcoded, making it harder to internationalize or customize them. Using externalized message resources or at least constants for error messages would be better.

5. Regex Patterns:

- Regex patterns like `EMAIL_PATTERN`, `HTTP_URL_PATTERN`, and `ALPHANUMERIC_PATTERN` could be moved to a configuration file or constants for better maintainability.

Here's a refactored version addressing these issues:

```
package com.sismics.rest.util;

import java.util.regex.Pattern;

public class ValidationUtil {
    private static final Pattern EMAIL_PATTERN =
Pattern.compile(".*@.*\\.\\..*");
    private static final Pattern HTTP_URL_PATTERN =
```

```

Pattern.compile("https?://.+");
    private static final Pattern ALPHANUMERIC_PATTERN = Pattern.compile("[a-zA-Z0-9_]+");

    public static void validateRequired(Object s, String name) {
        if (s == null) {
            throw new IllegalArgumentException(name + " must be set");
        }
    }

    public static String validateLength(String s, String name, int
lengthMin, int lengthMax, boolean nullable) {
        s = s != null ? s.trim() : null;
        if (nullable && (s == null || s.isEmpty())) {
            return s;
        }
        validateRequired(s, name);
        if (s.length() < lengthMin) {
            throw new IllegalArgumentException(name + " must be more than "
+ lengthMin + " characters");
        }
        if (lengthMax > 0 && s.length() > lengthMax) {
            throw new IllegalArgumentException(name + " must be less than "
+ lengthMax + " characters");
        }
        return s;
    }

    public static String validateLength(String s, String name, int
lengthMin, int lengthMax) {
        return validateLength(s, name, lengthMin, lengthMax, false);
    }

    public static String validateStringNotBlank(String s, String name) {
        return validateLength(s, name, 1, Integer.MAX_VALUE);
    }

    public static void validateHexColor(String s, String name, boolean
nullable) {
        validateLength(s, name, 7, 7, nullable);
    }

    public static void validateEmail(String s, String name) {
        if (!EMAIL_PATTERN.matcher(s).matches()) {
            throw new IllegalArgumentException(name + " must be an email");
        }
    }

    public static String validateHttpUrl(String s, String name) {
        s = s != null ? s.trim() : null;
        if (s == null || !HTTP_URL_PATTERN.matcher(s).matches()) {
            throw new IllegalArgumentException(name + " must be an HTTP(s)
URL");
        }
    }

```

```

        return s;
    }

    public static void validateAlphanumeric(String s, String name) {
        if (!ALPHANUMERIC_PATTERN.matcher(s).matches()) {
            throw new IllegalArgumentException(name + " must have only
alphanumeric or underscore characters");
        }
    }
}

```

This refactoring separates concerns, simplifies error handling, and makes the code more maintainable and testable. Additionally, it removes unnecessary dependencies and improves error message handling.

Chat 2: After being told that it was a Missing Hierarchy design smell:

To address the missing hierarchy design smell, we can refactor the `ValidationUtil` class to better organize the validation methods into separate classes based on their responsibilities. We can create validator classes for different types of validations, such as string validations, URL validations, email validations, etc. Each validator class can implement a common interface or extend a common abstract class to ensure consistency and ease of use.

Here's how we can refactor the `ValidationUtil` class:

1. Create an interface called `Validator` that defines a method for validation.
2. Create validator classes for string validation, URL validation, email validation, etc., implementing the `Validator` interface.
3. Modify the `ValidationUtil` class to use instances of these validator classes for different types of validations.

Here's a possible refactored version:

```

package com.sismics.rest.util;

import org.codehaus.jettison.json.JSONException;

/**
 * Interface for validators.
 */
public interface Validator<T> {
    void validate(T value, String name) throws JSONException;
}

```

```

package com.sismics.rest.util;

import com.sismics.rest.exception.ClientException;
import org.apache.commons.lang.StringUtils;

```

```

import org.codehaus.jettison.json.JSONException;

import java.text.MessageFormat;
import java.util.regex.Pattern;

/**
 * Validator for string parameters.
 */
public class StringValidator implements Validator<String> {
    private static final Pattern ALPHANUMERIC_PATTERN = Pattern.compile("[a-zA-Z0-9_]+");

    @Override
    public void validate(String value, String name) throws JSONException {
        value = StringUtils.strip(value);
        if (StringUtils.isEmpty(value)) {
            throw new ClientException("ValidationError",
MessageFormat.format("{0} must be set", name));
        }
    }

    public void validateNotBlank(String value, String name) throws
JSONException {
        validate(value, name);
        if (value.trim().isEmpty()) {
            throw new ClientException("ValidationError",
MessageFormat.format("{0} must not be blank", name));
        }
    }

    public void validateAlphanumeric(String value, String name) throws
JSONException {
        validateNotBlank(value, name);
        if (!ALPHANUMERIC_PATTERN.matcher(value).matches()) {
            throw new ClientException("ValidationError",
MessageFormat.format("{0} must have only alphanumeric or underscore
characters", name));
        }
    }
}

```

```

package com.sismics.rest.util;

import com.sismics.rest.exception.ClientException;
import org.codehaus.jettison.json.JSONException;

import java.text.MessageFormat;
import java.util.regex.Pattern;

/**
 * Validator for email addresses.

```



```

*/
public class EmailValidator implements Validator<String> {
    private static final Pattern EMAIL_PATTERN =
Pattern.compile(".*@.*\\.\\..*");

    @Override
    public void validate(String value, String name) throws JSONException {
        if (!EMAIL_PATTERN.matcher(value).matches()) {
            throw new ClientException("ValidationError",
MessageFormat.format("{0} must be an email", name));
        }
    }
}

```

```

package com.sismics.rest.util;

import com.sismics.rest.exception.ClientException;
import org.codehaus.jettison.json.JSONException;

import java.text.MessageFormat;
import java.util.regex.Pattern;

/**
 * Validator for HTTP URLs.
 */
public class HttpUrlValidator implements Validator<String> {
    private static final Pattern HTTP_URL_PATTERN =
Pattern.compile("https?://.*");

    @Override
    public void validate(String value, String name) throws JSONException {
        value = StringUtils.strip(value);
        if (!HTTP_URL_PATTERN.matcher(value).matches()) {
            throw new ClientException("ValidationError",
MessageFormat.format("{0} must be an HTTP(s) URL", name));
        }
    }
}

```

With this refactoring, each validation type has its own validator class, making the code more modular, maintainable, and easier to extend. You can further expand this structure by adding validators for other types of validations as needed.

Here, we see that the LLM was not able to figure out that it was a Missing Hierarchy design smell, and thus was only able to make minor changes for the refactoring - in a completely irrelevant manner.

However, when the issues and the location were highlighted, the LLM performed much better and was able to show successful rectification of the issue by forming a decent hierarchy with no other information.