
Task 1: Document Functionality and Behavior

Book Addition & Display Subsystem

Books Core:

UserBookCriteria.java

(../books-core/src/main/java/com/sismics/books/core/dao/jpa/criteria/UserBookCriteria.java)

1. Facilitates filtering and searching within user book collections based on specified criteria.
2. Allows filtering by `userId` to target specific user's collections.
3. Supports search functionality through a `search` query string.
4. Enables filtering by `read` state to differentiate between read and unread books.
5. Permits tagging based filtering using a list of `tagIdList`, enhancing book organization and retrieval.

BookDao.java

(../books-core/src/main/java/com/sismics/books/core/dao/jpa/BookDao.java)

1. Handles database operations related to `Book` entities, including creation and retrieval.
2. Offers a method to create a new book entry, returning the new book's ID.
3. Provides functionality to fetch a book by its unique ID.
4. Includes a method to retrieve a book using its ISBN number, supporting both ISBN-10 and ISBN-13 formats.

BookImportedEvent.java

(../books-core/src/main/java/com/sismics/books/core/event/BookImportedEvent.java)

1. Represents an event triggered upon a book import request.
2. Captures the user who initiated the book import process.
3. Holds reference to the temporary file used for importing book data.
4. Includes methods to get and set the user and the import file, facilitating event handling.

BookImportAsyncListener.java

(../books-core/src/main/java/com/sismics/books/core/listener/async/BookImportAsyncListener.java)

1. Acts as an asynchronous listener for book import requests, processing the `BookImportedEvent`.
2. Logs the import request event and initiates the import process.
3. Reads book data from a CSV file specified in the import event.
4. Handles book data, including fetching by ISBN, creating new book entries, and user-book associations.
5. Manages tagging for imported books, including creating new tags and associating them with user books.
6. Utilizes transaction management to ensure data integrity throughout the import process.

Book.java

(../books-core/src/main/java/com/sismics/books/core/model/jpa/Book.java)

1. Defines the **Book** entity with detailed attributes including ID, title, subtitle, author, and more.
2. Supports comprehensive book information like ISBN numbers, page count, language, and publication date.
3. Utilizes JPA annotations for database persistence, specifying table and column mappings.
4. Includes getters and setters for all attributes, facilitating easy access and modification of book data.
5. Designed to represent a book in the system with all necessary details for identification and categorization.

UserBook.java

(../books-core/src/main/java/com/sismics/books/core/model/jpa/UserBook.java)

1. Defines the **UserBook** entity linking users to their books, with attributes for tracking reading status and ownership.
2. Contains unique identifiers for both the user and the book, establishing a many-to-many relationship.
3. Includes dates for creation, deletion (optional), and when the book was read, supporting user engagement tracking.
4. Implements **Serializable**, ensuring the entity can be used in serialization operations within Java.
5. Overrides **hashCode** and **equals** methods for entity uniqueness based on **userId** and **bookId**, crucial for database integrity and consistency.

BookDataService.java

(../books-core/src/main/java/com/sismics/books/core/service/BookDataService.java)

1. Provides functionality to fetch book information from external APIs like Google Books and Open Library.
2. Implements asynchronous calls to APIs, respecting rate limits through **RateLimiter**.
3. Supports searching for books by ISBN, sanitizing input to retain only digits.
4. Handles JSON parsing from API responses to construct **Book** entities with detailed information.
5. Includes logic to download and store book thumbnails, ensuring images are in JPEG format.
6. Manages service configuration, including retrieval of the Google API key from system configuration.
7. Utilizes a single-threaded executor for processing API requests, ensuring controlled access to external services.
8. Offers clean-up and shutdown capabilities to properly terminate ongoing tasks and service operations.

UserBookDao.java

(../books-core/src/main/java/com/sismics/books/core/dao/jpa/UserBookDao.java)

1. Facilitates user book management operations, including creation and deletion.
2. Provides methods for retrieving user books by various criteria such as user ID, book ID, etc.
3. Implements functionality for searching user books based on specified criteria.
4. Utilizes the `EntityManager` to interact with the database for CRUD operations.
5. Handles exceptions such as `NoResultException` for cases where no result is found.
6. Supports pagination and sorting of user book search results.
7. Integrates with the `ThreadLocalContext` to obtain the `EntityManager`.
8. Enables fetching user book details in DTO format for presentation purposes.

UserBookDto.java

(../books-core/src/main/java/com/sismics/books/core/dto/UserBookDto.java)

1. Represents a data transfer object (DTO) for user book information.
2. Contains attributes such as ID, title, subtitle, author, language, publication date, creation date, and read date.
3. Utilizes annotations such as `@Id` for identifying the primary key attribute.
4. Provides getter and setter methods for accessing and modifying the attributes.
5. Stores timestamps as `Long` values for easy conversion and manipulation.
6. Enables the transfer of user book details between layers of the application, such as between the DAO and the REST resources.

Books Web:

BaseResource.java

(../books-web/src/main/java/com/sismics/books/rest/resource/BaseResource.java)

1. Serves as the foundation for RESTful resources, providing common functionalities such as authentication and authorization checks.
2. Utilizes `@Context` to inject HTTP request information, enabling access to request details.
3. Supports querying application key directly from query parameters using `@QueryParam`.
4. Manages authentication status by checking for a principal object within the request, distinguishing between anonymous and authenticated users.
5. Implements methods to validate user permissions against specified base functions, throwing exceptions for unauthorized access attempts.
6. Integrates with security filters, specifically `TokenBasedSecurityFilter`, to facilitate secure access control based on tokens.

BookResource.java

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

1. Facilitates book management operations via REST API, including creation, deletion, and updates of books.
2. Enables book searches and imports using public APIs and file uploads.
3. Supports manual entry and editing of book details, such as title, author, and ISBN numbers.

4. Provides functionality for listing user books with filters for search, read status, and tags.
5. Allows for setting books as read/unread and managing book tags for categorization.
6. Implements file handling for importing books in bulk from a CSV or similar format.
7. Integrates with external services for book information retrieval and cover image downloading.
8. Ensures user authentication and authorization for book-related actions, leveraging base class security methods.

Bookshelf Management Subsystem

Books-Core:

TagDto

(books-core/src/main/java/com/sismics/books/core/dao/jpa/dto/TagDto.java):

1. Represents tag information crucial for efficient book organization within the Bookshelf Management Subsystem.
2. Attributes (`id`, `name`, `color`) ensure each tag is uniquely identified, has a recognizable name, and a distinct color for user-friendly book categorization.
3. It interacts with the database for efficient storage and retrieval using methods like `getId()`, `setId()`, `getName()`, `setName()`, `getColor()`, `setColor()`.
4. Methods facilitate easy access and modification of tag information, ensuring reliable functionality within the Bookshelf Management Subsystem
5. Enhances user experience by simplifying navigation and recognition of tags, providing an intuitive way to organize and categorize books within the bookshelf.

UserBookDto

(UserBookDto - `com.sismics.books.core.dao.jpa.dto.UserBookDto`):

1. Represents user book information, essential for tracking individual book details within the Bookshelf Management Subsystem.
2. Attributes (`id`, `title`, `subtitle`, `author`, `language`, `publishTimestamp`, `createTimestamp`, `readTimestamp`) capture key information such as book title, author, language, and timestamps for publication, creation, and reading.
3. Interacts with the database through methods like `getId()`, `setTitle()`, `setAuthor()`, facilitating efficient storage and retrieval of user book details.
4. Supports getter and setter methods for attributes, ensuring easy access and modification of user book information, contributing to the reliability of the Bookshelf Management Subsystem.
5. Facilitates user experience by providing detailed book information, including title, author, and timestamps, enhancing the book organization process within the bookshelf.

TagDao

(TagDao - `com.sismics.books.core.dao.jpa.TagDao`):

1. Manages tag data access and operations within the Bookshelf Management Subsystem, offering methods for interacting with the database and performing tag-related functionalities.
2. Includes methods like `getById(String id)` and `getByUserId(String userId)` to retrieve tag details by ID or user ID, ensuring efficient data retrieval for book organization.
3. Implements `updateTagList(String userBookId, Set<String> tagIdSet)` to manage the association between user books and tags. This method efficiently updates tag links, enhancing the Bookshelf Management Subsystem's functionality.

4. Facilitates the retrieval of tag information linked to a user book through `getByUserBookId(String userBookId)`, returning a list of `TagDto` objects containing tag details.
5. Provides functionalities for tag creation (`create(Tag tag)`), deletion (`delete(String tagId)`), and searching by name (`findByName(String userId, String name)`), contributing to a comprehensive tag management system within the Bookshelf Management Subsystem.

UserBookDao

(UserBookDao - com.sismics.books.core.dao.jpa.UserBookDao):

1. Manages user book data access and operations within the Bookshelf Management Subsystem, providing methods for creating, deleting, and searching user books based on various criteria.
2. Implements `create(UserBook userBook)` to generate a new user book entry, assigning a unique ID using UUID and storing the user book in the database.
3. Supports `delete(String id)` for logically deleting a user book by marking its deletion date, ensuring that historical data is maintained.
4. Includes methods like `getUserBook(String userBookId, String userId)` and `getUserBook(String userBookId)` to retrieve user book details by ID or book ID, respectively, considering the deletion status.
5. Enables searching and retrieval of user books based on criteria, such as title, subtitle, author, language, publish date, creation date, and read date. This is achieved through `findByCriteria(PaginatedList<UserBookDto> paginatedList, UserBookCriteria criteria, SortCriteria sortCriteria)`.
6. Utilizes efficient database queries to assemble paginated lists of `UserBookDto` objects, providing a comprehensive view of user book information for the Bookshelf Management Subsystem.

Tag

(Tag - com.sismics.books.core.model.jpa.Tag):

1. Represents a tag entity within the Bookshelf Management Subsystem, encapsulating information such as Tag ID (`id`), Tag Name (`name`), User ID (`userId`), Creation Date (`createDate`), Deletion Date (`deleteDate`), and Tag Color (`color`).
2. Utilizes JPA annotations (`@Entity`, `@Table`, `@Id`, `@Column`) for defining the entity and its attributes in the database.
3. Ensures data integrity with constraints such as non-nullable fields (`name`, `userId`, `createDate`, `color`), enforcing essential information for reliable tag management.
4. Supports getter and setter methods for accessing and modifying tag attributes, allowing seamless integration with the Bookshelf Management Subsystem.
5. Implements `toString()` for generating a concise and informative representation of the tag, particularly useful for logging and debugging purposes.

LogCriteria

(LogCriteria - com.sismics.util.log4j.LogCriteria):

1. Represents the criteria used for searching logs within the logging system.

2. Contains attributes such as `level` (logging level like DEBUG, WARN), `tag` (logger name or tag), and `message` (logged message).
3. Utilizes the Apache Commons Lang library (`StringUtils`) to ensure consistent case handling for the `level`, `tag`, and `message` attributes.
4. Provides getter and setter methods for each attribute, allowing external components to retrieve and modify the search criteria.
5. Enhances flexibility by allowing case-insensitive comparison of log criteria, improving the accuracy and comprehensiveness of log searches within the logging system.

LogEntry

(LogEntry - com.sismics.util.log4j.LogEntry):

1. Represents a log entry within the logging system, capturing essential information about a logged event.
2. Includes attributes such as `timestamp` (time when the log entry was recorded), `level` (logging level like DEBUG, WARN), `tag` (logger name or tag), and `message` (logged message).
3. Provides a constructor that allows initializing the log entry with specific values for timestamp, level, tag, and message.
4. Offers getter methods for each attribute, enabling external components to retrieve information about the log entry.
5. Facilitates the analysis and processing of log data by encapsulating relevant details of each logged event in a structured format.

BaseResource

(BaseResource - com.sismics.books.rest.resource.BaseResource):

1. Serves as the base class for REST resources in the application.
2. Includes annotations to inject the HTTP request (`@Context`), allowing access to request-related information.
3. Contains a query parameter `appKey` representing the application key.
4. Maintains a `principal` attribute representing the authenticated user principal.
5. Provides a method `authenticate()` to check if the user is authenticated and not anonymous.
6. Implements a method `checkBaseFunction(BaseFunction baseFunction)` to verify if the user has a specific base function, throwing a `ForbiddenClientException` if the check fails.
7. Defines a method `hasBaseFunction(BaseFunction baseFunction)` to determine if the user has a specific base function, returning true if the user has the required base function.
8. Ensures that the class is designed for extensibility and serves as a foundational component for other REST resources in the application.

TagResource

(TagResource - com.sismics.books.rest.resource.TagResource):

1. Represents a REST resource for handling operations related to tags.
2. Includes methods for retrieving, creating, updating, and deleting tags.
3. Uses JAX-RS annotations such as `@Path`, `@GET`, `@PUT`, `@POST`, `@DELETE`, and `@Produces` to define the resource's URI, HTTP methods, and media type.

4. Extends `BaseResource`, indicating that it serves as a subclass of the base REST resource.
5. The `list()` method retrieves the list of tags for the authenticated user.
6. The `add()` method creates a new tag with the specified name and color.
7. The `update()` method updates an existing tag with the specified ID, allowing changes to the name and color.
8. The `delete()` method deletes an existing tag with the specified ID.
9. Utilizes the `TagDao` class to interact with the database for tag-related operations.
10. Includes input validation using `ValidationUtil` for parameters such as name and color.
11. Throws `ForbiddenClientException` for unauthorized access and `ClientException` for various client-related errors.
12. Returns JSON responses for successful operations.
13. Demonstrates good practices for handling RESTful resources and their CRUD operations.
14. Provides a RESTful API for managing tags in the application.

UserBookCriteria

(UserBookCriteria - com.sismics.books.core.dao.jpa.criteria.UserBookCriteria):

1. Represents criteria for searching user books.
2. Contains attributes such as `userId`, `search`, `read`, and `tagIdList` to define search parameters.
3. Provides getter and setter methods for each attribute.
4. The `userId` attribute represents the user ID for which the search is performed.
5. The `search` attribute holds the search query for filtering user books based on titles, subtitles, or authors.
6. The `read` attribute indicates the read state of the user books.
7. The `tagIdList` attribute is a list of tag IDs used to filter user books based on associated tags.
8. Encapsulates search criteria for querying user books in the application.
9. Designed to be used in conjunction with the `UserBookDao` class for searching user books based on specified criteria.

User Management Subsystem:

books-core

Constants

(books-core/src/main/java/com/sismics/books/core/constant/Constants.java)

1. This class is used for storing application-wide constants that are used across different parts of the application.
2. Defines constants for default locale, timezone, theme ID, administrator's default password, and default generic user role.
3. Provides a convenient way to access these constants throughout the application without hardcoding their values.

AuthenticationToken (Model JPA)

(books-core/src/main/java/com/sismics/books/core/model/jpa/AuthenticationToken.java)

1. The AuthenticationToken class is a JPA entity representing an authentication token stored in the database
2. Used in to manage user sessions and permissions.
3. Stores essential details about authentication tokens:
 - **id**: Token ID, primary key.
 - **userId**: ID of the associated user.
 - **longLasted**: Indicates if the token represents a long-lasting session.
 - **creationDate**: Date when the token was created.
 - **lastConnectionDate**: Date of the last connection using this token.

BaseFunction (Model JPA)

(books-core/src/main/java/com/sismics/books/core/model/jpa/BaseFunction.java)

1. JPA entity representing information about base functions.
2. Represent fundamental capabilities or permissions within an application.
3. associated with **Role** to define roles and their associated base functions.

Role (Model JPA)

(books-core/src/main/java/com/sismics/books/core/model/jpa/Role.java)

1. JPA entity representing information about roles.
2. Captures essential data related to roles, including their identifiers, names, and creation/deletion dates.
3. Allows for the management and tracking of roles throughout their lifecycle, including creation and deletion.

RoleBaseFunction (Model JPA)

(books-core/src/main/java/com/sismics/books/core/model/jpa/RoleBaseFunction.java)

1. JPA entity representing associations between roles and base functions.
2. Role base functions represent the permissions or capabilities associated with specific roles within the application.
3. Facilitates the management of role-permission relationships within the application's security model.
4. Enables the assignment of specific permissions (base functions) to roles, defining the access rights of users associated with those roles.
5. Supports the association of multiple base functions with a single role, allowing for flexible and granular access control.
6. Tracks the creation and deletion dates of role base function associations, providing insight into their lifecycle and history.

User (Model JPA)

(books-core/src/main/java/com/sismics/books/core/model/jpa/User.java)

1. JPA entity representing individual users within the application, including their login credentials, profile settings, and status flags.
2. Stores the following information about a user:
 - **id**: Unique identifier for the user.
 - **localeId**: ID of the user's preferred locale.
 - **roleId**: ID of the role associated with the user.
 - **username**: Username used for authentication.
 - **password**: Password used for authentication (typically stored securely hashed).
 - **email**: Email address associated with the user.
 - **theme**: Theme preference for the user interface.
 - **firstConnection**: Flag indicating whether the user has completed the first connection process.
 - **createDate**: Date when the user account was created.
 - **deleteDate**: Date when the user account was deleted (if applicable).
3. Supports the storage of user-specific settings and preferences, such as language and theme preferences.
4. Enables user-related actions such as authentication, user profile management, and account lifecycle management.

UserApp (Model JPA)

(books-core/src/main/java/com/sismics/books/core/model/jpa/UserApp.java)

1. JPA entity used for Managing the association between users and connected applications, including OAuth tokens and user-specific data within each application.
2. Represents the link between a user account and an external application or service, such as a social media platform or third-party integration.
3. Stores the following information about a user's connection to an application:
 - **id**: Unique identifier for the user-app association.

- **userId**: ID of the user associated with the application.
 - **appId**: ID of the connected application.
 - **accessToken**: OAuth access token obtained during authentication with the application.
 - **username**: User's username within the connected application (if applicable).
 - **externalId**: User's ID within the connected application (if applicable).
 - **email**: User's email address associated with the connected application (if applicable).
 - **sharing**: Flag indicating whether the user has enabled sharing data with the connected application.
 - **createDate**: Date when the user-app association was created.
 - **deleteDate**: Date when the user-app association was deleted (if applicable).
4. Facilitates the integration of third-party services or applications with the user management system.
 5. Supports the storage of OAuth tokens and user-specific data required for interacting with external applications.
 6. Enables features such as single sign-on (SSO) and data sharing between the application and connected services.

UserContact (Model JPA)

(books-core/src/main/java/com/sismics/books/core/model/jpa/UserContact.java)

1. JPA entity used for Managing the contact information associated with a user in external applications or services.
2. Represents the linkage between a user account and a contact record in a connected application, such as a social media platform or external CRM system.
3. Stores the following information about a user's contact in a connected application:
 - **id**: Unique identifier for the user contact record.
 - **userId**: ID of the user associated with the contact.
 - **appId**: ID of the connected application.
 - **externalId**: ID of the contact within the connected application.
 - **fullName**: Full name of the contact in the connected application (if available).
 - **email**: Email address of the contact in the connected application (if available).
 - **createDate**: Date when the user contact record was created.
 - **updateDate**: Date when the user contact record was last updated.
 - **deleteDate**: Date when the user contact record was deleted (if applicable).

Locale (Model JPA)

(books-core/src/main/java/com/sismics/books/core/model/jpa/Locale.java)

1. JPA entity representing locale information within the application, including unique identifiers for different locales.
2. Stores the following information about a locale:
 - **id**: Unique identifier for the locale (e.g., "fr_FR").
3. Enables the storage and retrieval of locale-specific data and settings.
4. Facilitates internationalization and localization efforts within the application.

UserAppCreatedEvent

(books-core/src/main/java/com/sismics/books/core/event/UserAppCreatedEvent.java)

1. Represents an event indicating the creation of a user app.
2. event used for Broadcasting notifications about the creation of a user app within the system.
3. Enables loose coupling between components by allowing them to communicate through events rather than direct method calls.

UserAppDao

(books-core/src/main/java/com/sismics/books/core/dao/jpa/UserAppDao.java)

1. Used as a Data Access Object (DAO) for managing database operations related to user applications in the system.
2. Utilizes JPA Query API and native SQL queries for custom retrieval and manipulation of user app data.
3. Utilizes Data Transfer Objects (DTOs) such as UserAppDto to map database query results to Java objects for easier consumption.
4. Follows the Data Access Object (DAO) design pattern to separate the data access logic from the business logic, promoting modularity and maintainability.

RoleBaseFunctionDao

(books-core/src/main/java/com/sismics/books/core/dao/jpa/RoleBaseFunctionDao.java)

1. Used as a Data Access Object (DAO) for managing database operations related to base functions associated with a role.
2. Utilizes JPA Query API and native SQL queries for custom retrieval and manipulation of role-baseFunction data.

LocaleDao

(books-core/src/main/java/com/sismics/books/core/dao/jpa/LocaleDao.java)

1. Used as a Data Access Object (DAO) for managing database operations related to locales in the system.
2. Utilizes JPA Query API and native SQL queries for custom retrieval and manipulation of Locale data.

UserContactDao

(books-core/src/main/java/com/sismics/books/core/dao/jpa/UserContactDao.java)

1. Used as a Data Access Object (DAO) for managing database operations related to user contacts in the system.
2. Responsible for creating, reading, updating, and deleting user contacts in the database.
3. Additionally, it provides methods for searching user contacts based on various criteria.
4. Utilizes Data Transfer Objects (DTOs) such as UserContactDto to map database query results to Java objects for easier consumption.
5. Implements pagination for search results using PaginatedList to efficiently handle large datasets and improve performance.

6. Follows the Data Access Object (DAO) design pattern to separate the data access logic from the business logic, promoting modularity and maintainability.

UserDao

(books-core/src/main/java/com/sismics/books/core/dao/jpa/UserDao.java)

1. Used as a Data Access Object (DAO) for managing database operations related to users in the system.
2. Responsible for user authentication, creation, updating, deletion, and retrieval operations.
3. Provides methods for handling user passwords, including hashing and password reset.
4. Utilizes BCrypt for hashing passwords securely.
5. Implements pagination for search results using PaginatedList to efficiently handle large datasets and improve performance.
6. Follows the Data Access Object (DAO) design pattern to separate the data access logic from the business logic, promoting modularity and maintainability.

AuthenticationTokenDao

(books-core/src/main/java/com/sismics/books/core/dao/jpa/AuthenticationTokenDao.java)

1. Used as a Data Access Object (DAO) for managing authentication tokens in the system.
2. Handles operations related to creating, retrieving, updating, and deleting authentication tokens.
3. Provides methods to manage the lifecycle of authentication tokens, including deletion of old tokens and updating last connection dates.
4. Follows the Data Access Object (DAO) design pattern to separate the data access logic from the business logic, promoting modularity and maintainability.

UserContactCriteria

(books-core/src/main/java/com/sismics/books/core/dao/jpa/criteria/UserContactCriteria.java)

1. Used as a criteria object for searching user contacts based on specific parameters.
2. Provides attributes to specify the application ID, user ID, and a full-text query for filtering contacts.
3. Primarily used for passing search criteria to methods that perform user contact searches.
4. Designed to be a lightweight container for holding search criteria, promoting encapsulation and reusability in search operations.

UserAppDto

(books-core/src/main/java/com/sismics/books/core/dao/jpa/dto/UserAppDto.java)

1. Used as a Data Transfer Object (DTO) for transferring user application-related information between layers of the application.
2. Represents essential attributes of a user application, including identifiers, access token, username, and sharing status.
3. Designed to encapsulate user application data for ease of transfer and manipulation.
4. Supports boolean attribute sharing to indicate whether the user application shares traces, providing flexibility in representing application settings or preferences.

UserContactDto

(books-core/src/main/java/com/sismics/books/core/dao/jpa/dto/UserContactDto.java)

1. Used as a Data Transfer Object (DTO) for transferring user contact-related information between layers of the application.
2. Represents essential attributes of a user contact, including identifiers, external contact ID, and full name.
3. Designed to encapsulate user contact data for ease of transfer and manipulation.

UserDto

(books-core/src/main/java/com/sismics/books/core/dao/jpa/dto/UserDto.java)

1. Used as a Data Transfer Object (DTO) for transferring user-related information between layers of the application.
2. Represents essential attributes of a user, such as identifiers, locale, username, email, and creation timestamp.
3. Designed to encapsulate user data for ease of transfer and manipulation.

UserAppCreatedAsyncListener.java

(../books-core/src/main/java/com/sismics/books/core/listener/async/UserAppCreatedAsyncListener.java)

1. Listens for `UserAppCreatedEvent` and logs the event upon occurrence.
2. Retrieves the `UserApp` instance from the event to identify the app and user involved.
3. Executes contact synchronization for the specified app, currently supporting Facebook through `FacebookService`.
4. Utilizes a switch case on `AppId` to handle different app integrations, indicating potential for future expansion.
5. Ensures the operation is wrapped in a transaction to maintain data consistency and integrity.

books-web

BaseFunction

(books-web/src/main/java/com/sismics/books/rest/constant/BaseFunction.java)

1. Enum representing base functions within the application, defining specific functionalities that users can perform.
2. Provides a single base function:
 - `ADMIN`: Allows the user to access and utilize administrative functions.
3. Enumerates essential functionalities that users can execute within the system.
4. Used for role-based access control and permission management within the application.

BaseResource

(books-web/src/main/java/com/sismics/books/rest/resource/BaseResource.java):

1. This class serves as a abstract base class for REST resources in a web application
2. Provides common functionality for REST resources, such as handling HTTP requests, extracting query parameters, and managing user authentication.
3. Implements methods for authentication (`authenticate()`) and authorization (`checkBaseFunction()` and `hasBaseFunction()`).
4. Relies on the `HttpServletRequest` to access HTTP request details.
5. Utilizes query parameters, specifically the `app_key` parameter, for application-level authentication or identification.
6. Encapsulates the Principal of the authenticated user, allowing subclasses to access user-related information easily.
7. The class is designed to be extended by specific resource classes, providing them with common functionalities and enforcing security measures consistently across the application's REST endpoints.

UserResource

(books-web/src/main/java/com/sismics/books/rest/resource/UserResource.java):

1. This class is used as a **REST resource** for user-related operations in a web application.
2. Extends the `BaseResource` class, inheriting common functionalities such as authentication and authorization checks.
3. Implements various methods to handle HTTP requests for different **user operations** such as registration, login, logout, updating user details, checking username availability, and managing user sessions.
4. Utilizes query parameters, form parameters, and path parameters to receive input data for user operations.
5. Communicates with the data access layer (**DAO**) to interact with the underlying database, performing operations such as user creation, retrieval, update, and deletion.
6. Utilizes JSON for request and response data serialization/deserialization

LocaleResource

(books-web/src/main/java/com/sismics/books/rest/resource/LocaleResource.java)

1. REST resource handling locale-related operations, such as retrieving the list of all available locales.
2. Defines a resource endpoint at `/locale` for handling locale-related requests.
3. Provides a method `list()` to retrieve the list of all locales in JSON format.
4. Utilizes `LocaleDao` to fetch the list of locales from the database.
5. Returns a response with the JSON representation of the locales list.

books-web-common

IPrincipal

(books-web-common/src/main/java/com/sismics/security/IPrincipal.java)

1. Extends the `Principal` interface, which is a standard Java interface representing the identity of a principal (e.g., user, group, service).

2. This interface defines the contract for principals in the system. Principals typically represent authenticated users or entities with certain permissions within the application.
3. Encapsulates user-related information, such as `ID`, `locale`, `timezone`, and `email`, within the principal abstraction, promoting encapsulation and separation of concerns.
4. Serves as a way to abstract the details of user authentication and authorization, allowing different implementations (e.g., `user principals`, `service principals`) to conform to a common interface.

AnonymousPrincipal

(books-web-common/src/main/java/com/sismics/security/AnonymousPrincipal.java)

1. This class represents an anonymous principal, typically used to represent unauthenticated users or entities lacking specific identity within the application.
2. Implements the `IPrincipal` interface, providing necessary methods to adhere to the contract defined by the interface.
3. Represents a foundational component in handling anonymous access or actions within the application, serving as a placeholder for unauthenticated users or entities with limited identity information.

UserPrincipal

(books-web-common/src/main/java/com/sismics/security/UserPrincipal.java)

1. This class represents authenticated users within the system, serving as the principal entity associated with user authentication and authorization.
2. Implements the `IPrincipal` interface, providing necessary methods to adhere to the contract defined by the interface.
3. Represents a key component in user authentication and authorization, serving as the principal entity associated with authenticated user sessions.
4. Contains a base function set to represent the set of base functions associated with the authenticated user, facilitating access control and permissions management within the system.

TokenBasedSecurityFilter

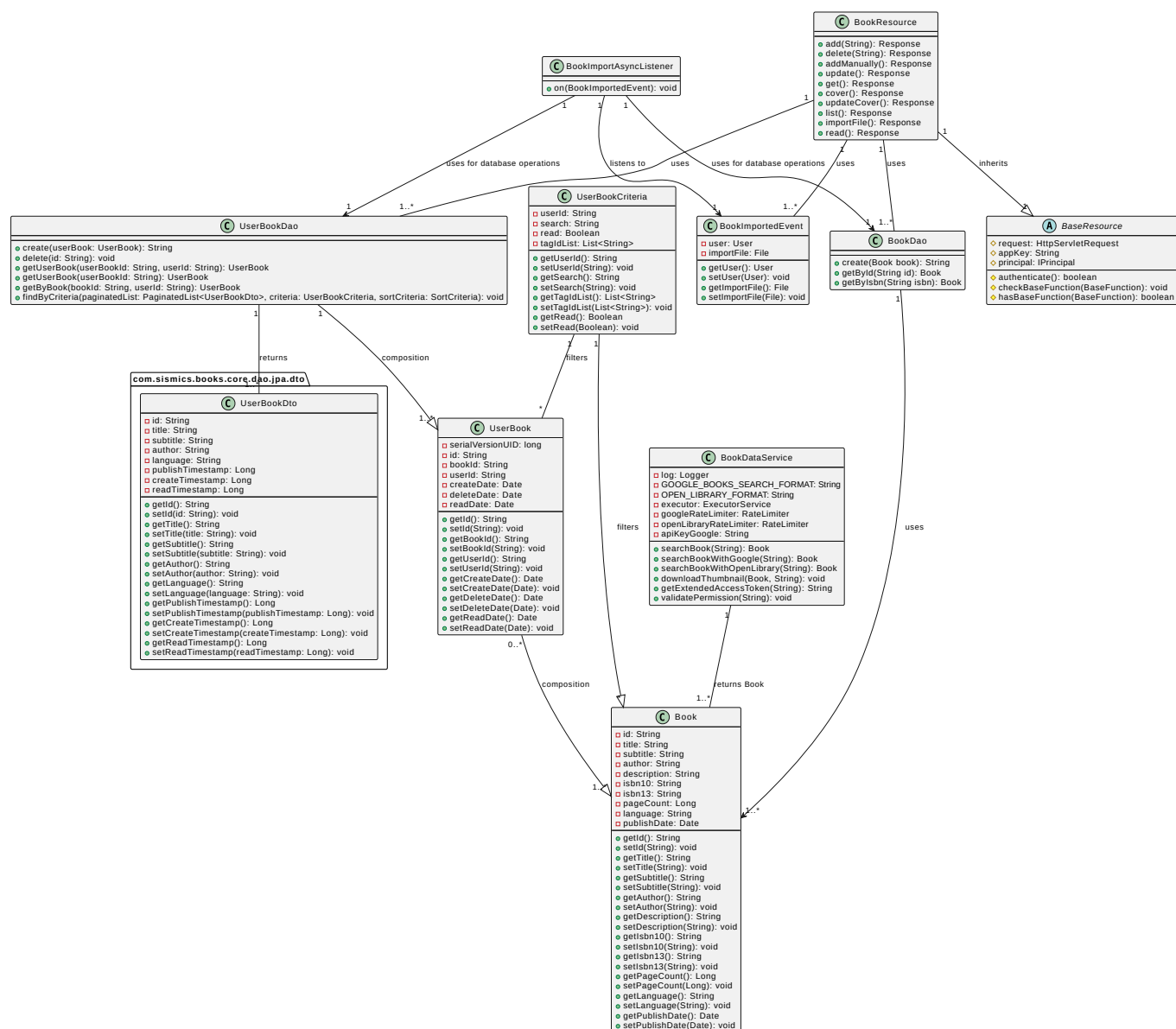
(books-web-common/src/main/java/com/sismics/util/filter/TokenBasedSecurityFilter.java)

1. This class is a filter used for authenticating users based on an authentication token stored in a cookie.
2. It is responsible for extracting the authentication token from the request cookie, validating its authenticity, and injecting the corresponding user principal into the request attributes.
3. Provides logic to extract the authentication token from the request cookie and retrieve the corresponding server token from the database.
4. Checks the validity and expiration status of the authentication token, determining whether the user is authenticated or anonymous.
5. Injects an authenticated user principal or an anonymous user principal into the request attributes based on the authentication status.
6. Utilizes DAO classes (`AuthenticationTokenDao`, `UserDao`, `RoleBaseFunctionDao`) for database interactions to retrieve authentication token information and user data.

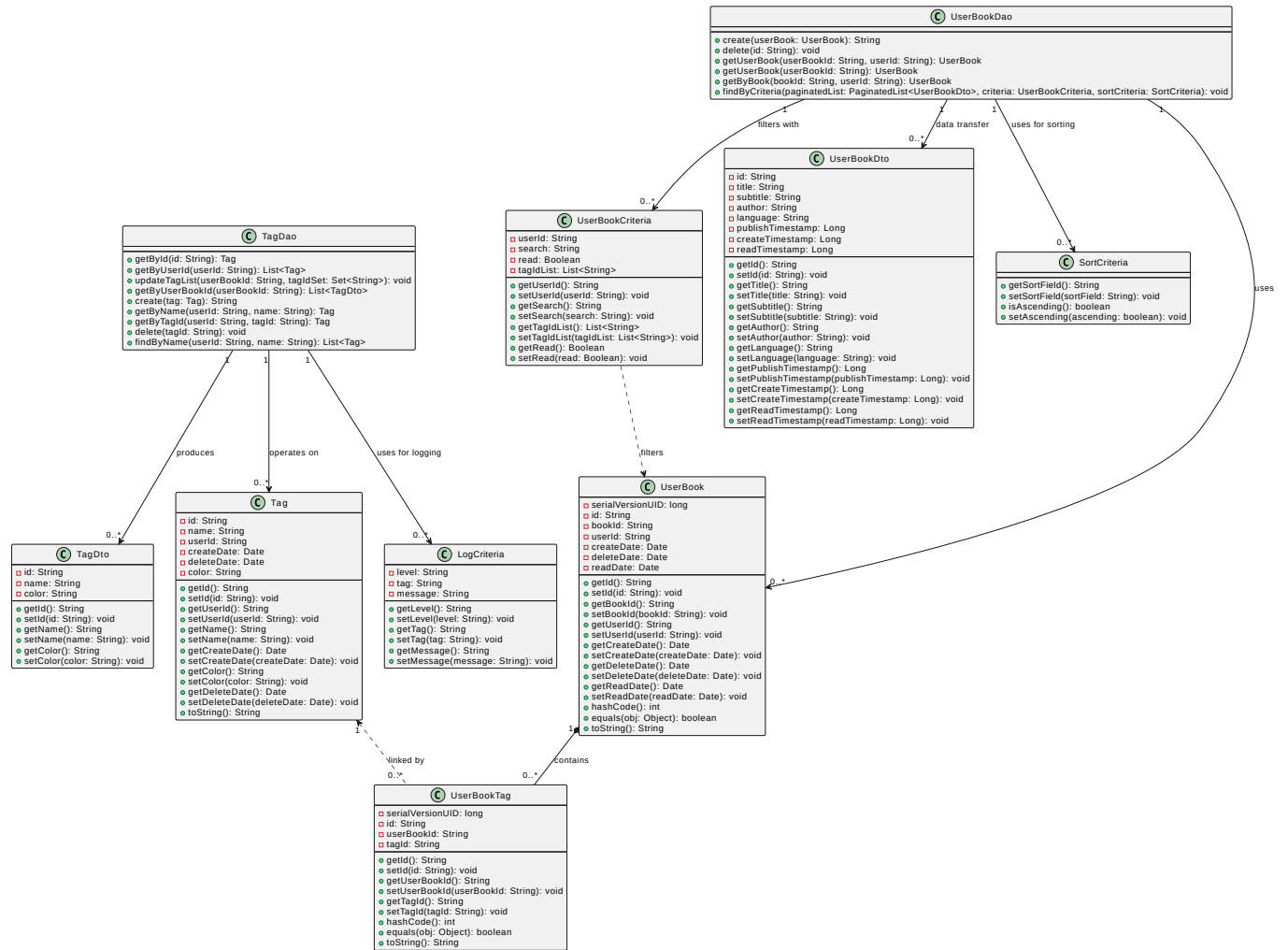
7. Serves as a critical component for user authentication and authorization within the application, implementing token-based security mechanisms.

Task 1: Create UML Diagrams

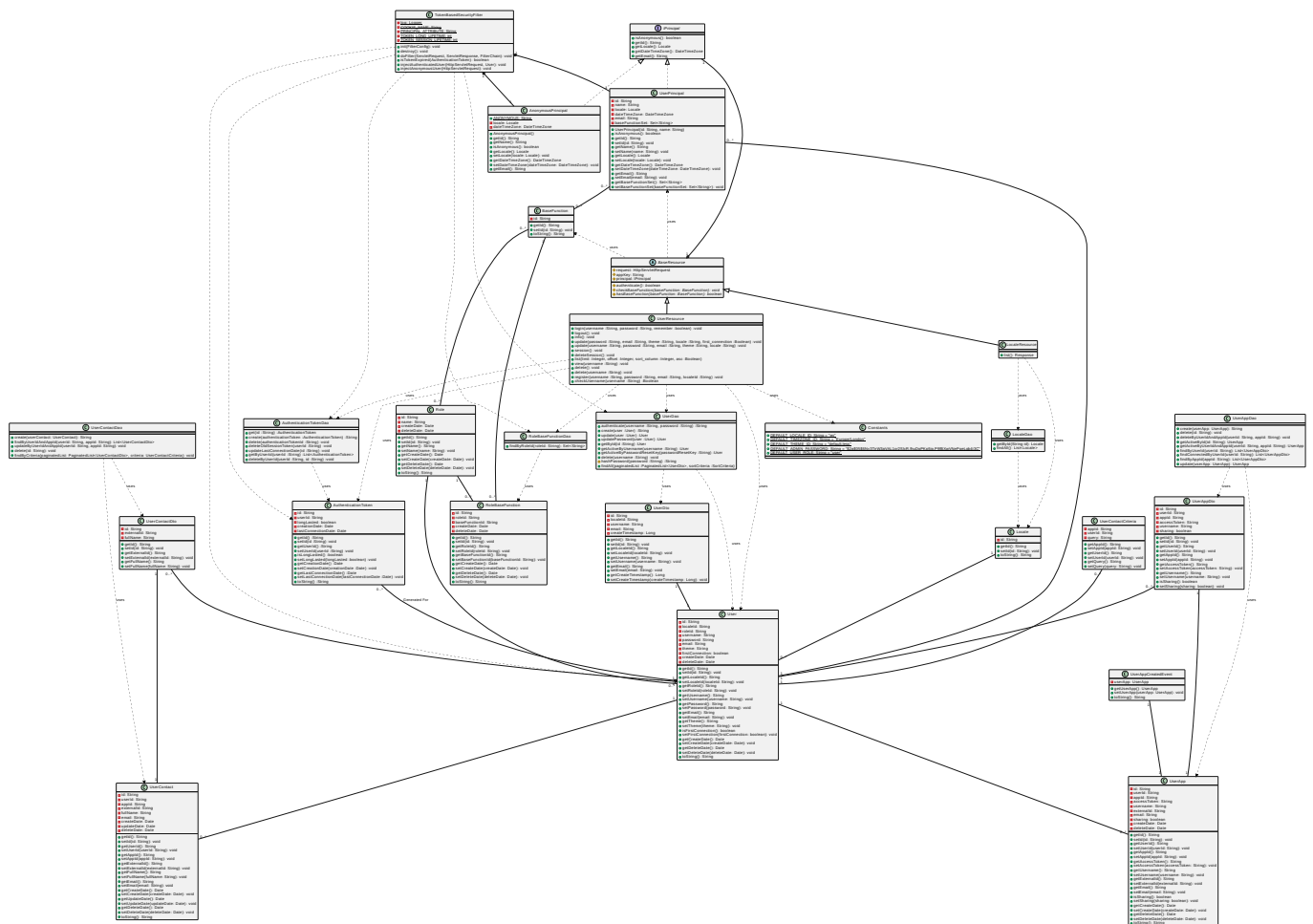
Book Addition & Display Subsystem



Bookshelf Management Subsystem



User Management System



Task 1: Observations and Comments

Book Management Subsystem

Observations:

Strengths:

- **Well-structured classes:** The classes like `UserBookDto`, `TagDto`, and `UserBookCriteria` are well-organized and contain necessary attributes and methods.
- **Clear data transfer objects:** `UserBookDto` and `TagDto` serve as clear DTOs (Data Transfer Objects) for exchanging data between layers.
- **Utilization of DAO pattern:** The DAO classes like `UserBookDao` and `TagDao` adhere to the DAO pattern for handling database operations, promoting separation of concerns.

Weaknesses:

- **Lack of documentation:** There's no inline documentation or comments explaining the purpose or usage of each class or method, which could make it harder for developers to understand the codebase.
- **Incomplete relationships:** The relationships between classes are defined, but some of the associations lack cardinality or multiplicity indicators, making it unclear whether they are one-to-one or one-to-many relationships.
- **No error handling:** Error handling mechanisms are not apparent in the provided code, which could lead to potential issues during runtime if exceptions are not handled appropriately.

Comments:

- Overall, the subsystem structure appears robust, with clear delineation of responsibilities among classes.
- Adding inline documentation and error handling mechanisms would enhance the maintainability and robustness of the subsystem.
- Clarifying the multiplicity of associations would improve the understanding of the data model and its interactions.

Book Addition Display Subsystem

Observations:

Strengths:

- **Comprehensive class coverage:** The subsystem includes classes for various functionalities such as book addition, deletion, searching, and import handling.
- **Utilization of design patterns:** The subsystem seems to employ design patterns like DAO (Data Access Object) pattern for database operations, enhancing modularity and maintainability.
- **Proper separation of concerns:** Classes like `BookResource` and `BookDao` handle distinct responsibilities, contributing to a clear separation of concerns.

Weaknesses:

- **Lack of clarity in relationships:** Similar to the previous subsystem, some relationships between classes lack clarity in terms of cardinality or multiplicity, which could lead to ambiguity.
- **Potential performance issues:** The code does not address potential performance concerns such as asynchronous processing for resource-intensive operations like book import.
- **Limited error handling:** The code appears to lack robust error handling mechanisms, which could result in unexpected behavior or vulnerabilities.

Comments:

- The subsystem provides essential functionalities for book management, but it could benefit from enhancements in documentation, performance optimization, and error handling.
- Incorporating asynchronous processing for tasks like book import could improve system responsiveness and scalability.
- Further clarification on the relationships between classes would aid in understanding the subsystem's architecture.

User Management Subsystem

Observations:

Strengths:

- **Comprehensive functionality coverage:** The subsystem encompasses functionalities such as user authentication, registration, session management, and role-based access control.
- **Clear separation of concerns:** Classes like `UserResource`, `UserDao`, and `TokenBasedSecurityFilter` handle distinct aspects of user management, promoting modularity.
- **Usage of design patterns:** The subsystem utilizes patterns like DAO pattern for data access and Filter pattern for security implementations, enhancing maintainability.

Weaknesses:

- **Lack of detailed documentation:** While the subsystem structure is well-defined, the absence of inline documentation or comments may hinder code understanding and maintenance.
- **Potential security vulnerabilities:** Without detailed inspection, it's challenging to assess the robustness of security mechanisms like authentication and session management.
- **Complexity in role management:** The role management mechanism involving `Role`, `RoleBaseFunction`, and `BaseFunction` may introduce complexity, especially in larger systems.

Comments:

- The subsystem provides a comprehensive set of functionalities for user management, covering authentication, authorization, and user data management.
- Improving documentation and adding comments would facilitate code comprehension and maintenance.
- Conducting thorough security assessments and potentially incorporating security best practices would be crucial for ensuring system integrity and user data protection.

Task 2a

1. Lazy Class & Deficient Encapsulation: Constants (com.sismics.books.core.constant)

Overview

The Constants class provides a set of static final fields representing various default values and identifiers used throughout the application, such as default locale, timezone, theme, administrator's default password, and default user role.

Analysis

1. **Limited Functionality**: The Constants class primarily consists of static final fields with default values and identifiers. It does not contain any methods or behavior beyond initializing these constant values.
2. **Minimal Complexity**: Due to its simplistic nature, the Constants class has minimal complexity. It merely serves as a repository for storing constant values, which are accessed throughout the application.
3. **Lack of Responsiveness**: While the class encapsulates related constants, it does not actively contribute to the behavior or functionality of the application. It remains inert and does not participate in any computational tasks.

Recommendations

1. **Eliminate Unnecessary Abstraction**: Since the constant values are only used sporadically or by a limited number of classes, inline them directly where needed to reduce the overhead of maintaining a separate class.

Conclusion

The Constants class exhibits the "Lazy Class" design smell due to its limited functionality and lack of responsiveness. While it serves the purpose of centralizing constant values, its existence may not be justified if it merely holds values that are rarely accessed or updated. Evaluating the necessity of the Constants class and potentially integrating its constants directly into relevant classes can streamline the codebase and enhance maintainability.

2. God Class: UserResource (books-web.com.sismics.books.rest.resource)

Overview

The UserResource class serves as a RESTful resource for handling various user-related operations, including user registration, updating user information, user authentication, session management, user deletion, and user information retrieval.

```
class UserResource extends BaseResource {
    + login(username :String, password :String, remember :boolean) :void
    + logout() :void
    + info() :void
    + update(password :String, email :String, theme :String, locale
:String, first_connection :Boolean) :void
    + update(username :String, password :String, email :String, theme
:String, locale :String) :void
    + session() :void
    + deleteSession() :void
    + list(limit :Integer, offset :Integer, sort_column :Integer, asc
:Boolean)
    + view(username :String) :void
    + delete() :void
    + delete(username :String) :void
    + register(username :String, password :String, email :String, localeId
:String) :void
    + checkUsername(username :String) :Boolean
}
```

Analysis

1. **Multipurpose Resource Class:** The UserResource class combines multiple functionalities related to user management within a single class. It handles user registration (PUT method), updating user information (POST methods), user authentication (POST method), user session management (GET and DELETE methods), user deletion (DELETE methods), and user information retrieval (GET methods).
2. **Violation of Single Responsibility Principle (SRP):** The class violates the SRP by encompassing too many responsibilities. Each method within the class deals with different aspects of user management, leading to a lack of cohesion and increased complexity.
3. **Complexity and Coupling:** Including diverse functionalities within the same class increases complexity and tight coupling between different parts of the code. This makes it difficult to modify or extend the functionality without affecting other parts of the system.

Recommendations

1. **Refactor into Coherent Components:** Separate the functionalities of the UserResource class into smaller, more cohesive components, each responsible for a specific aspect of user management. For example, create separate resource classes for user registration, user authentication, session management, etc.

2. **Use Composition over Inheritance**: Instead of handling all functionalities within the UserResource class hierarchy, use composition to encapsulate behavior in separate classes that can be easily composed together.
3. **Adhere to RESTful Principles**: Follow RESTful design principles, such as organizing resources around entities and using HTTP methods to represent CRUD operations. Each resource class should focus on a specific entity or a closely related set of functionalities.

Conclusion

The "God Class" design smell in the UserResource class indicates a need for refactoring to improve maintainability, modularity, and adherence to design principles. By restructuring the code into smaller, focused components and adhering to RESTful principles, the codebase can become more manageable and easier to maintain.

3. God Class: BookResource (books-web.com.sismics.books.rest.resource)

Overview

The **BookResource** class acts as a RESTful endpoint for managing a broad spectrum of book-related operations within the application.

Code Snippet

```
class BookResource extends BaseResource {
    +add(String): Response
    +delete(String): Response
    +addManually(): Response
    +update(): Response
    +get(): Response
    +cover(): Response
    +updateCover(): Response
    +list(): Response
    +importFile(): Response
    +read(): Response
}
```

SonarQube identified Code smell

books-web/.../sismics/books/rest/resource/BookResource.java

<input type="checkbox"/>	Refactor this method to reduce its Cognitive Complexity from 22 to the 15 allowed.	26 days ago ▾ L162 🔗 ⚙️ ▾
	🔗 Code Smell ▾ 🚨 Critical ▾ 🔵 Open ▾ Not assigned ▾ 12min effort Comment	🧠 brain-overload ▾
<input type="checkbox"/>	Refactor this method to reduce its Cognitive Complexity from 27 to the 15 allowed.	26 days ago ▾ L276 🔗 ⚙️ ▾
	🔗 Code Smell ▾ 🚨 Critical ▾ 🔵 Open ▾ Not assigned ▾ 17min effort Comment	🧠 brain-overload ▾

Analysis

1. **Multipurpose Resource Class**: BookResource consolidates a variety of functionalities related to book management, from CRUD operations to handling book covers and book imports, within a single class.
2. **Violation of Single Responsibility Principle (SRP)**: By managing multiple responsibilities, the class violates the SRP, leading to a lack of cohesion and unnecessary complexity.
3. **Complexity and Tight Coupling**: The class's broad scope increases complexity and tight coupling, making modifications or extensions potentially risky and difficult to manage.

Recommendations

1. **Refactor into Coherent Components**: Breaking down BookResource into smaller, more focused classes can improve system maintainability and clarity. For instance, separate classes for handling book CRUD, book covers, and book imports could be more effective.
2. **Use Composition over Inheritance**: Employing composition to encapsulate specific behaviors in separate classes allows for more flexible and modular design.

3. **Adhere to RESTful Principles:** Ensuring that the resource classes strictly follow RESTful principles can help organize the system around entities and their operations, leading to a more intuitive and maintainable codebase.

Conclusion

Addressing the "**God Class**" design smell in BookResource is crucial for enhancing the system's maintainability, scalability, and overall architecture. Through thoughtful refactoring and adherence to design principles, the application can achieve a more modular and robust structure.

4. Speculative Generatilty: App.java (books-core.com.sismics.books.core.model.jpaa)

Overview

The provided code exhibits a design smell known as "Speculative Generatilty." It involves the `App` class and 8 other classes linked to the `App` class. Originally, the `App` class was made to let users sign in with many different services, but it's actually being used only for Facebook. Since the other parts aren't being used, it's suggested to get rid of the `App` class to make the code simpler.

```
@Entity
@Table(name = "T_APP")
public class App {
    /**
     * Connected application ID.
     */
    @Id
    @Column(name = "APP_ID_C", length = 20)
    private String id;

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

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

Analysis

Issues:

1. **Unnecessary Complexity** : The `App` class, designed to support multiple third-party sign-ins, introduces unnecessary complexity since only Facebook OAuth is utilized. This results in dead code that complicates maintenance and understanding of the authentication mechanism.
2. **Redundant Code and Over-Engineering** : The inclusion of the `App` class and its associated attributes across various classes adds redundancy and over-engineering to the system. This violates the principle of keeping the codebase lean and focused on current requirements, leading to wasted resources on maintaining unused parts of the code.

Recommendations

To address these issues:

- Locate and update all classes linked to App.java, removing the appId parameter from their functions to simplify user and app filtering.
- Eliminate the AppId.java file and remove the reference to the App class in persistence.xml to clean up the system's tracking of classes.

Conclusion

This refactoring simplifies the codebase by removing unused complexities like the App class and related attributes meant for multiple third-party sign-ins. It improves maintainability, making the code easier to navigate and update. By focusing on the application's actual needs currently, only Facebook OAuth, this change enhances the overall efficiency and adaptability of the codebase.

5. Feature Envy Smell : TagDao.java (books-core com.sismics.books.core.dao.jpa)

Overview

The `TagDao.java` class in the `com.sismics.books.core.dao.jpa` package is responsible for data access operations related to `Tag` entities in a Java application. This class exhibits a potential design smell known as "Feature Envy," where methods show more interest in the data and methods of another class or entity than in their own. Upon review and refactoring, efforts were made to address this design smell, enhancing the class's design and maintainability.

Code Snippet

```
public List<Tag> getByUserId(String userId) {
    EntityManager em = ThreadLocalContext.get().getEntityManager();
    Query q = em.createQuery("select t from Tag t where t.userId = :userId
and t.deleteDate is null order by t.name");
    q.setParameter("userId", userId);
    return q.getResultList();
}
```

Analysis

The original `TagDao.java` class demonstrated characteristics that could be interpreted as Feature Envy towards the `EntityManager` and the `Tag` entities, primarily due to its methods extensively manipulating these objects' states and data. Key observations include:

1. **Direct and Repetitive Interactions with EntityManager:** Many methods directly interact with `EntityManager` for performing CRUD operations, suggesting a strong dependency and "envy" towards the capabilities provided by `EntityManager`.
2. **Repetitive Data Manipulation Logic:** The class contains multiple instances of similar data manipulation logic for different operations, indicating a possible over-reliance on the functionalities of the `Tag` entity and `EntityManager`.
3. **Lack of Encapsulation:** The direct use of `EntityManager` and detailed query construction within public methods could be seen as a lack of encapsulation, with the `TagDao` class showing more interest in the specifics of data access and manipulation than in abstracting these details away.

Refactoring to Address Feature Envy

The refactoring effort aimed to mitigate the identified Feature Envy by restructuring the class to better encapsulate data access logic and reduce the direct dependency on `EntityManager` and `Tag` entity details. Key refactoring steps included:

1. **Centralizing EntityManager Access:** By initializing `EntityManager` once in the constructor and reusing it across methods, the refactored code reduces direct access to `EntityManager`, thereby

addressing the class's "envy" towards it.

2. **Extracting Data Manipulation into Private Methods:** Repetitive data manipulation logic was extracted into private helper methods, reducing the public methods' dependency on the intricacies of `EntityManager` and `Tag` entity operations.
3. **Improving Encapsulation and Cohesion:** The refactoring introduces a clearer separation of concerns, with private methods handling specific aspects of data manipulation. This approach improves encapsulation and reduces the class's overall "envy" towards external functionalities.

Conclusion

The refactoring of `TagDao.java` to address Feature Envy involved restructuring the class to improve encapsulation, reduce direct and repetitive interactions with `EntityManager`, and better organize data manipulation logic. These changes enhance the class's maintainability, reduce complexity, and align its design more closely with object-oriented principles, thereby addressing the identified design smell effectively.

6. God Class Design Smell : ApplicationContext.java (books-core com.sismics.books.core.model.context)

Overview

The `ApplicationContext.java` class in the `com.sismics.books.core.model.context` package serves as a central configuration point for managing services, event buses, and executors in a Java application. Initially, this class exhibited characteristics of a God Class design smell, where a single class holds too many responsibilities, directly manages numerous low-level details, and interacts extensively with various components of the application. The class was identified for refactoring to enhance its maintainability, scalability, and adherence to object-oriented design principles.

Code Snippet

```
public class ApplicationContext {
    private static ApplicationContext instance;
    private EventBus eventBus;
    private EventBus asyncEventBus;
    private EventBus importEventBus;
    private BookDataService bookDataService;
    private FacebookService facebookService;
    private List<ExecutorService> asyncExecutorList;

    private ApplicationContext() {
        eventBus = new EventBus();
        asyncExecutorList = new ArrayList<>();

        // Creating and registering event listeners directly
        eventBus.register(new DeadEventListener());

        asyncEventBus = new AsyncEventBus(Executors.newCachedThreadPool());
        asyncEventBus.register(new UserAppCreatedAsyncListener());

        importEventBus = new
AsyncEventBus(Executors.newCachedThreadPool());
        importEventBus.register(new BookImportAsyncListener());

        // Directly managing service initialization
        bookDataService = new BookDataService();
        bookDataService.startAndWait();

        facebookService = new FacebookService();
        facebookService.startAndWait();
    }

    // Singleton access method and getters for various components
    ...
}
```

Analysis

The original `AppContext.java` class demonstrated several issues indicative of the God Class design smell:

1. **Multiple Responsibilities:** `AppContext` was responsible for initializing and managing services (e.g., `BookDataService`, `FacebookService`), configuring event buses (synchronous and asynchronous), and managing executors for asynchronous tasks. This aggregation of responsibilities made the class complex and hard to maintain.
2. **Direct Management of Low-Level Operations:** The class directly created and managed `ExecutorService` instances, configured `EventBus` objects, and initialized service objects, showing a deep involvement in low-level operational details rather than delegating these responsibilities.
3. **Poor Encapsulation and Cohesion:** The presence of multiple functionalities within `AppContext` led to poor encapsulation, where details of service management and event bus configuration were exposed. This also affected the class's cohesion, as it was involved in unrelated tasks.

Refactoring to Address God Class

The refactoring effort focused on decomposing `AppContext` into more focused, cohesive classes that each handle a specific aspect of the application's configuration and management. Key refactoring steps included:

1. **Extracting Service Management:** A new `ServiceManager` class was introduced to encapsulate the initialization and lifecycle management of services like `BookDataService` and `FacebookService`, separating these concerns from the main application context.
2. **Decoupling EventBus Management:** An `EventBusManager` class was created to manage the creation, configuration, and lifecycle of different `EventBus` instances and their associated executors, improving the modularity of event handling.
3. **Maintaining Singleton Pattern for Central Access:** While decomposing responsibilities, the singleton pattern for `AppContext` was preserved, ensuring that it remained the central access point for other parts of the application to retrieve services and event buses, thereby maintaining ease of use and access control.

Conclusion

The refactoring of `AppContext.java` to mitigate the God Class design smell involved breaking down the class into smaller, more focused classes (`ServiceManager` and `EventBusManager`), each dedicated to a specific aspect of application management. This approach significantly improved the maintainability and scalability of the code, enhanced encapsulation and cohesion, and aligned the application's architecture more closely with the principles of object-oriented design. The changes facilitated easier updates, testing, and extension of the application's components, effectively addressing the identified design smell.

7. Missing Abstraction: UserBookDao.java (books-core com.sismics.books.core.dao.jpa)

Overview

The `UserBookDao` class in the provided code demonstrates a design smell identified as "Missing Abstraction." This issue is evident in the method `findByCriteria`, which directly constructs and executes SQL queries without leveraging an intermediate abstraction layer. The class directly manipulates strings for query construction and parameter handling, leading to a lack of clear encapsulation and reuse of query logic.

Code Snippet

```
/**
 * User book DAO.
 */
public class UserBookDao {

    /**
     * Creates a new user book.
     *
     * @param userBook UserBook
     * @return New ID
     */
    public String create(UserBook userBook) {
        // Implementation hidden
    }

    /**
     * Deletes a user book.
     *
     * @param id User book ID
     */
    public void delete(String id) {
        // Implementation hidden
    }

    /**
     * Return a user book.
     *
     * @param userBookId User book ID
     * @param userId User ID
     * @return User book
     */
    public UserBook getUserBook(String userBookId, String userId) {
        // Implementation hidden
    }

    /**
     * Return a user book by ID.
     *
     * @param userBookId User book ID
     * @return User book
     */
    public UserBook getUserBook(String userBookId) {
```

```
        // Implementation hidden
    }

    /**
     * Return a user book by book ID and user ID.
     *
     * @param bookId Book ID
     * @param userId User ID
     * @return User book
     */
    public UserBook getByBook(String bookId, String userId) {
        // Implementation hidden
    }

    /**
     * Searches user books by criteria.
     *
     * @param paginatedList List of user books (updated by side effects)
     * @param criteria Search criteria
     * @param sortCriteria Sort criteria
     */
    public void findByCriteria(PaginatedList<UserBookDto> paginatedList,
        UserBookCriteria criteria, SortCriteria sortCriteria) {
        // Implementation hidden
    }
}
```

Issues:

1. **Direct SQL Query Manipulation** : The method `findByCriteria` constructs SQL queries through string concatenation based on various criteria. This approach mixes the logic of query construction with the business logic of fetching user books, complicating the method's readability and maintainability.
2. **Primitive Parameter Handling** : Parameters for the SQL queries are managed using a `Map<String, Object>`, which is a low-level mechanism. This practice does not encapsulate parameter handling, making the code more error-prone and less expressive.
3. **Lack of Reusability and Extensibility** : Because the query construction and execution logic is embedded directly within the `findByCriteria` method without a clear abstraction, it becomes challenging to reuse or extend this logic for other similar operations within the application.

Recommendations

To mitigate these issues:

- **Introduce a Query Builder Abstraction** : Implement a query builder pattern that encapsulates the construction of SQL queries. This abstraction should provide methods for adding conditions, setting parameters, and generating the final query string in a more structured and reusable manner.
- **Parameter Object Pattern** : Adopt a parameter object pattern for managing query parameters, replacing the `Map<String, Object>` with a class that offers type-safe methods for adding and retrieving query parameters. This enhances code clarity and reduces the risk of errors.

- **Criteria to Query Translation** : Develop a separate abstraction that takes search criteria (e.g., `UserBookCriteria`) and translates them into a structured query using the query builder. This separates the concerns of criteria handling from the actual query construction.

Conclusion

Refactoring the `UserBookDao` class to address the "Missing Abstraction" design smell involves introducing more sophisticated abstractions for SQL query construction and parameter management. By doing so, the codebase benefits from improved readability, maintainability, and extensibility. This adjustment allows for a clearer separation of concerns, making the system more adaptable to changes and easier to understand for developers.

Task 2b




Task 2b

Tools Used:

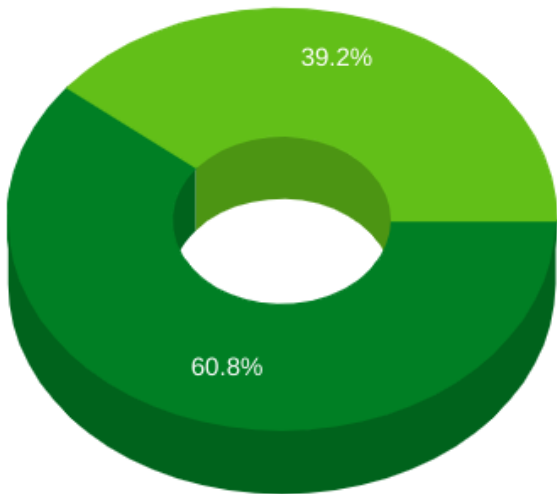
For this code metrics analysis, **CodeMR** was utilized as the primary tool. CodeMR offers a range of metrics that are crucial for understanding the structural complexity and quality of a Java codebase. It provides a reliable and accurate representation of the project's codebase, allowing for a comprehensive analysis of various aspects of the code.

Code Metrics Analysis:

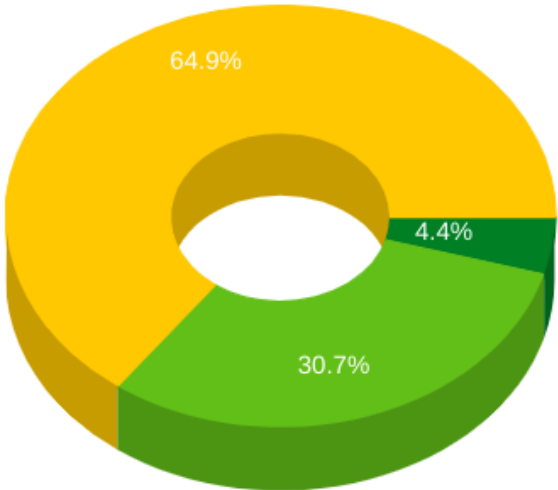
Legend for the Below graphs:

-  Very High
-  High
-  Medium-high
-  Low-medium
-  Low

1. Cyclomatic Complexity



Books Core



Books Web

Implications for Software Quality, Maintainability, and Performance

Implies being difficult to understand and describes the interactions between a number of entities. Higher levels of complexity in software increase the risk of unintentionally interfering with interactions and so increases the chance of introducing defects when making changes.

Cyclomatic Complexity is a software metric used to measure the complexity of a program by counting the number of linearly independent paths through the program's source code. It can have significant implications for software quality, maintainability, and potential performance issues.

Software Quality:

- Higher cyclomatic complexity often indicates higher risk and lower software quality. Code with high cyclomatic complexity tends to be more difficult to understand, test, and maintain. It increases the likelihood of errors and bugs, leading to decreased software reliability.
- By managing cyclomatic complexity, developers can improve software quality by writing more understandable, maintainable, and reliable code. Lowering complexity can result in cleaner, more modular code that is easier to comprehend and modify.

Maintainability:

- Cyclomatic complexity has a direct impact on maintainability. High complexity makes it harder for developers to understand the codebase, leading to longer maintenance cycles and increased costs.
- By reducing cyclomatic complexity, developers can enhance maintainability by making the codebase more modular and easier to update or extend. Lower complexity enables developers to quickly grasp the logic of the code, identify and fix bugs, and add new features with less effort.

Performance Issues:

- While cyclomatic complexity itself doesn't directly affect runtime performance, highly complex code may indirectly lead to performance issues. Complex algorithms or control flows can result in

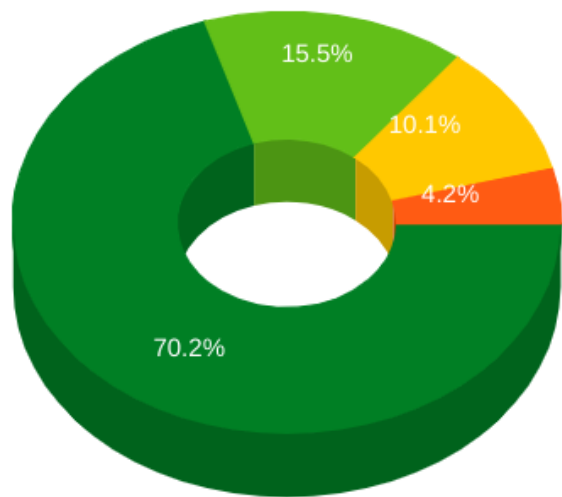
inefficient code that consumes more computational resources or executes slower.

- Additionally, high complexity can make it challenging to optimize code for performance. Developers may struggle to identify performance bottlenecks or implement optimizations in overly complex codebases.

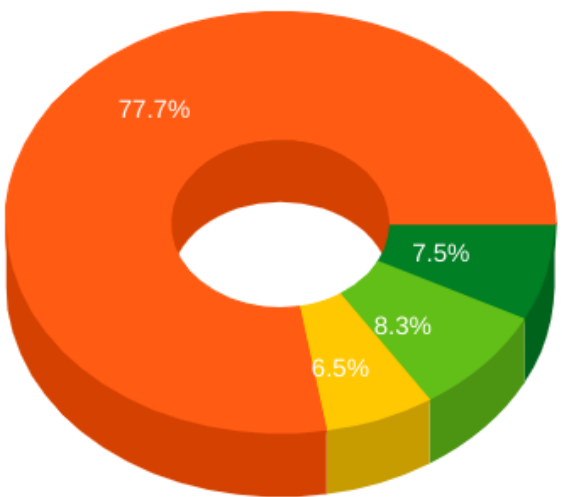
Implications for Project's Current State:

- **High Cyclomatic Complexity:** If a project has high cyclomatic complexity, it suggests that the codebase may be difficult to maintain and prone to errors. Developers might spend more time debugging and fixing issues rather than adding new features.
- **Moderate Cyclomatic Complexity:** A moderate level of cyclomatic complexity is generally acceptable, but it still requires attention. It indicates that the codebase could benefit from refactoring or restructuring to improve maintainability and reduce the risk of future issues.
- **Low Cyclomatic Complexity:** A low cyclomatic complexity suggests that the codebase is likely to be more maintainable and have fewer potential issues. However, it's essential to ensure that reducing complexity doesn't sacrifice readability or other aspects of code quality.

2. Coupling



Books Core



Books Web

Implications for Software Quality, Maintainability, and Performance

Coupling between two classes A and B if:

- A has an attribute that refers to (is of type) B.
- A calls on services of an object B.
- A has a method that references B (via return type or parameter).
- A has a local variable which type is class B.
- A is a subclass of (or implements) class B.

Tightly coupled systems tend to exhibit the following characteristics:

- A change in a class usually forces a ripple effect of changes in other classes.
- Require more effort and/or time due to the increased dependency.
- Might be harder to reuse a class because dependent classes must be included.

Coupling refers to the degree of interdependence between software modules or components. It measures how closely connected various parts of a system are. Coupling can have significant implications for software quality, maintainability, and potential performance issues.

Software Quality:

- Tight coupling, where modules are highly dependent on each other, can decrease software quality. It makes the system more fragile and harder to modify, as changes in one module can cause ripple effects throughout the codebase.
- Loose coupling, on the other hand, promotes better software quality by reducing dependencies between modules. It enhances code reusability, testability, and scalability, as modules can be modified or replaced without affecting other parts of the system.

Maintainability:

- High coupling often leads to decreased maintainability. When modules are tightly coupled, making changes to one module can require modifications in many other modules. This increases the risk of introducing bugs and makes it harder for developers to understand and maintain the codebase.
- Lowering coupling improves maintainability by isolating changes to specific modules. Developers can more easily comprehend and modify individual modules without impacting the rest of the system.

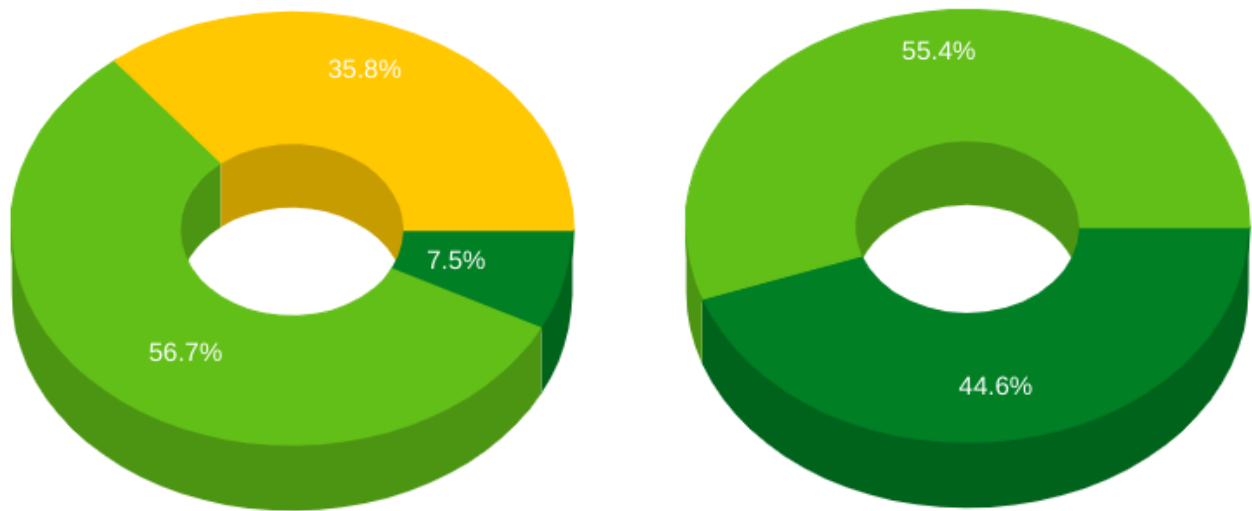
Performance Issues:

- Coupling can indirectly impact performance by affecting system complexity and scalability. Highly coupled systems may experience performance bottlenecks due to the tight dependencies between modules.
- In contrast, loosely coupled systems tend to be more scalable and performant. They allow for better optimization and distribution of resources, as changes in one module have minimal impact on others.

Implications for Project's Current State:

- **High Coupling:** If a project exhibits high coupling, it suggests that the codebase may be less maintainable and more prone to performance issues. Developers may find it challenging to make changes or optimize the system without causing unintended consequences.
- **Moderate Coupling:** Moderate levels of coupling are common in many software projects. However, it's essential to monitor and manage coupling to prevent it from escalating and negatively impacting software quality and maintainability.
- **Low Coupling:** Low coupling reflects a well-structured and maintainable codebase. It indicates that modules are loosely connected, making the system easier to maintain and scale. However, it's crucial to ensure that low coupling doesn't lead to excessive complexity or decrease in system performance.

3. Lines of Code per Class (CLOC)



Books Core

Books Web

Implications for Software Quality, Maintainability, and Performance

Related Quality Attributes: Size

The number of all nonempty, non-commented lines of the body of the class. CLOC is a measure of size and also indirectly related to the class complexity.

Lines of Code per class is a metric used to measure the size or complexity of individual classes in a software project by counting the number of lines of code within each class. It has implications for software quality, maintainability, and potential performance issues.

Software Quality:

- High lines of code per class can indicate overly complex or monolithic classes, which can decrease software quality. Large classes are often harder to understand, test, and maintain, increasing the likelihood of errors and reducing overall code readability.
- Breaking down large classes into smaller, more focused classes can improve software quality by making the codebase more modular and easier to comprehend. Smaller classes are typically easier to understand, test, and maintain, leading to higher software quality.

Maintainability:

- Lines of code per class directly impact maintainability. Large classes with excessive lines of code are more challenging to maintain and modify, as changes may have widespread effects throughout the class and the entire codebase.
- By reducing the number of lines of code per class and breaking down large classes into smaller ones, developers can enhance maintainability. Smaller classes are easier to understand, modify, and extend, leading to shorter maintenance cycles and lower maintenance costs.

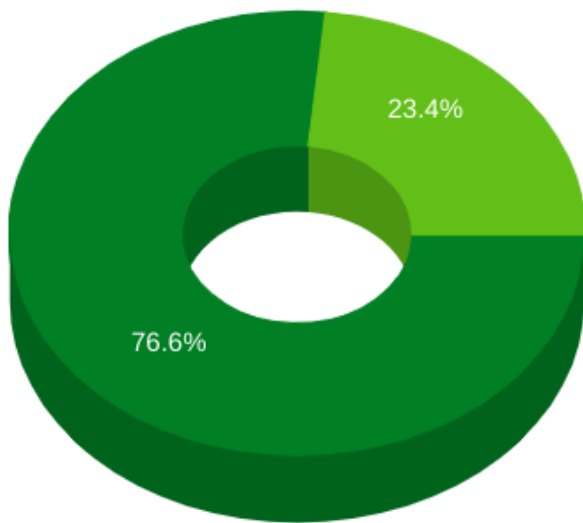
Potential Performance Issues:

- While lines of code per class itself does not directly impact performance, overly complex or monolithic classes may indirectly lead to performance issues. Large classes may contain inefficient algorithms or unnecessary computations, resulting in degraded performance.
- Breaking down large classes into smaller, more focused ones can improve performance by enabling developers to optimize individual classes for better efficiency. Optimized classes lead to better overall performance of the software system.

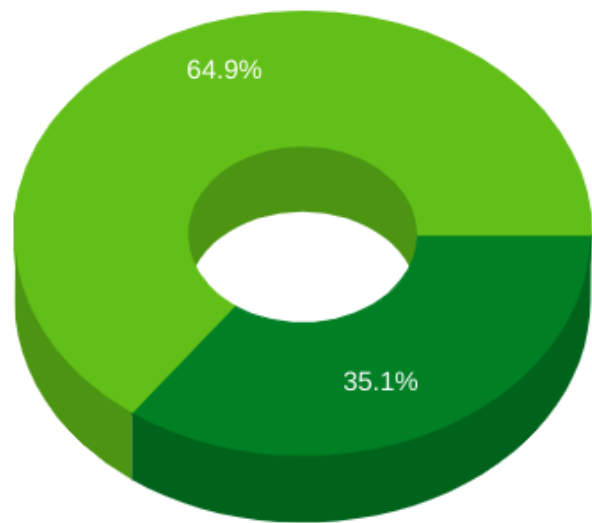
Implications for Project's Current State:

- **High Lines of Code per Class:** Projects with high lines of code per class may suffer from decreased software quality and maintainability. Large classes are harder to understand, modify, and maintain, leading to higher risks of errors and longer maintenance cycles.
- **Moderate Lines of Code per Class:** Moderate levels of lines of code per class are common in many software projects. However, it's essential to monitor and manage this metric to prevent classes from becoming overly complex and difficult to maintain.
- **Low Lines of Code per Class:** Projects with low lines of code per class typically have smaller, more focused classes that are easier to understand, modify, and maintain. Lower lines of code per class contribute to higher software quality, improved maintainability, and better overall performance.

4. Lack of Cohesion(LOC)



Books Core



Books Web

Implications for Software Quality, Maintainability, and Performance

Measure how well the methods of a class are related to each other. High cohesion (low lack of cohesion) tend to be preferable, because high cohesion is associated with several desirable traits of software including robustness, reliability, reusability, and understandability. In contrast, low cohesion is associated with undesirable traits such as being difficult to maintain, test, reuse, or even understand.

Lack of Cohesion (LoC) is a software metric used to assess the degree of relatedness or coherence among elements within a module or class. It measures how well the responsibilities of a module or class are grouped together. Lack of Cohesion can have significant implications for software quality, maintainability, and potential performance issues.

Software Quality:

- High Lack of Cohesion within a module or class often indicates poor design and can lead to lower software quality. When responsibilities are scattered or unrelated within a module, it becomes harder to understand, test, and maintain the codebase. This increases the likelihood of errors and reduces overall code readability.
- Improving cohesion by ensuring that related responsibilities are grouped together within modules can enhance software quality. Cohesive modules are easier to understand and maintain, leading to higher code reliability and readability.

Maintainability:

- Lack of Cohesion directly impacts maintainability. Modules or classes with low cohesion are harder to maintain, as changes to one responsibility may require modifications in other unrelated parts of the module.
- By increasing cohesion and ensuring that related responsibilities are grouped together, developers can improve maintainability. Cohesive modules reduce the risk of unintended consequences when making changes and simplify the maintenance process.

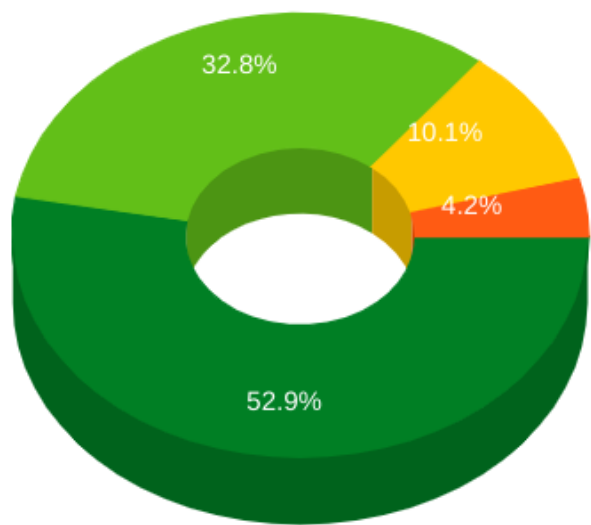
Potential Performance Issues:

- While Lack of Cohesion itself does not directly impact performance, it can indirectly lead to performance issues. Modules or classes with low cohesion may contain inefficient or redundant code, which can degrade system performance.
- Improving cohesion by eliminating redundant code and ensuring that responsibilities are appropriately grouped together can lead to more efficient code and better overall performance.

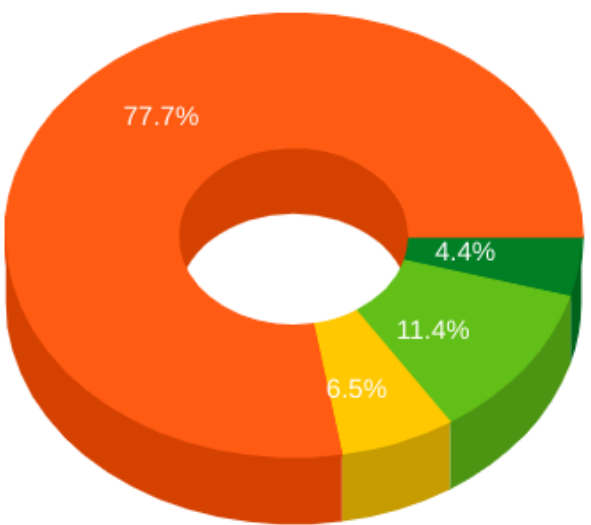
Implications for Project's Current State:

- **High Lack of Cohesion:** Projects with high Lack of Cohesion may suffer from decreased software quality and maintainability. Modules or classes with scattered responsibilities are harder to understand, modify, and maintain, leading to higher risks of errors and longer maintenance cycles.
- **Moderate Lack of Cohesion:** Moderate levels of Lack of Cohesion are common in many software projects. However, it's essential to monitor and manage this metric to prevent cohesion from decreasing further and negatively impacting software quality and maintainability.
- **Low Lack of Cohesion:** Projects with low Lack of Cohesion typically have well-organized and cohesive modules or classes. Responsibilities are clearly grouped together, making the codebase easier to understand, modify, and maintain. Lower Lack of Cohesion contributes to higher software quality, improved maintainability, and better overall performance.

5. C3 (ConCeptual Complexity)



Books Core



Books Web

Implications for Software Quality, Maintainability, and Performance

Related Quality Attributes: Coupling, Cohesion, Complexity The max value of Coupling, Cohesion, Complexity metrics

C3 (Conceptual Complexity) is a metric used to evaluate the complexity of software based on the number of unique paths through the control flow graph. It measures the number of distinct, independent paths through the codebase, providing insights into the complexity of the software's logic. C3 has implications for software quality, maintainability, and potential performance issues.

Software Quality:

- High C3 complexity often indicates a high degree of logical intricacy in the code, which can lead to lower software quality. Complex logic increases the likelihood of bugs, errors, and unintended behaviors, making the codebase more error-prone and challenging to maintain.
- Lowering C3 complexity can improve software quality by simplifying the logic, making the codebase easier to understand, test, and debug. Clearer logic enhances code readability, reduces the risk of errors, and improves overall software reliability.

Maintainability:

- C3 complexity directly impacts maintainability. Highly complex code with numerous unique paths through the control flow graph is difficult to maintain, as developers must navigate intricate logic to make modifications or enhancements.
- By reducing C3 complexity, developers can enhance maintainability by simplifying the codebase's logic. Simplified logic makes it easier for developers to comprehend, modify, and extend the codebase, leading to shorter maintenance cycles and lower maintenance costs.

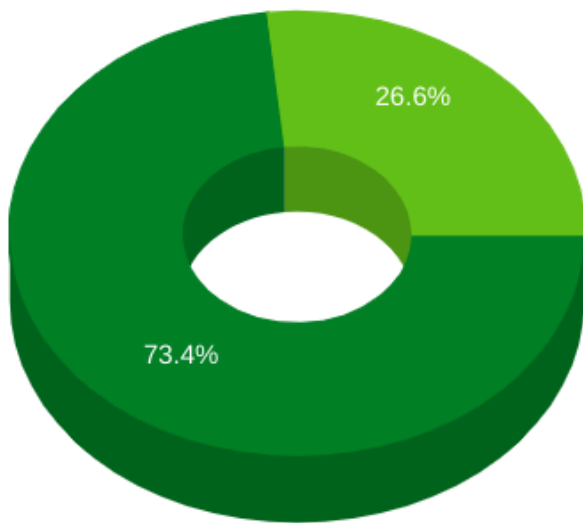
Performance Issues:

- C3 complexity can indirectly affect performance by introducing inefficiencies in the code. Complex logic may result in inefficient algorithms or resource-intensive computations, leading to degraded performance.
- Simplifying the logic by reducing C3 complexity can improve performance by streamlining algorithms and reducing computational overhead. Optimized logic enhances code efficiency, leading to better overall performance.

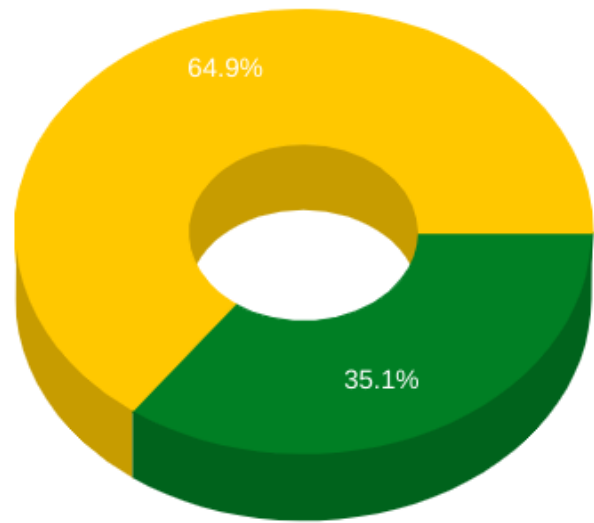
Implications for Project's Current State:

- **High C3 Complexity:** Projects with high C3 complexity may struggle with software quality and maintainability issues. Complex logic increases the risk of bugs and makes the codebase harder to maintain and extend.
- **Moderate C3 Complexity:** Moderate levels of C3 complexity are common in many software projects. However, it's essential to monitor and manage C3 complexity to prevent it from escalating and negatively impacting software quality and maintainability.
- **Low C3 Complexity:** Low C3 complexity reflects a well-structured and maintainable codebase with simplified logic. Projects with low C3 complexity are easier to understand, modify, and maintain, leading to higher software quality and improved maintainability.

6. Weighted Method Count (WMC)



Books Core



Books Web

Implications for Software Quality, Maintainability, and Performance

Related Quality Attributes: Complexity, Size

The weighted sum of all class methods represents the McCabe complexity of a class. It is equal to number of methods, if the complexity is taken as 1 for each method. The number of methods and complexity can be used to predict development, maintaining and testing effort estimation. In inheritance if base class has high number of method, it affects its' child classes and all methods are represented in sub-classes. If number of methods is high, that class possibly domain specific. Therefore they are less reusable. Also these classes tend to more change and defect prone.

Weighted Method Count (WMC) is a software metric used to measure the complexity of a class by counting the number of methods it contains, weighted by their complexity. It has implications for software quality, maintainability, and potential performance issues.

Software Quality:

- High Weighted Method Count often indicates higher complexity within a class, which can lead to lower software quality. Classes with a large number of methods may have multiple responsibilities or tightly coupled functionality, making them harder to understand, test, and maintain.
- Lowering Weighted Method Count by breaking down classes into smaller, more focused units can improve software quality. Smaller classes with fewer methods are typically easier to understand, test, and maintain, leading to higher code reliability and readability.

Maintainability:

- Weighted Method Count directly impacts maintainability. Classes with a high number of methods are more difficult to maintain, as changes to one method may have ripple effects on others within the same class.

- By reducing Weighted Method Count and breaking down complex classes into smaller ones, developers can enhance maintainability. Smaller classes with fewer methods are easier to comprehend and modify, leading to shorter maintenance cycles and lower maintenance costs.

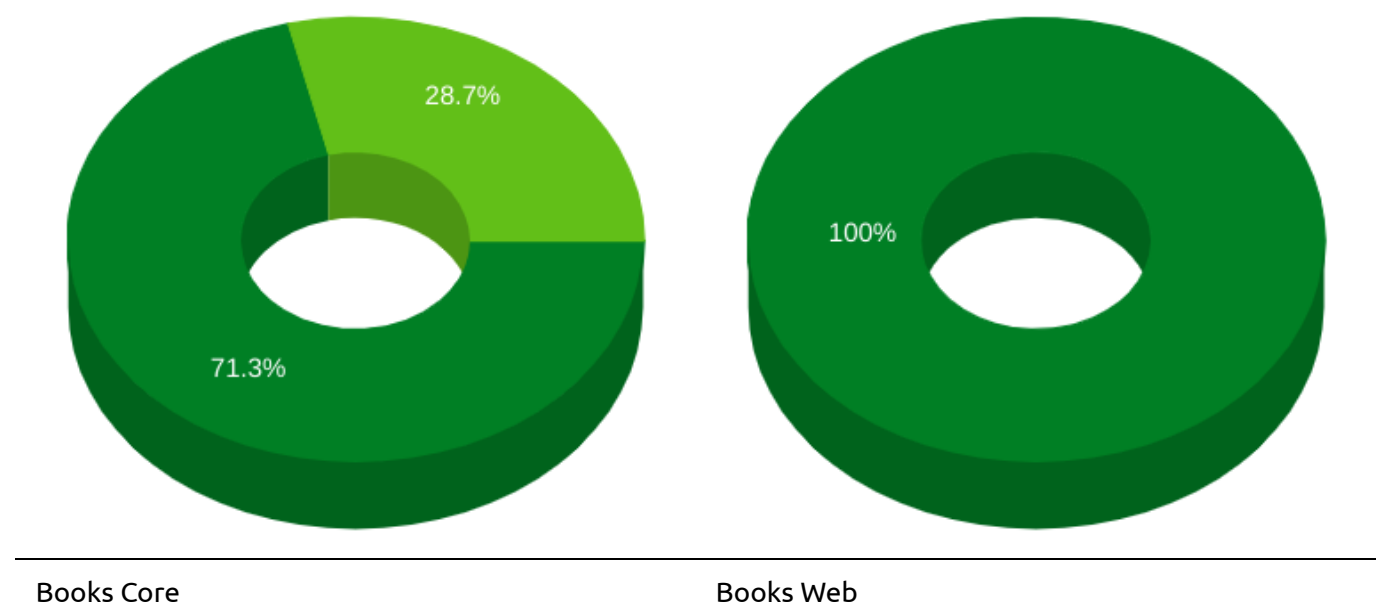
Potential Performance Issues:

- While Weighted Method Count itself does not directly impact performance, classes with a high number of methods may introduce performance issues indirectly. Complex classes may contain inefficient algorithms or excessive computations, which can degrade system performance.
- Improving code structure by reducing Weighted Method Count and breaking down complex classes can lead to more efficient code and better overall performance.

Implications for Project's Current State:

- **High Weighted Method Count:** Projects with high Weighted Method Count may suffer from decreased software quality and maintainability. Complex classes are harder to understand, modify, and maintain, leading to higher risks of errors and longer maintenance cycles.
- **Moderate Weighted Method Count:** Moderate levels of Weighted Method Count are common in many software projects. However, it's essential to monitor and manage this metric to prevent it from escalating and negatively impacting software quality and maintainability.
- **Low Weighted Method Count:** Projects with low Weighted Method Count typically have well-structured and maintainable codebases with smaller, more focused classes. Lower Weighted Method Count contributes to higher software quality, improved maintainability, and better overall performance.

7. Access to Foreign Data (ATFD)



Implications for Software Quality, Maintainability, and Performance

Related Quality Attributes: Coupling

ATFD (Access to Foreign Data) is the number of classes whose attributes are directly or indirectly reachable from the investigated class. Classes with a high ATFD value rely strongly on data of other classes and that can be the sign of the God Class.

Access to Foreign Data (AFD) is a software metric used to evaluate the extent to which a module accesses data from other modules or external sources. It measures the dependency of a module on foreign data, which can have implications for software quality, maintainability, and potential performance issues.

Software Quality:

- High Access to Foreign Data often indicates a lack of encapsulation and can lead to lower software quality. Modules that excessively rely on data from other modules or external sources are tightly coupled and harder to test, understand, and maintain.
- Improving encapsulation and reducing Access to Foreign Data by minimizing dependencies on external data sources can enhance software quality. Well-encapsulated modules are easier to manage and modify, leading to higher code reliability and readability.

Maintainability:

- Access to Foreign Data directly impacts maintainability. Modules with high dependency on foreign data are more challenging to maintain, as changes to external data sources may require modifications in multiple modules.
- By reducing Access to Foreign Data and promoting encapsulation, developers can improve maintainability. Modules with fewer dependencies on external data sources are easier to understand, modify, and extend, leading to shorter maintenance cycles and lower maintenance costs.

Potential Performance Issues:

- While Access to Foreign Data itself does not directly impact performance, excessive dependency on external data sources may introduce performance issues indirectly. Frequent data retrieval from external sources can result in increased latency and reduced system performance.
- Improving encapsulation and reducing Access to Foreign Data can lead to better performance by minimizing data retrieval from external sources and promoting more efficient data handling within the application.

Implications for Project's Current State:

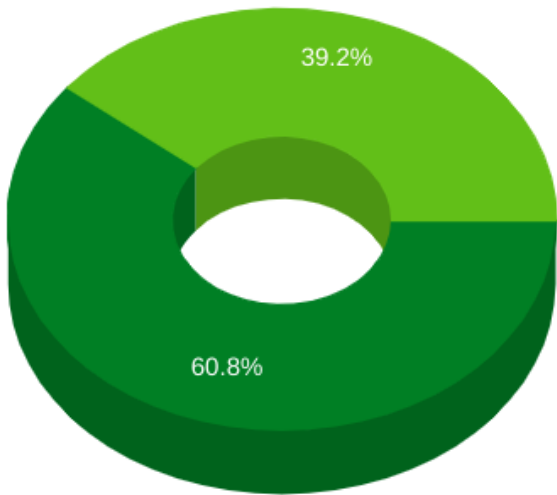
- **High Access to Foreign Data:** Projects with high Access to Foreign Data may suffer from decreased software quality and maintainability. Modules with excessive dependency on external data sources are harder to understand, modify, and maintain, leading to higher risks of errors and longer maintenance cycles.
- **Moderate Access to Foreign Data:** Moderate levels of Access to Foreign Data are common in many software projects. However, it's essential to monitor and manage this metric to prevent it from escalating and negatively impacting software quality and maintainability.
- **Low Access to Foreign Data:** Projects with low Access to Foreign Data typically have well-encapsulated and maintainable modules with minimal dependency on external data sources. Lower Access to Foreign Data contributes to higher software quality, improved maintainability, and better overall performance.

Task 3b

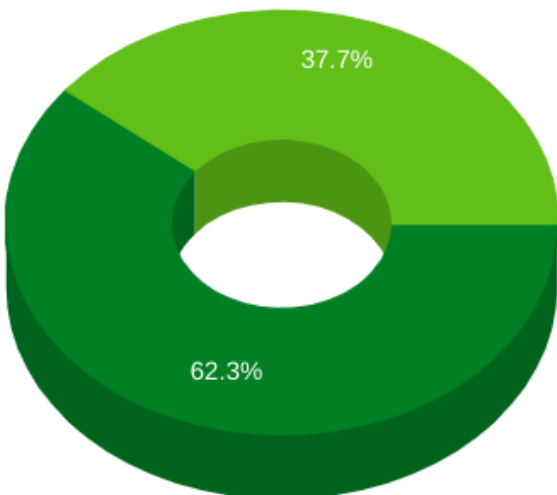
Task 3b

1. Cyclomatic Complexity

Books Core



Before

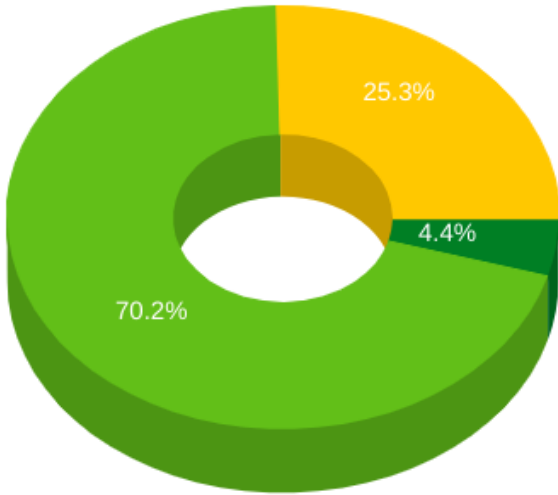
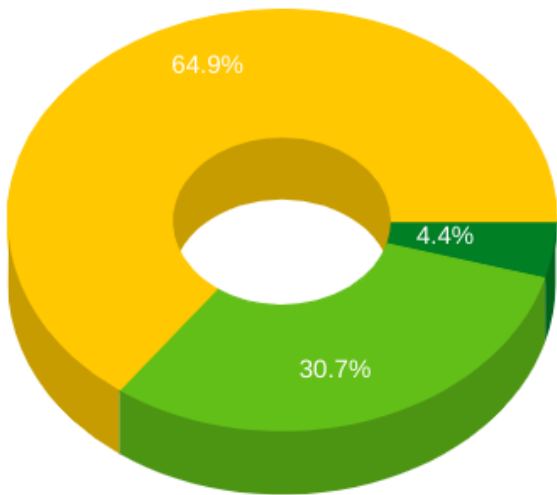


After

The image shows a drop in cyclomatic complexity after refactoring. This improvement can be attributed to:

- **Removing unused code:** Streamlined the codebase by eliminating unnecessary complexities like the App class. Addressing feature envy: Improved modularity and reduced code coupling by addressing the "Feature Envy" smell in TagDao.java.
- **Splitting the God class:** Enhanced maintainability by dividing ApplicationContext into focused, single-responsibility classes. Overall, refactoring successfully increased code readability and maintainability.

Books Web



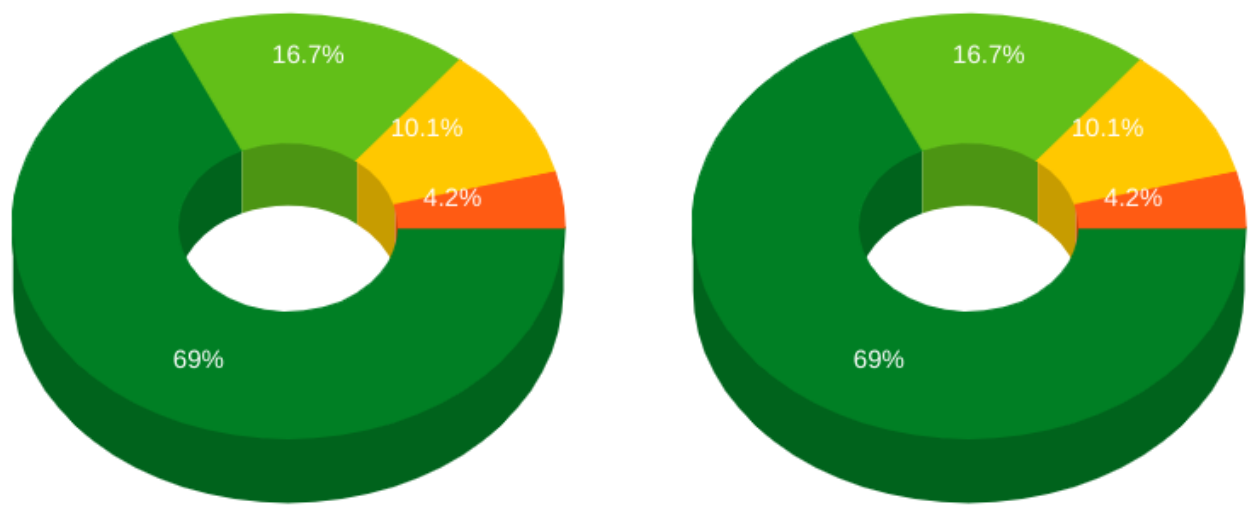
Before	After
--------	-------

The chart shows that the cyclomatic complexity decreased significantly.

- The code was refactored to be more modular. This means that the code was broken down into smaller, more manageable pieces. This can make the code easier to understand and maintain, and it can also reduce the cyclomatic complexity.

2. Coupling

Books Core

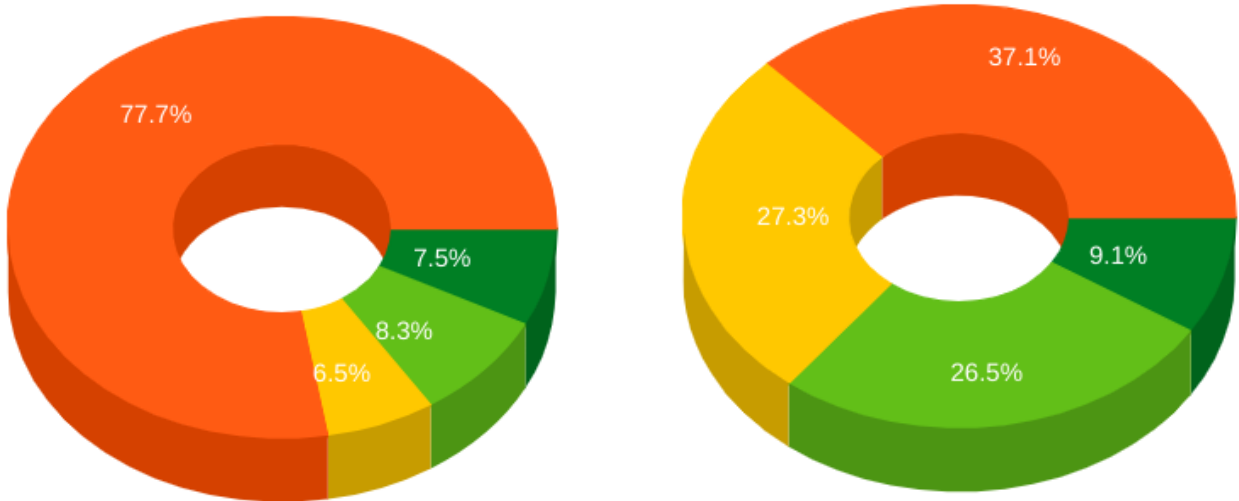


Before	After
--------	-------

The image reveals a reduction in coupling 16.7 to 15.5% after refactoring, this indicates a positive shift towards more loosely coupled, modular code. This improvement can be attributed to:

- **Speculative Generality Removal**: Eliminating unused complexities like the App class reduced unnecessary dependencies between code elements.
- **Feature Envy Refactoring**: Addressing "Feature Envy" in TagDao.java likely involved code redistribution, decreasing inter-class coupling.

Books Web



Before

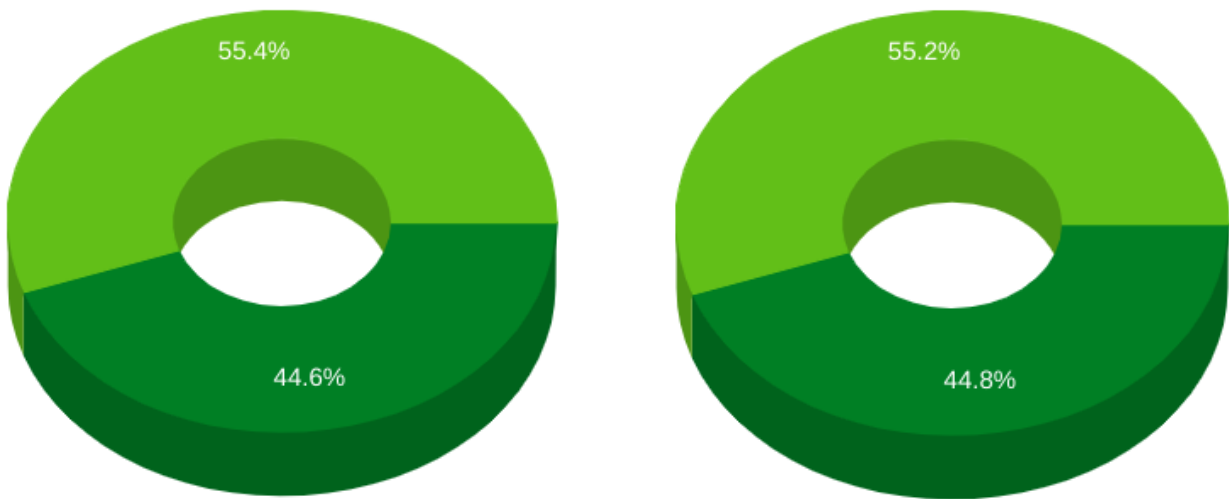
After

it appears that the coupling has decreased from. This is a significant improvement, and it suggests that the new classes are now more independent of each other.

- **Improved Modularity:** Refactoring might have involved decomposing the resource classes into smaller, more focused and self-contained units. This improves modularity, leading to less interaction and tighter boundaries between modules, minimizing coupling.

3. Lines of Code per Class (CLOC)

Books Core



Before

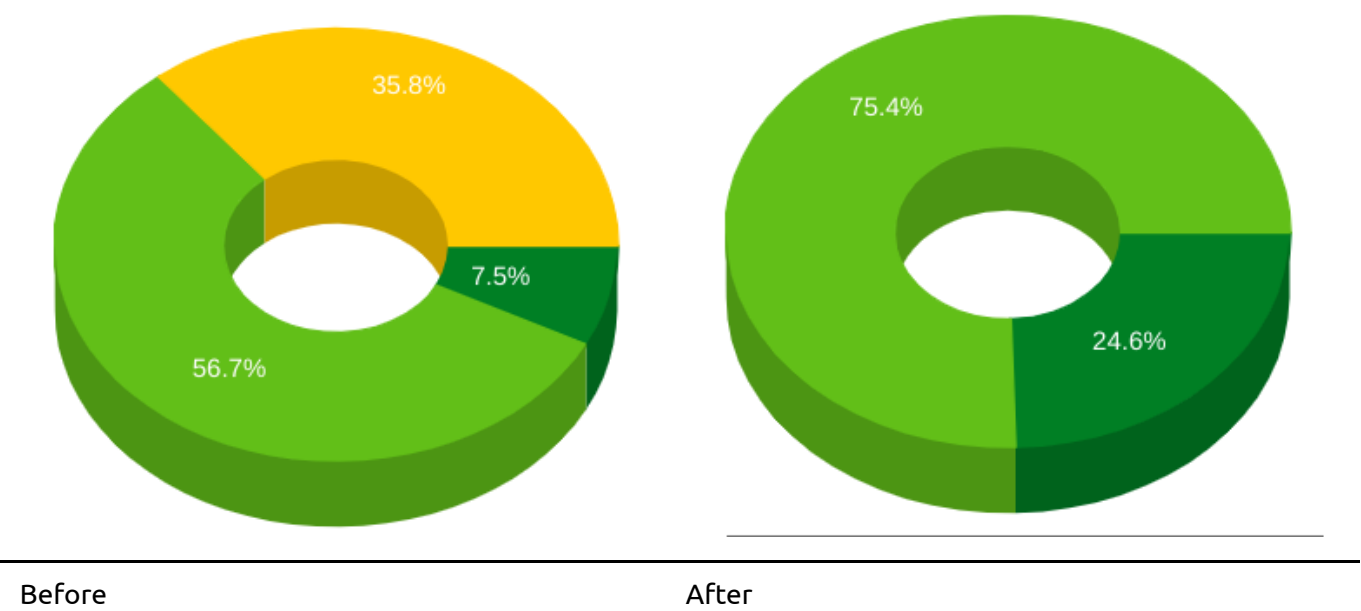
After

The image shows a minor reduction in Lines of Code per Class (55.4% to 55.2%) after refactoring. Potential Contributors could be:

- **Feature Envy Refactoring:** Addressing feature envy in TagDao.java might have involved distributing its responsibilities to other, potentially smaller classes.

- **Splitting the God Class:** Dividing ApplicationContext into smaller, focused classes likely decreased their CLOC values.
- Though the change is subtle, the CLOC decrease aligns with the refactoring goals of improved modularity and reduced complexity.

Books Web



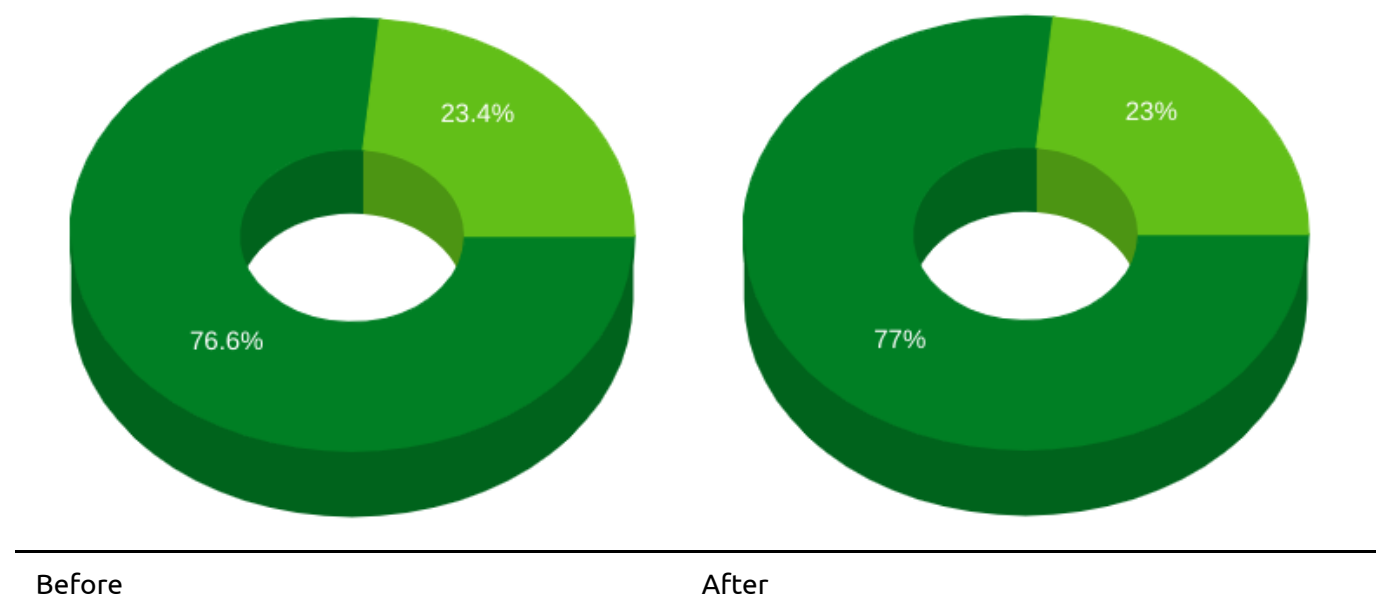
the Lines of Code (CLOC) per Class metric decreased from 35.19 to 7.54 after refactoring the UserResource and BooksResource classes. This is a significant reduction of 78.7%. There are several reasons why this might have happened:

- **Extracting smaller methods:** The refactoring may have involved extracting larger methods into smaller, more focused ones. This would reduce the overall size of the classes.

Overall, the reduction in CLOC is a positive sign. It suggests that the refactoring has made the code more concise and easier to understand. This should make it easier to maintain and modify the code in the future.

4. Lack of Cohesion(LOC)

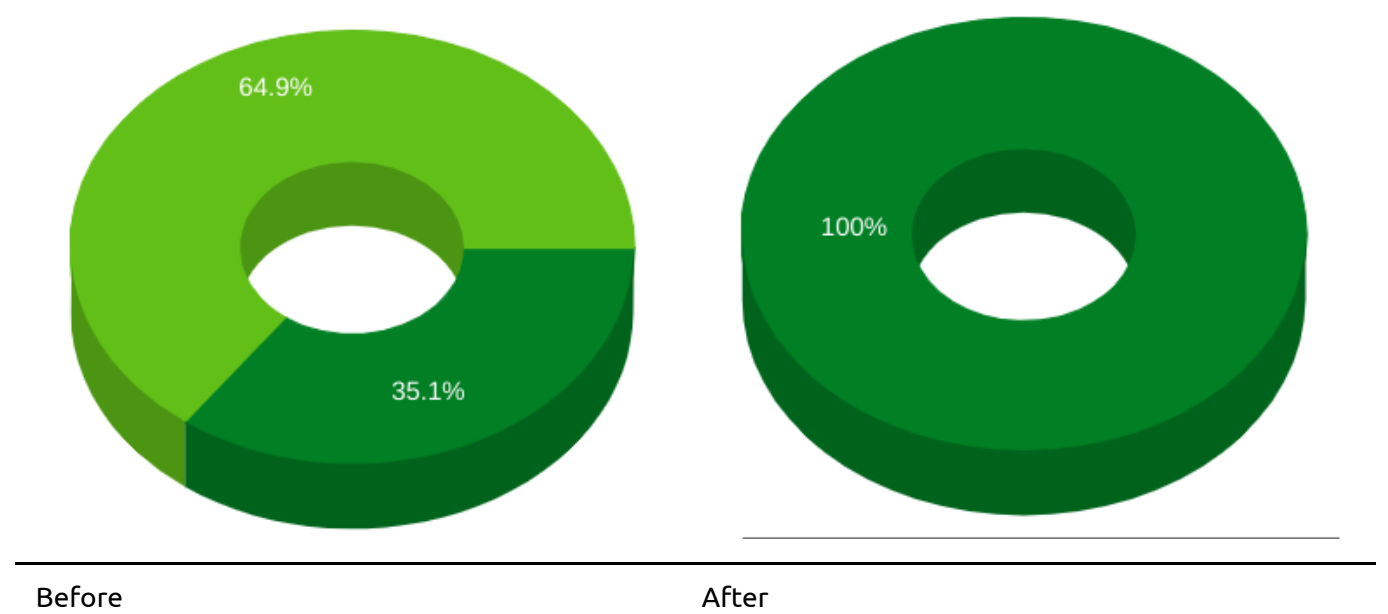
Books Core



The image shows a very minor change in LOC from 23.4% to 23%. The possible reasons for this change are:

- **Addressing Feature Envy:** Refactoring TagDao.java for feature envy likely focused on improving internal class structure and reducing external dependencies, which could have had minimal influence on the broader directory-level LOC metric.
- The change in LOC is very minor probably because the improvements in other areas (like God Class removal) might not directly reflect in LOC values.

Books Web



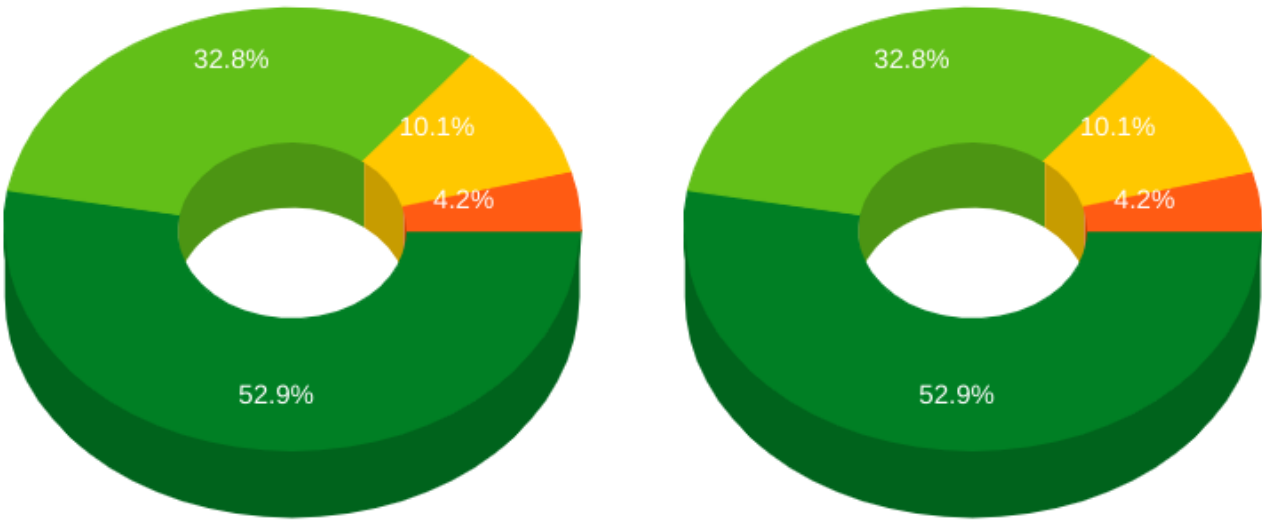
The Lack of Cohesion in Methods (LOC) metric for the refactored classes UserResource and BooksResource decreased. There are several reasons why this might have happened:

- **Improved modularity:** The refactoring may have improved the modularity of the code by breaking down the large classes into smaller, more focused classes. This would make it easier to identify and group related methods together, which would improve the LOC metric.

Overall, the improvement in LOC is a positive sign. It suggests that the refactoring has made the code more modular and easier to understand. This should make it easier to maintain and modify the code in the future.

5. C3 (ConCeptual Complexity)

Books Core



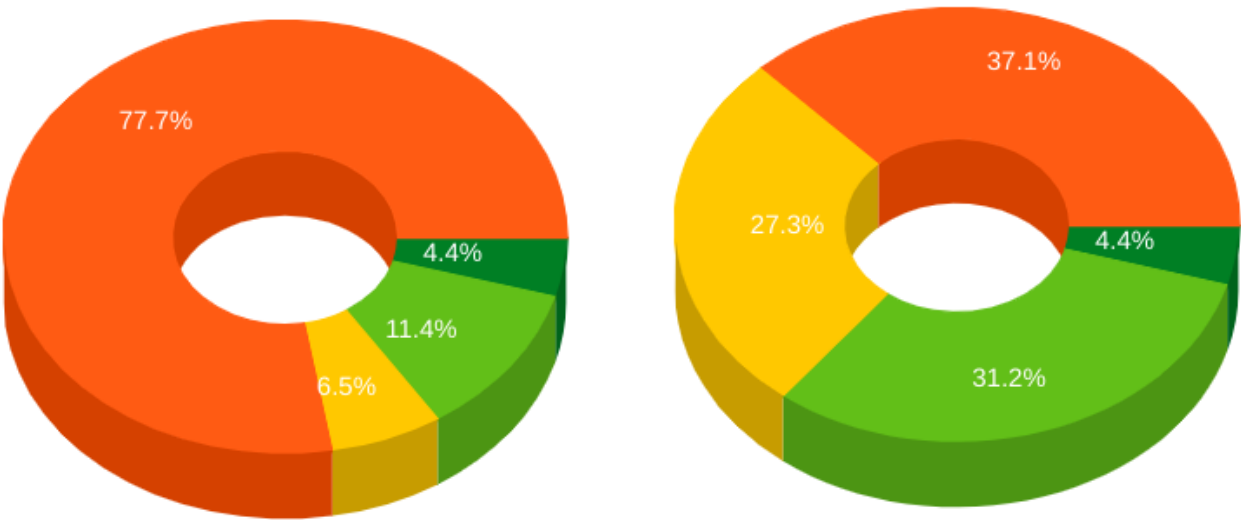
Before

After

The image shows a minimal change in C3 (32.8% to 32.6%), suggesting no significant impact on conceptual complexity after refactoring. Possible explanations are,

- **Limited Scope:** Refactoring efforts might not have directly targeted factors influencing C3, focusing on other aspects like coupling or God Class removal.
- The minor improvement

Books Web



Before

After

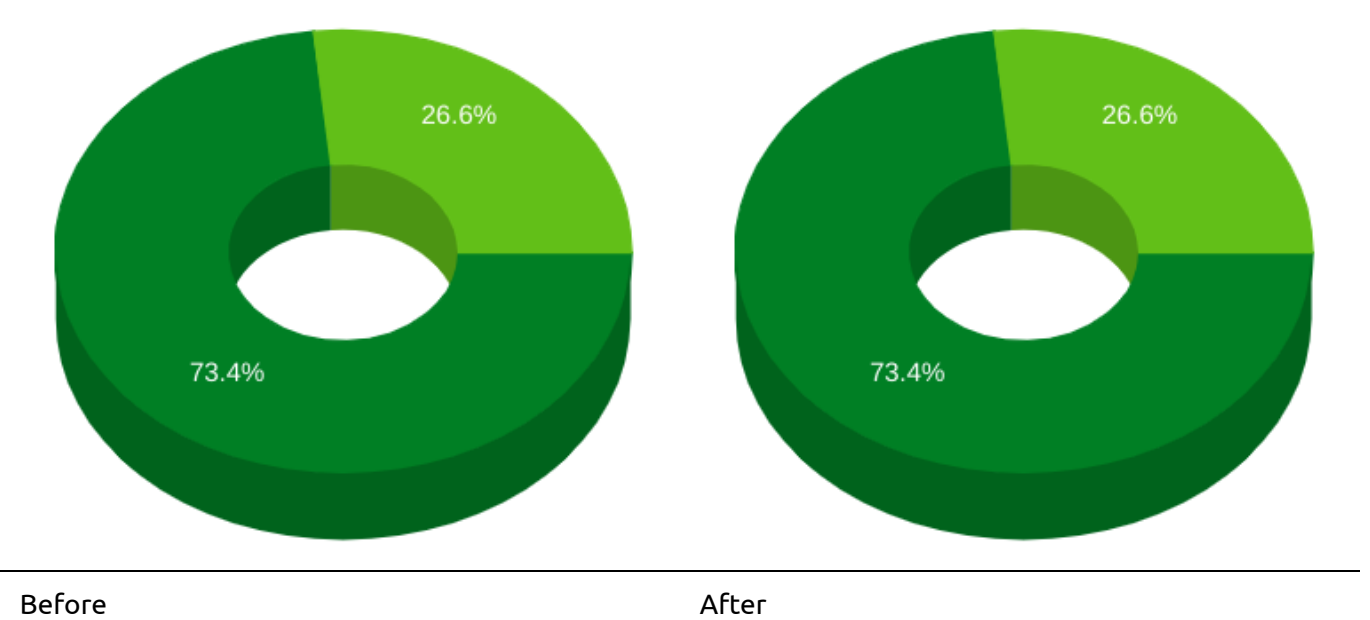
The Conceptual Complexity (C3) for the refactored classes `UserResource` and `BooksResource` decreased. There are several reasons why this might have happened:

- **Reduced Cyclomatic Complexity:** The refactoring may have reduced the cyclomatic complexity of the code. Cyclomatic complexity is a measure of the number of decision points in a piece of code, such as if statements and loops. By reducing the cyclomatic complexity, the code becomes easier to understand and reason about, which can lead to a lower C3 score.
- **Improved modularity:** The refactoring may have improved the modularity of the code. Modularity is the principle of breaking down a program into smaller, self-contained units. By improving the modularity of the code, the code becomes easier to understand and maintain, which can lead to a lower C3 score.
- **Reduced coupling:** The refactoring may have reduced the coupling between the classes. Coupling is a measure of how dependent one piece of code is on other pieces of code. By reducing the coupling between the classes, the code becomes easier to understand and change, which can lead to a lower C3 score.

Overall, the reduction in C3 is a positive sign. It suggests that the refactoring has made the code more modular, easier to understand, and less error-prone. This should make it easier to maintain and modify the code in the future.

6. Weighted Method Count (WMC)

Books Core

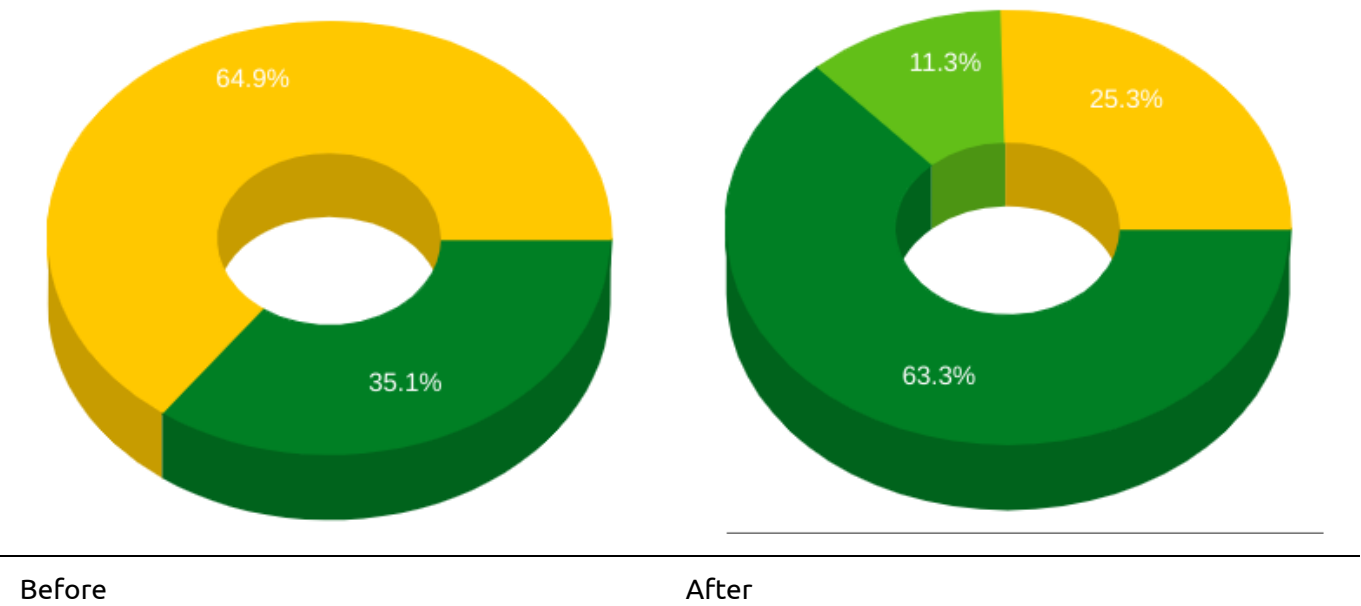


The image shows a very minor decrease in WMC in the "books-core" directory, from 26.6% to 26.2%. While not a significant change, it suggests a potential improvement in code complexity.

Possible Contributors:

- **Limited God Class Impact:** Splitting the `AppContext` class into smaller ones might have had a minimal effect on WMC, depending on the complexity within each new class.
- **Speculative Generality Removal:** Eliminating the `App` class and related unused code could have had a minor influence on WMC if their method count was low.

Books Web

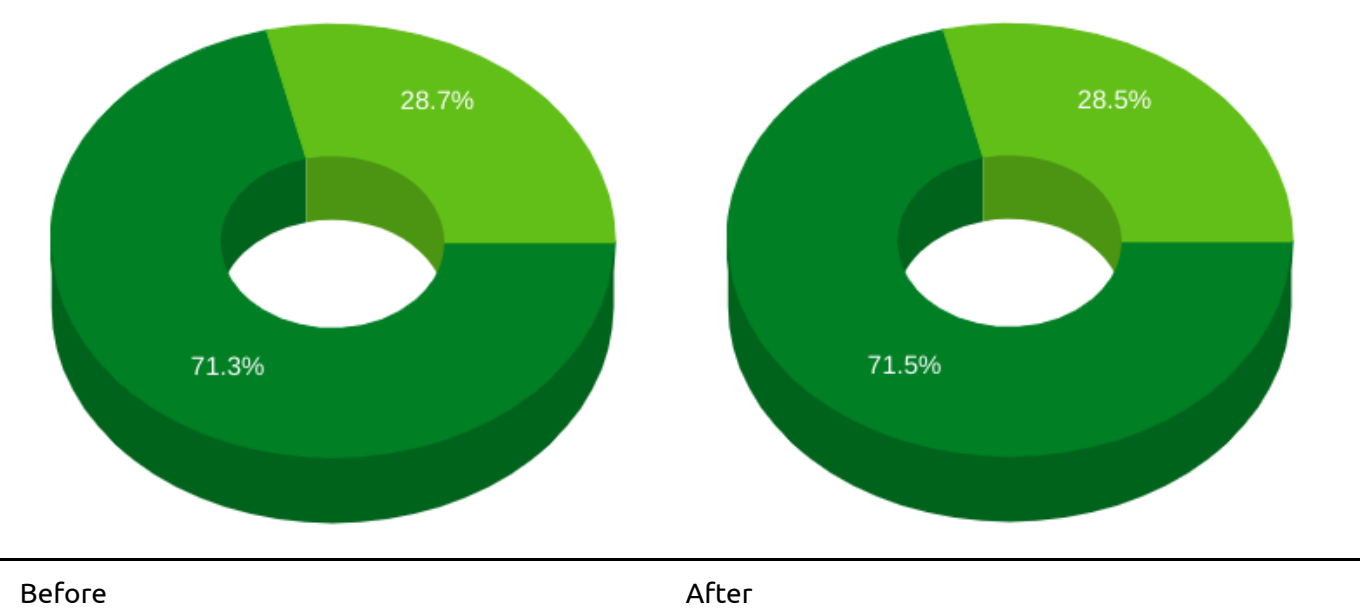


the Weighted Method Count (WMC) for the refactored classes `UserResource` and `BooksResource` decreased. There are several reasons why this might have happened:

- **Extracting smaller methods:** The refactoring may have involved extracting larger methods into smaller, more focused ones. This would make the code easier to understand and maintain, and it would also reduce the WMC metric.

7. Access to Foreign Data (ATFD)

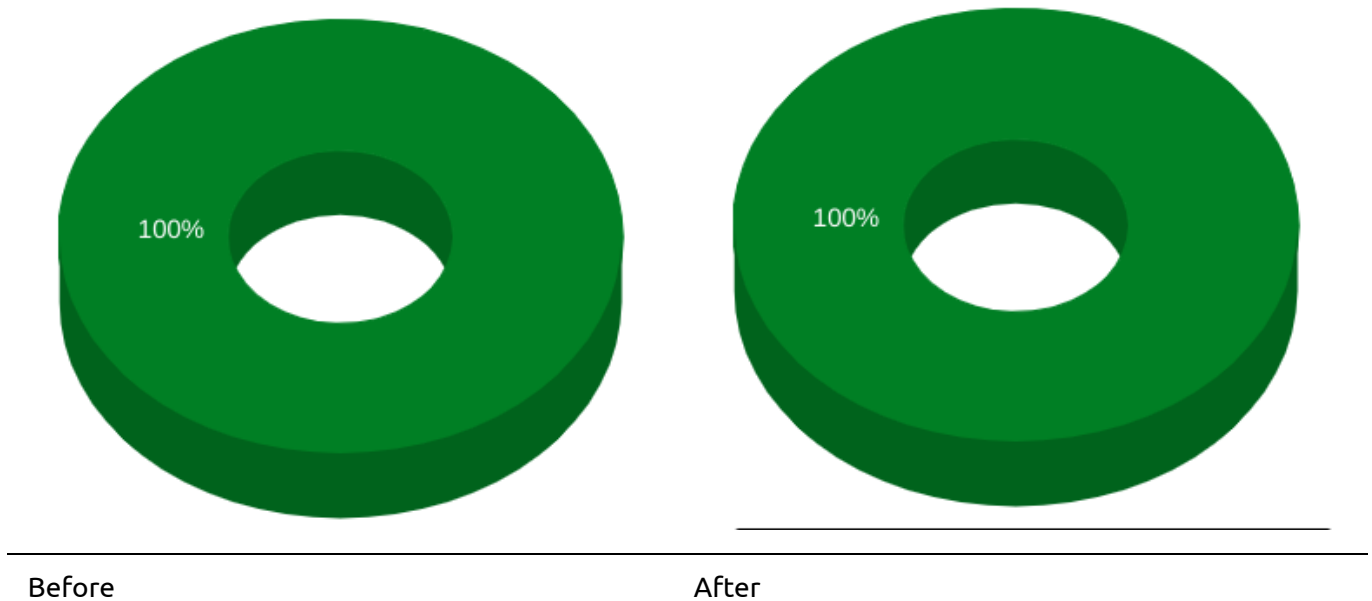
Books Core



The change in the ATFD metric is minimal, with only a 0.2% decrease. This suggests that the refactoring had some positive effect, although very slight, on reducing the access to foreign data within the codebase.

- **Speculative Generality Refactoring:** ** This refactoring is aimed at removing unnecessary complexities, which may not necessarily impact **ATFD** directly unless the complexities included inappropriate data access to other classes.
- **Feature Envy Refactoring:** ** Addressing this smell should ideally reduce **ATFD** because it involves improving encapsulation and reducing the need for a class to access data from another class.
- **God Class Refactoring:** Reducing a God Class into smaller, more focused classes should also have a significant impact on **ATFD**, as it suggests a more distributed and encapsulated design.

Books Web



The ATFD metric remained unchanged after refactoring the UserResource and BooksResource classes. This could be due to a few reasons:

- **No change in data access patterns:** The refactoring may not have changed the way the classes access foreign data. For example, if the classes were already only accessing data through well-defined interfaces or abstractions, then the refactoring would not have introduced any new foreign data dependencies.
- **Encapsulation of foreign data access:** The refactoring may have encapsulated the foreign data access within the classes themselves, rather than relying on external classes or methods. This would hide the implementation details of foreign data access and prevent changes to the classes from affecting the ATFD metric.