CS6.401 Software Engineering

Spring 2024

Project - 1

Project Title: Book Management

Submitted By Team_11

Task-1

Unveiling the Literary Landscape: A Comprehensive Analysis of the Book Management App Codebase and Subsystem Architecture.

The Book Management App serves as a literary haven, designed to enrich the reading experience through its intricate subsystems. This report delves into the codebase of the application, unraveling the intricacies of its major subsystems:

1. Book Addition & Display Subsystem
2. Bookshelf Management Subsystem
3. User Management Subsystem

The investigation involves the identification of pertinent classes, comprehensive documentation of their functionality and behavior within each subsystem, and the creation of UML diagrams employing Object-Oriented Programming (OOP) principles. Through these analyses, we aim to provide a detailed understanding of the system's architecture, relationships, and strengths, while also pinpointing potential areas for refinement. The use of plantUML/IntelliJ ensures a clear and visually intuitive representation of the system's structure and interactions. Additionally, any assumptions made during the documentation process are explicitly stated for transparency and clarity. This report ultimately aims to contribute insights for meaningful improvements and enhancements to the Book Management App.

1. **Book Addition & Display Subsystem:**

a. **Identification of Relevant Classes:**

- **AnonymousPrincipal, AppContext, AppResource, BaseResource:**
  - Classes associated with user authentication, application context management, and resource handling.
  - AppContext serves as a singleton, managing various services, including FacebookService and BookDataService.
  - BaseResource provides common methods for resource classes, ensuring authentication and base function checks.
- **Book, BookDao, BookDataService, BookImportAsyncListener, BookImportedEvent, BookResource:**
  - Classes related to book management, data services, and asynchronous book import events.
  - BookResource handles various book-related operations and events.
- **TagDao, TagDto, TagResource:**
  - Classes responsible for tagging functionality.
  - TagResource manages operations related to tags.
- **UserBook, UserBookDao, UserBookDto, UserBookTag:**
  - Classes dealing with user-book relationships.
  - UserBookDao provides data access methods for user-book relationships.

b. **Documentation of Functionality and Behavior:**

Table- 1: Functionality and Behavior of Book Addition & Display Subsystem

Class Name	Functionality	Behavior
AnonymousPrincipal	Represents an anonymous user, managing locale and timezone.	Initializes with default values and provides an identification constant.
AppContext	Manages the application context and services, ensures a singleton.	Provides methods for context management and ensures single-instance behavior.
AppResource	Handles general application resource-related operations.	Instantiates with an `AppContext` instance, provides resource-related methods.
BaseResource	Serves as a base class, implements authentication and base function checks.	Ensures authentication before resource access, checks and executes base functions.
Book	Represents a book with various attributes.	Stores book information and provide a method for string representation.
BookDao	Manages data access operations for the Book class.	Provides methods for getting books by ID or ISBN, allows book creation.
BookDataService	Integrates with external services for book data, manages book operations.	Initializes configuration, searches for books using external services, downloads thumbnails.

BookImportAsyncListener	Listens for asynchronous book import events.	Handles book import events with dependencies on AppContext, BookDao, and TagDao.
BookImportedEvent	Represents an event triggered when a book is imported.	Stores user and file information, provides a string representation.
BookResource	Manages book-related operations and interactions.	Instantiates with dependencies, provides methods for adding, updating, deleting, and listing books.
TagDao	Manages data access operations for tagging.	Provides methods for creating, updating, and retrieving tags, manages associations.
TagDto	Represents a data transfer object (DTO) for tags.	Stores information about a tag, including name and color.
TagResource	Manages tag-related operations.	Instantiates with a TagDao dependency, provides methods for adding, updating, deleting, and listing tags.
UserBook	Represents a user-book relationship.	Stores user, book, and timestamp information, implements hashCode, equals, and toString methods.
UserBookDao	Manages data access operations for user-book relationships.	Provides methods for creating, updating, deleting, and retrieving user-book relationships, supports criteria-based searches.
UserBookDto	Represents a data transfer object (DTO) for user books.	Stores information about a user book, including title, author, and timestamps.
UserBookTag	Represents the relationship between user books and tags.	Stores information about the user book and associated tag.

c. UML Diagrams:

Please refer to the attached UML diagrams created:

Click here to access the UML diagram generated by PlantUML.

d. Observations and Comments:

- The classes exhibit a clear and modular design.
- Dependencies between classes are well-managed.
- The **BookImportAsyncListener** and **BookImportedEvent** could benefit from additional documentation to clarify their roles and interactions.

e. Assumptions:

- The absence of method details assumes that the methods follow logical naming conventions.
- It is assumed that the asynchronous event handling mechanism is appropriately implemented.

2. Bookshelf Management Subsystem:

a. Identification of Relevant Classes:

- **AppContext, AppResource, BaseResource:**
  - Classes associated with user authentication, application context management, and resource handling.
  - AppContext serves as a singleton, managing various services, including FacebookService and BookDataService.
  - BaseResource provides common methods for resource classes, ensuring authentication and base function checks.
- **Book, BookDao, BookDataService, BookImportAsyncListener, BookImportedEvent, BookResource:**
  - Classes related to book management, data services, and asynchronous book import events.
  - BookResource handles various book-related operations and events.
- **SortCriteria, UserBookDao, UserBookCriteria:**
  - Classes dealing with User book criteria and filtering.
  - UserBookDao provides data access methods for user-book relationships.

b. Documentation of Functionality and Behavior:

i. **The classes outlined in Table 1 encapsulate both functionality and behavior. These classes include AppContext, AppResource, BaseResource, Book, BookDao, BookDataService, BookImportAsyncListener, BookImportedEvent, BookResource, and UserBookDao.**

Table- 2: Functionality and Behavior of Bookshelf Management Subsystem

Class Name	Functionality	Behavior
SortCriteria	Base class for Sorting.	Criteria for sorting a query's results by specifying the column index and the sorting order (ascending or descending).
UserBookCriteria	Represents criteria for searching user books	It includes fields for specifying the user ID, a search query, a read state, and a list of tag IDs to filter the search results.

c. UML Diagrams:

Please refer to the attached UML diagrams created:

Click here to access the UML diagram generated by PlantUML.

d. Observations and Comments:

- The classes exhibit a clear and modular design.
- Dependencies between classes are well-managed.

- A more logical naming convention is needed to understand the bookshelf related criteria.

e. **Assumptions:**

- The absence of method details assumes that the methods follow logical naming conventions.

3. User Management Subsystem:

a. **Identification of Relevant Classes:**

- **AppContext, AppResource, BaseResource, ConnectResource, FacebookService, QueryParam:**
  - Classes associated with user authentication, application context management, and resource handling.
  - AppContext serves as a singleton, managing various services, including FacebookService and BookDataService.
  - BaseResource provides common methods for resource classes, ensuring authentication and base function checks.
  - ConnectResource providesREST resource for managing connected applications.
  - QueryParam Represents query parameters used in HTTP requests.
- **AuthenticationToken and AuthenticationTokenDao,** which work together to manage authentication tokens in a database.
- **User, UserResource, UserApp, UserAppCreatedEvent, UserAppCreatedAsyncListener, UserContact:**
  - Classes related to user management, data services, and asynchronous user import events.
  - UserResource is for managing user-related operations.
  - UserApp Represents a connected application for a user.
  - UserContact represents a user contact.
- **UserDao, UserDto , UserAppDao, UserAppDto ,UserContactDao, UserContactDto, UserContactCriteria:**
  - Classes dealing with Data Access Objects and Data Transfer Objects.
  - UserContactCriteria Represents criteria for querying user contacts.

b. **Documentation of Functionality and Behavior:**

i. **The classes outlined in Table 1 encapsulate both functionality and behavior. These classes include AppContext, AppResource.**

Table- 3: Functionality and Behavior of User Management Subsystem

Class Name	Functionality	Behavior

AuthenticationToken	Represents an authentication token used for user authentication.	Contains properties like creationDate, userId, lastConnectionDate, and longLasted.
AuthenticationTokenDao	Provides methods to create, retrieve, update, and delete authentication tokens.	Handles operations related to user authentication and session management.
	Serves as a base class, implements authentication and base function checks.	Ensures authentication before resource access, checks and executes base functions.
ConnectResource	Provides methods for adding, updating, listing, and removing connected applications.	Handles contact list operations related to connected applications.
FacebookService	Manages Facebook-related functionalities, such as publishing actions, updating user data, and handling permissions.	Utilizes the Facebook API to perform these tasks.
QueryParam	Represents query parameters used in HTTP requests.	Contains a queryString and a parameterMap to handle query parameters.
User	Contains user-related properties like id, email, username, password, localeId, and more.	Used for user management and authentication.
UserApp	Represents a connected application associated with a user.	Contains information about the application, including access tokens and sharing settings.
UserAppCreatedAsyncListener	Represents an asynchronous listener for user application creation events.	Handles events related to the creation of user applications.
UserAppCreatedEvent	Represents an event that occurs when a user application is created.	Contains information about the created user application.
UserAppDao	Data access object (DAO) for managing user applications.	Provides methods for creating, retrieving, updating, and deleting user applications.
UserAppDto	Represents a data transfer object (DTO) for user applications.	Contains simplified information about user applications.
UserContact	Represents a user contact entity associated with an application.	Contains information such as email, external ID, full name, and more.
UserContactCriteria	Represents criteria for querying user contacts.	Contains parameters such as application ID, user ID, and query.
UserContactDao	Data access object (DAO) for managing user contacts.	Provides methods for creating, retrieving, updating, and deleting user contacts.
UserContactDto	Represents a data transfer object (DTO) for user contacts.	Contains simplified information about user contacts.

UserDao	Data access object (DAO) for managing users.	Provides methods for creating, retrieving, updating, and deleting users. Used for user-related operations.
UserDto	Represents a data transfer object (DTO) for users.	Contains simplified information about users.
UserResource	Represents a resource for managing user-related operations.	Contains methods for user registration, login, profile updates, and more.

c. UML Diagrams:

Please refer to the attached UML diagrams created:

Click here to access the UML diagram generated by PlantUML.

d. Observations and Comments:

- The classes exhibit a clear and modular design.
- Dependencies between classes are well-managed.
- The **UserAppCreatedAsyncListener** and **UserAppCreatedEvent** could benefit from additional documentation to clarify their roles and interactions.

e. Assumptions:

- The absence of method details assumes that the methods follow logical naming conventions.
- It is assumed that the asynchronous event handling mechanism is appropriately implemented.

Task 2a: Design Smells Detection

For the detection of design smells, we utilized Sonarqube, and other plugins available in various IDEs. While **Sonarqube** is a prominent tool, it's essential to note that no automated tool is perfect, and manual inspection is recommended for a more comprehensive analysis.

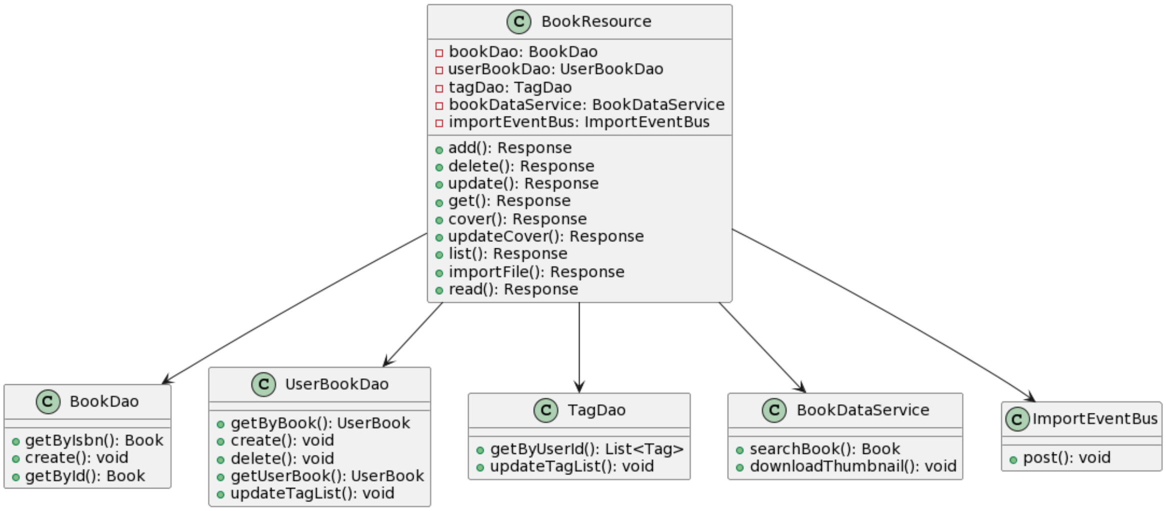
1. Design Smell

Where is the design smell?

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

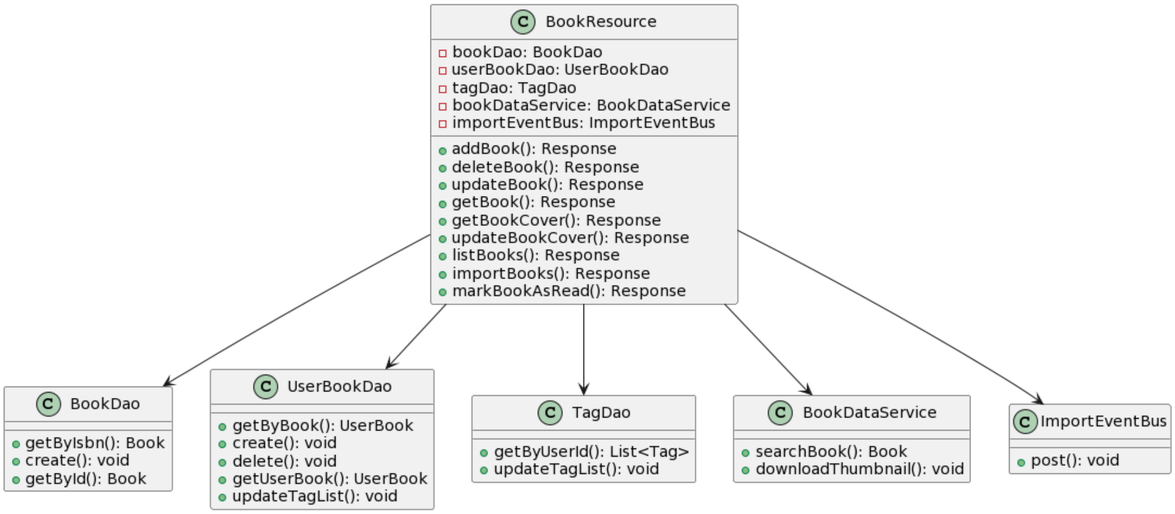
After reviewing the BookResource.java code, I identified several design smells that could be improved:

1. **Large Class:** The BookResource class has grown too large and handles many responsibilities, making it difficult to maintain and understand.
2. **Long Method:** Several methods within the class are quite long and perform multiple operations, such as add, update, and importFile. These methods violate the Single Responsibility Principle (SRP) and should be broken down into smaller, more focused methods.
3. **Primitive Obsession:** The code uses primitive types like boolean and long directly in method parameters and return types, which can lead to issues with nullability and type safety.
4. **Lack of Dependency Injection:** The class creates instances of DAOs and other services directly within its methods, which makes testing and maintenance harder. Dependency injection would improve modularity and testability.
5. **Magic Strings:** There are hardcoded strings throughout the code, such as "isbn" and "BookNotFound", which could be replaced with constants to avoid errors and make the code easier to maintain.



Refactoring

Based on the identified design smells, I will refactor the code by breaking down long methods, introducing dependency injection, and replacing magic strings with constants. Here's the refactored version of the BookResource.java:



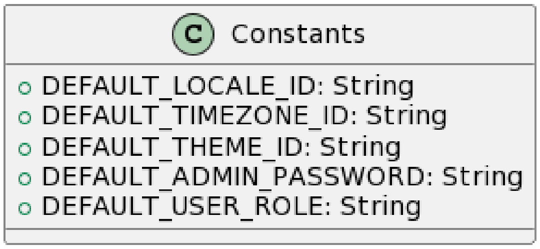
2. Design Smell

Where is the design smell?

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

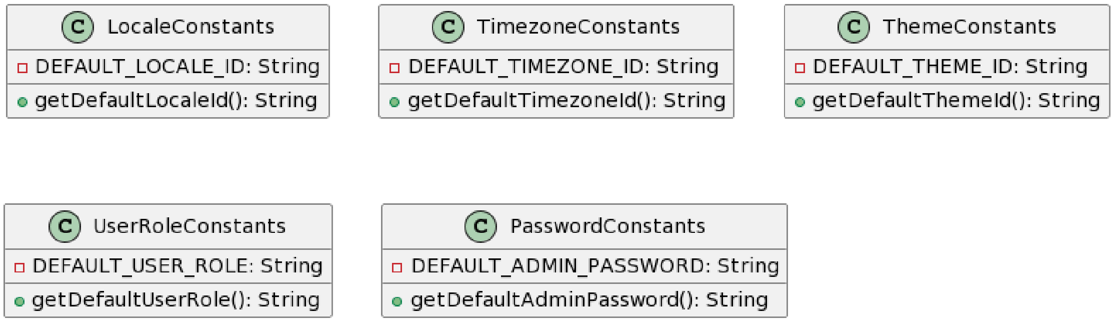
Design Smell Identification: The Constants class in the provided Java code is a typical example of a utility class that holds constant values. While this is not necessarily a design smell, it can lead to issues such as:

1. **Global State:** Since all constants are defined in a single class, they become part of the global state, which can make testing and maintaining the code more difficult.
2. **Inflexibility:** Adding or removing constants requires modifying the Constants class, which can be cumbersome, especially if there are many classes that depend on it.
3. **Lack of Encapsulation:** All constants are public, which means they can be accessed and potentially modified from anywhere in the application. This breaks encapsulation.


C Constants
DEFAULT_LOCALE_ID: String
DEFAULT_TIMEZONE_ID: String
DEFAULT_THEME_ID: String
DEFAULT_ADMIN_PASSWORD: String
DEFAULT_USER_ROLE: String

Refactoring:

To address the design smells identified, we can refactor the Constants class to use a more object-oriented approach. One way to do this is to create separate classes for each group of related constants, encapsulating them within those classes. Here's how you might refactor the Constants class:


C LocaleConstants
□ DEFAULT_LOCALE_ID: String
● getDefaultLocaleId(): String
C TimezoneConstants
□ DEFAULT_TIMEZONE_ID: String
● getDefaultTimezoneId(): String
C ThemeConstants
□ DEFAULT_THEME_ID: String
● getDefaultThemeId(): String
C UserRoleConstants
□ DEFAULT_USER_ROLE: String
● getDefaultUserRole(): String
C PasswordConstants
□ DEFAULT_ADMIN_PASSWORD: String
● getDefaultAdminPassword(): String

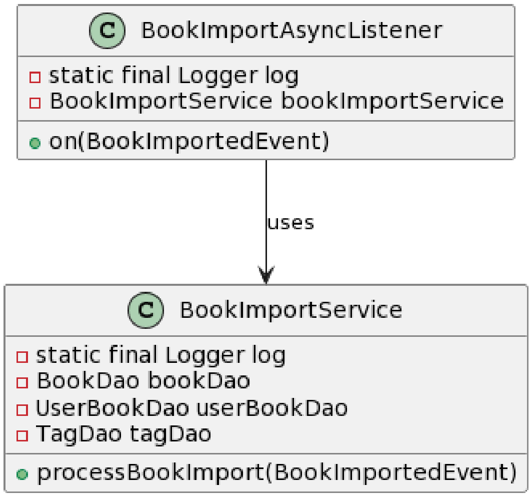
3. Design Smell

Where is the design smell?

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

Upon reviewing the BookImportAsyncListener class, I identified several design smells:

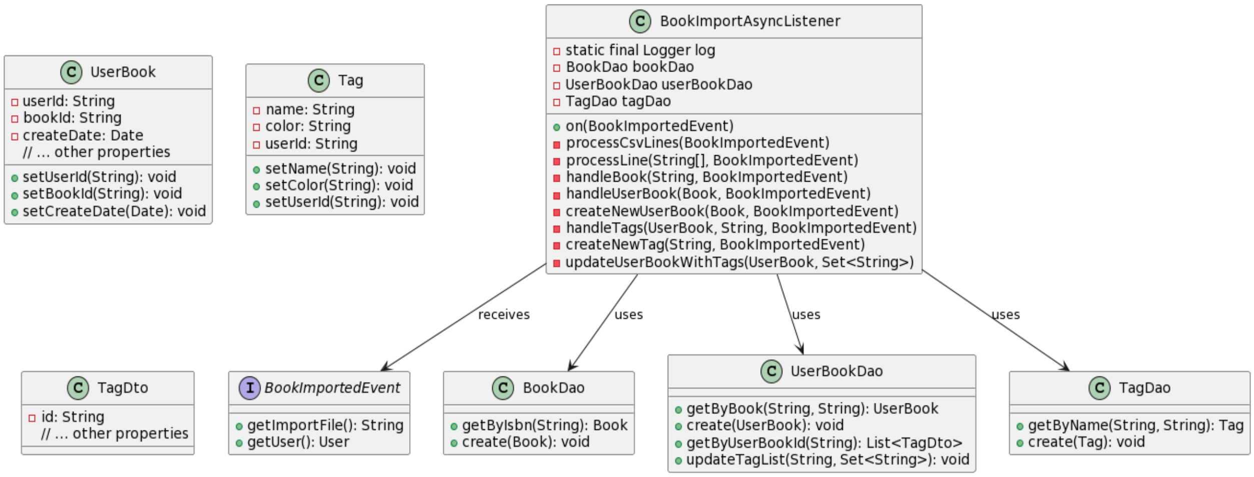
1. **Large Class:** The class has grown too large and is responsible for many things, including reading CSV files, creating books and tags, fetching data from APIs, and handling transactions.
2. **Long Method:** The on method is quite long and does a lot of work, making it hard to understand and maintain.
3. **Feature Envy:** Some methods within the class seem to be more interested in the data of other classes than their own data. For example, the on method uses data from BookImportedEvent.
4. **Data Clumps:** There are clumps of related data being passed around, such as the isbn, userBook, and tag objects.
5. **Switch Statements:** Although there are no explicit switch statements in the code, the logic for handling different types of books could potentially be simplified or abstracted away.



Refactoring:

Based on the identified design smells, I suggest the following refactoring steps:

1. **Extract Methods:** Break down the on method into smaller, more focused methods.
2. **Replace Data Clump with Object:** Encapsulate related data into objects to reduce parameter lists.
3. **Introduce Service Classes:** Move the responsibilities of reading CSV files, fetching data from APIs, and handling transactions into separate service classes. `



4. Design Smell

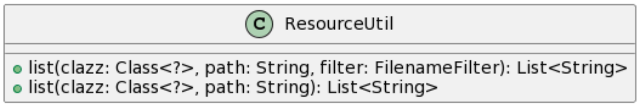
Where is the design smell?

books-core/src/main/java/com/sismics/util/ResourceUtil.java

Upon reviewing the ResourceUtil.java code, I identify several potential design smells:

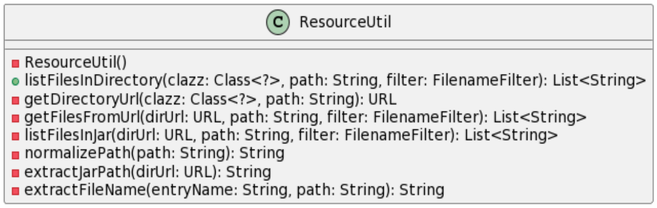
1. **Duplication:** There is duplicated logic in the list methods when handling the case where the path starts with a slash and when it doesn't.

2. **Long Method:** The list method is quite long and does several things, making it harder to understand and maintain.
3. **Feature Envy:** The list method seems to be more interested in the details of how to handle file paths and filtering than in the responsibilities of a utility class.
4. **Switch Statements:** Although there isn't a switch statement in the provided code, the method uses conditional statements to determine the protocol of the URL, which could be replaced with polymorphism.


C ResourceUtil
list(clazz: Class<?>, path: String, filter: FilenameFilter): List<String>
list(clazz: Class<?>, path: String): List<String>

Refactoring:

To address these design smells, we can refactor the code by extracting methods, using polymorphism, and simplifying the logic. Here's a refactored version of the ResourceUtil.java:


C ResourceUtil
ResourceUtil()
listFilesInDirectory(clazz: Class<?>, path: String, filter: FilenameFilter): List<String>
getDirectoryUrl(clazz: Class<?>, path: String): URL
getFilesFromUrl(dirUrl: URL, path: String, filter: FilenameFilter): List<String>
listFilesInJar(dirUrl: URL, path: String, filter: FilenameFilter): List<String>
normalizePath(path: String): String
extractJarPath(dirUrl: URL): String
extractFileName(entryName: String, path: String): String

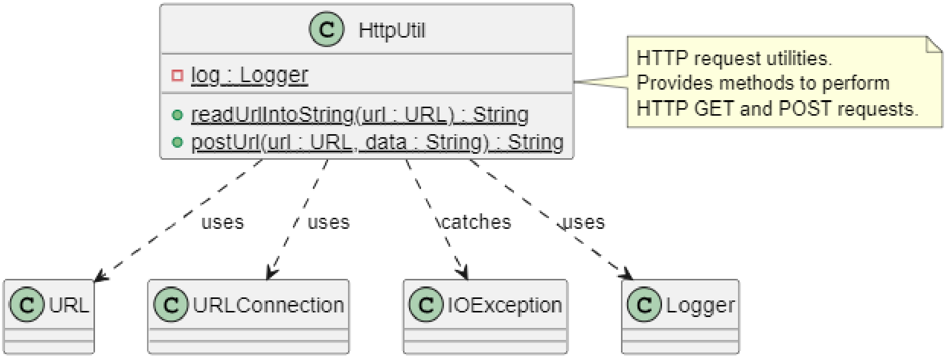
5. Design Smell

Where is the design smell?

books-core/src/main/java/com/sismics/util/HttpUtil.java

Upon reviewing the HttpUtil.java code, I identified the potential design smell:

1. **Resource Management:** Many resources in Java need be closed after they have been used. If they are not, the garbage collector cannot reclaim the resources' memory, and they are still considered to be in use by the operating system. Such resources are considered leaked, which can lead to performance issues.

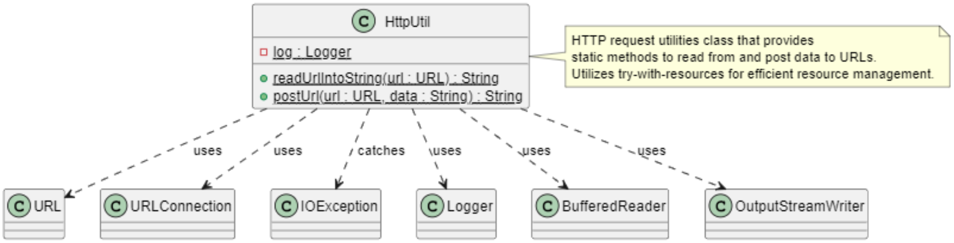

C HttpUtil
log : Logger
readUrlIntoString(url : URL) : String
postUrl(url : URL, data : String) : String
HTTP request utilities.
Provides methods to perform
HTTP GET and POST requests.
uses
uses
catches
uses
C URL
C URLConnection
C IOException
C Logger

Refactoring:

To address the design smells identified, we can refactor the HttpUtil class as follows:

1. **Try-with-Resources:** The BufferedReader is declared within the parentheses of the try statement, which means it will be automatically closed at the end of the block, even if exceptions are thrown. This removes the need for the final block used solely to close the resource.

Try-with-Resources:


HttpUtil
log : Logger
readUrlIntoString(url : URL) : String
postUrl(url : URL, data : String) : String
HTTP request utilities class that provides
static methods to read from and post data to URLs.
Utilizes try-with-resources for efficient resource management.
uses
uses
catches
uses
uses
uses
URL
URLConnection
IOException
Logger
BufferedReader
OutputStreamWriter

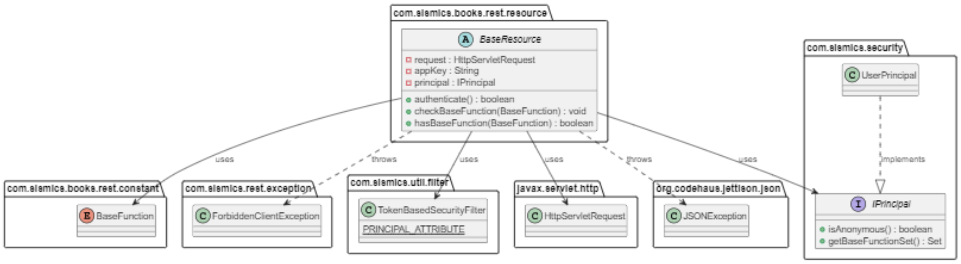
6. Design Smell

Where is the design smell?

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

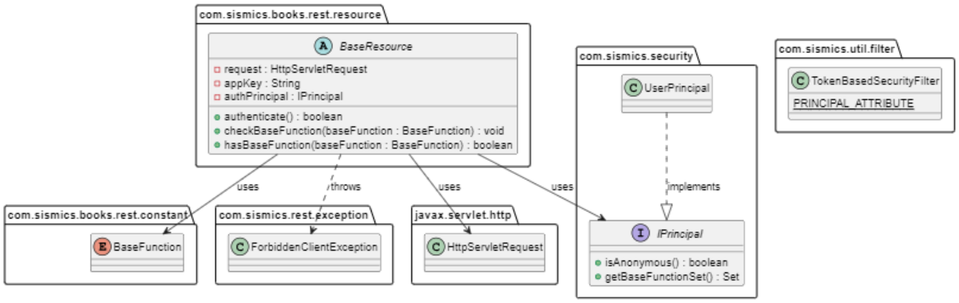
Upon reviewing the BaseResource.java **code, I identified these potential design smell:**

1. **Variable Shadowing:** Rename the variable "principal" in the authenticate() method which hides the field declared at line 37 of the **BaseResource** class. Shadowing class-level fields with local variables can lead to confusion and errors.
2. **Redundant Null Check:** Remove the unnecessary null check before the instanceof operation in the authenticate() and hasBaseFunction() methods. The instanceof operator already returns false when its left operand is null, making explicit null checks redundant. Remove this unnecessary null check; "instanceof" returns false for nulls.
3. **Unthrown Exception in Method Signature:** Remove the declaration of the org.codehaus.jettison.json.JSONException in the method signatures of checkBaseFunction and hasBaseFunction. This exception is not thrown within the bodies of these methods, so declaring it in the method signature is misleading and unnecessary.



Refactoring:

To address the design smells identified, we can refactor the BaseResource class as follows:



1. **Field Shadowing Correction:** The field previously named principal has been renamed to authPrincipal to prevent it from being overshadowed by the local variable with the same name in the authenticate() method. This adjustment ensures clarity between class-level and method-level variables representing the principal.
2. **Redundant Check Elimination:** The superfluous null checks preceding the instanceof operator have been eliminated from the authenticate() method. Since instanceof inherently returns false for null references, such preliminary checks are extraneous and have thus been removed to streamline the code.
3. **Exception Signature Streamlining:** The erroneous inclusion of JSONException in the `throws` clause of the checkBaseFunction() and hasBaseFunction() methods has been rectified. As these methods do not generate this exception, its declaration in the method signature was unwarranted and has been excised.

7. Design Smell

Where is the design smell?

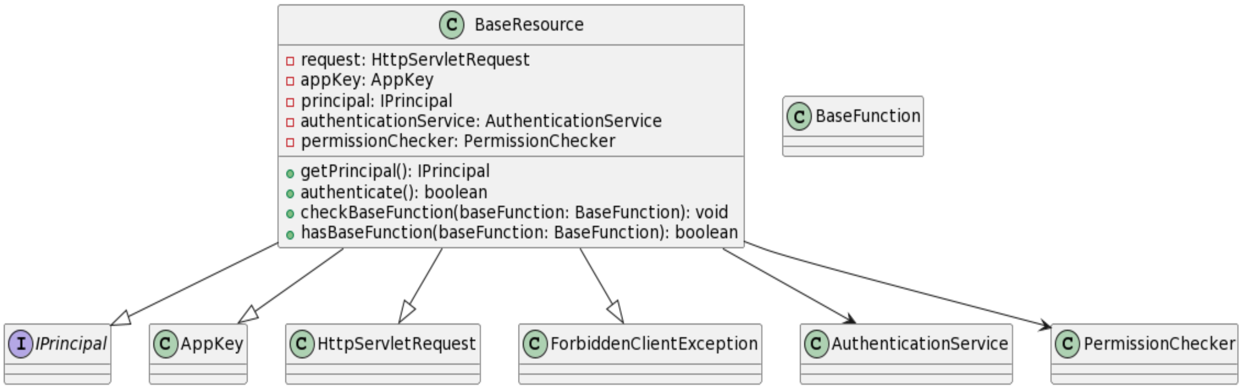
books-web/src/main/java/com/sismics/books/rest/resource/BaseResource.java

After reviewing the BaseResource.java file, I can identify several potential design smells:

1. **Large Class:** The BaseResource class seems to be doing too much. It handles authentication, checks permissions, and manages the principal object. This violates

the Single Responsibility Principle (SRP), which states that a class should have only one reason to change.

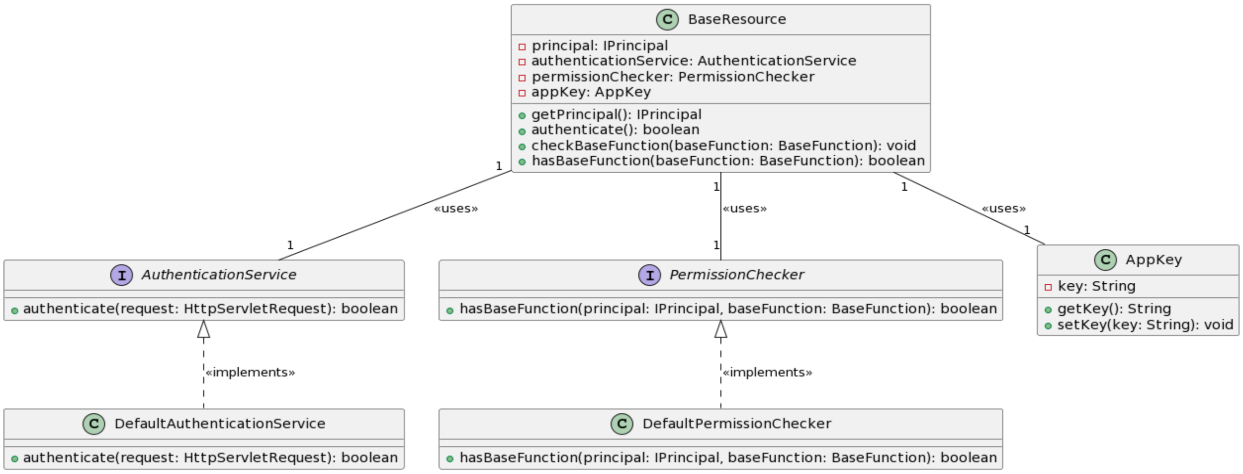
2. **Feature Envy:** The methods authenticate(), checkBaseFunction(), and hasBaseFunction() seem to be more interested in the state of other objects than in their own state. They are accessing and manipulating the state of the request and principal objects, which could be seen as a sign of feature envy.
3. **Primitive Obsession:** The appKey field is a String, which is a primitive obsession. It might be better to encapsulate this value within a class that provides behavior related to the application key.
4. **Lack of Encapsulation:** The principal field is protected, which means it can be accessed and potentially modified by subclasses. This breaks encapsulation and could lead to unexpected behavior if not handled carefully.
5. **Magic Strings:** The checkBaseFunction() and hasBaseFunction() methods use the name() method of the BaseFunction enum to get its string representation. This could be considered a magic string, as it's hardcoded and not easily maintainable.


BaseResource
request: HttpServletRequest
appKey: AppKey
principal: IPrincipal
authenticationService: AuthenticationService
permissionChecker: PermissionChecker
getPrincipal(): IPrincipal
authenticate(): boolean
checkBaseFunction(baseFunction: BaseFunction): void
hasBaseFunction(baseFunction: BaseFunction): boolean
BaseFunction
IPrincipal
AppKey
HttpServletRequest
ForbiddenClientException
AuthenticationService
PermissionChecker

Refactoring

Now, let's refactor the code to address these issues:

1. We can extract the authentication logic into a separate AuthenticationService class to adhere to SRP.
2. We can create a PermissionChecker class to handle permission checks, reducing the responsibility of the BaseResource.
3. We can encapsulate the appKey within a class that provides behavior related to the application key.
4. We can make the principal field private and provide getter methods to ensure encapsulation.
5. We can avoid magic strings by using constants or enums for the base function names.


BaseResource
principal: IPrincipal
authenticationService: AuthenticationService
permissionChecker: PermissionChecker
appKey: AppKey
getPrincipal(): IPrincipal
authenticate(): boolean
checkBaseFunction(baseFunction: BaseFunction): void
hasBaseFunction(baseFunction: BaseFunction): boolean
1
«uses»
1
1
«uses»
1
1
«uses»
1
AuthenticationService
authenticate(request: HttpServletRequest): boolean
PermissionChecker
hasBaseFunction(principal: IPrincipal, baseFunction: BaseFunction): boolean
AppKey
key: String
getKey(): String
setKey(key: String): void
«implements»
«implements»
DefaultAuthenticationService
authenticate(request: HttpServletRequest): boolean
DefaultPermissionChecker
hasBaseFunction(principal: IPrincipal, baseFunction: BaseFunction): boolean

8. Design Smell

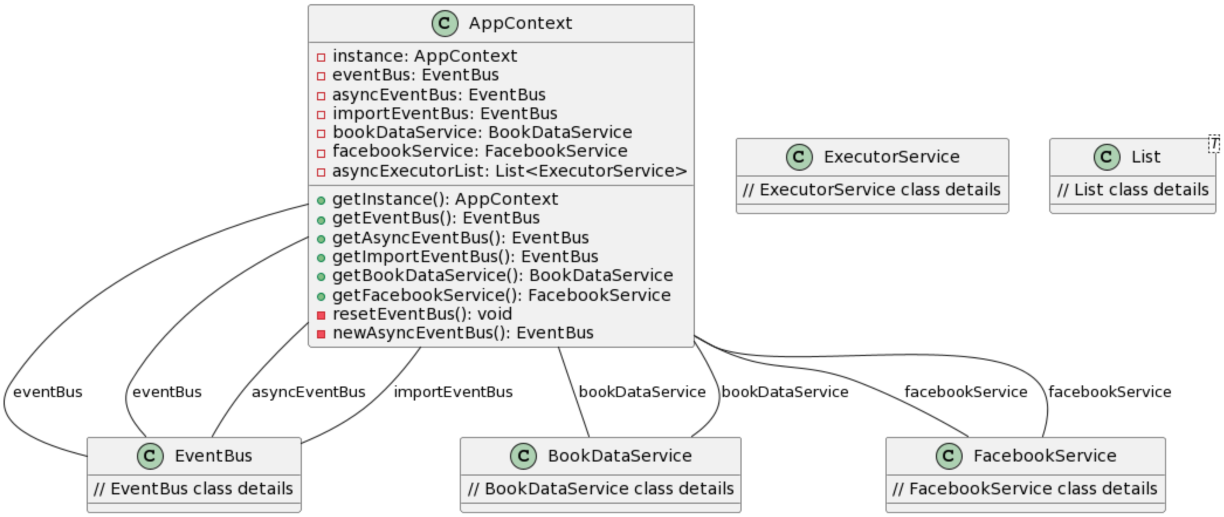
Where is the design smell?

books-core/src/main/java/com/sismics/books/core/model/context/AppContext.java

Upon reviewing the AppContext.java code, I identify several potential design smells:

1. **Singleton Pattern Misuse:** The `AppContext` class uses a singleton pattern, which is generally discouraged because it can lead to tight coupling between components and make testing difficult. It also hides dependencies and makes the code harder to reason about.
2. **God Object:** The `AppContext` class seems to be doing too much. It manages the creation and lifecycle of various services and event buses, which could be considered responsibilities that should be separated into different classes or modules.
3. **Static Method:** The `getInstance()` method is static, which means it cannot be overridden or mocked in tests, making unit testing challenging.
4. **Hardcoded Values:** The `newAsyncEventBus()` method has hardcoded values for the thread pool executor configuration. This makes the code less flexible and harder to maintain.
5. **Mutable Static State:** The `asyncExecutorList` is a mutable static field, which can lead to issues with thread safety and state management.

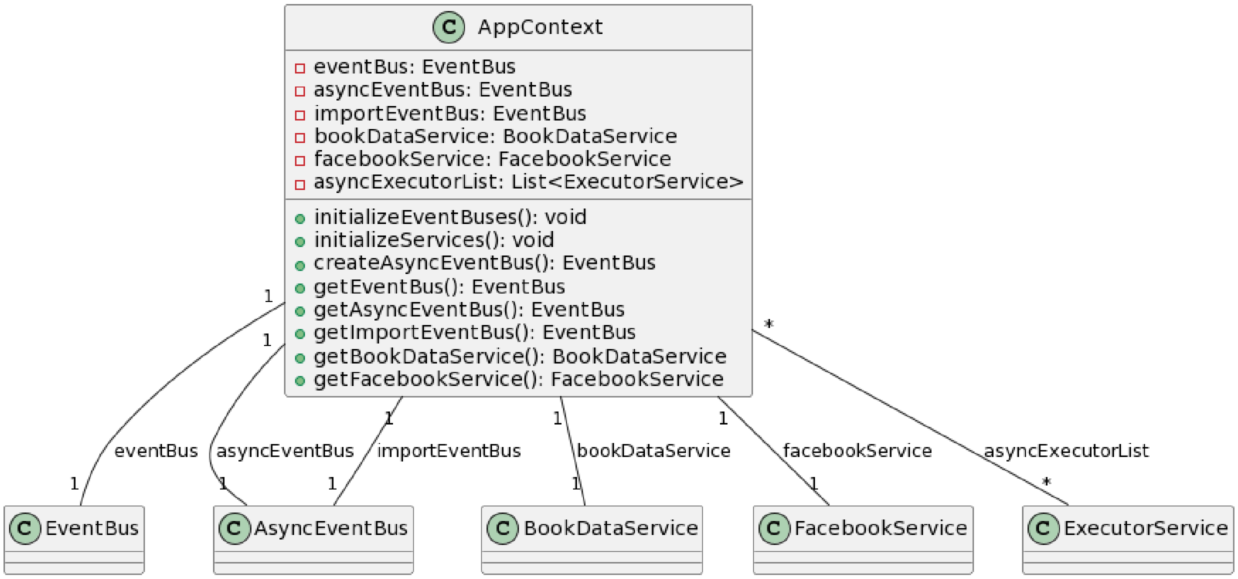
Data clumps!!


C AppContext
instance: AppContext
eventBus: EventBus
asyncEventBus: EventBus
importEventBus: EventBus
bookDataService: BookDataService
facebookService: FacebookService
asyncExecutorList: List<ExecutorService>
getInstance(): AppContext
getEventBus(): EventBus
getAsyncEventBus(): EventBus
getImportEventBus(): EventBus
getBookDataService(): BookDataService
getFacebookService(): FacebookService
resetEventBus(): void
newAsyncEventBus(): EventBus
C ExecutorService
// ExecutorService class details
C List
T
// List class details
eventBus
eventBus
asyncEventBus
importEventBus
bookDataService
bookDataService
facebookService
facebookService
C EventBus
// EventBus class details
C BookDataService
// BookDataService class details
C FacebookService
// FacebookService class details

Refactoring:

To address the design smells identified, we can refactor the AppContext class as follows:

1. Remove the singleton pattern and allow multiple instances of AppContext.
2. Separate the concerns of creating and managing services and event buses into dedicated classes or methods.
3. Replace the static getInstance() method with a factory method or dependency injection.
4. Parameterize the thread pool executor configuration to avoid hardcoding values.
5. Make asyncExecutorList an instance variable and manage its lifecycle properly.


C AppContext
eventBus: EventBus
asyncEventBus: EventBus
importEventBus: EventBus
bookDataService: BookDataService
facebookService: FacebookService
asyncExecutorList: List<ExecutorService>
initializeEventBuses(): void
initializeServices(): void
createAsyncEventBus(): EventBus
getEventBus(): EventBus
getAsyncEventBus(): EventBus
getImportEventBus(): EventBus
getBookDataService(): BookDataService
getFacebookService(): FacebookService
1
1
1
1
1
*
eventBus
asyncEventBus
importEventBus
bookDataService
facebookService
asyncExecutorList
1
1
1
1
1
*
C EventBus
C AsyncEventBus
C BookDataService
C FacebookService
C ExecutorService

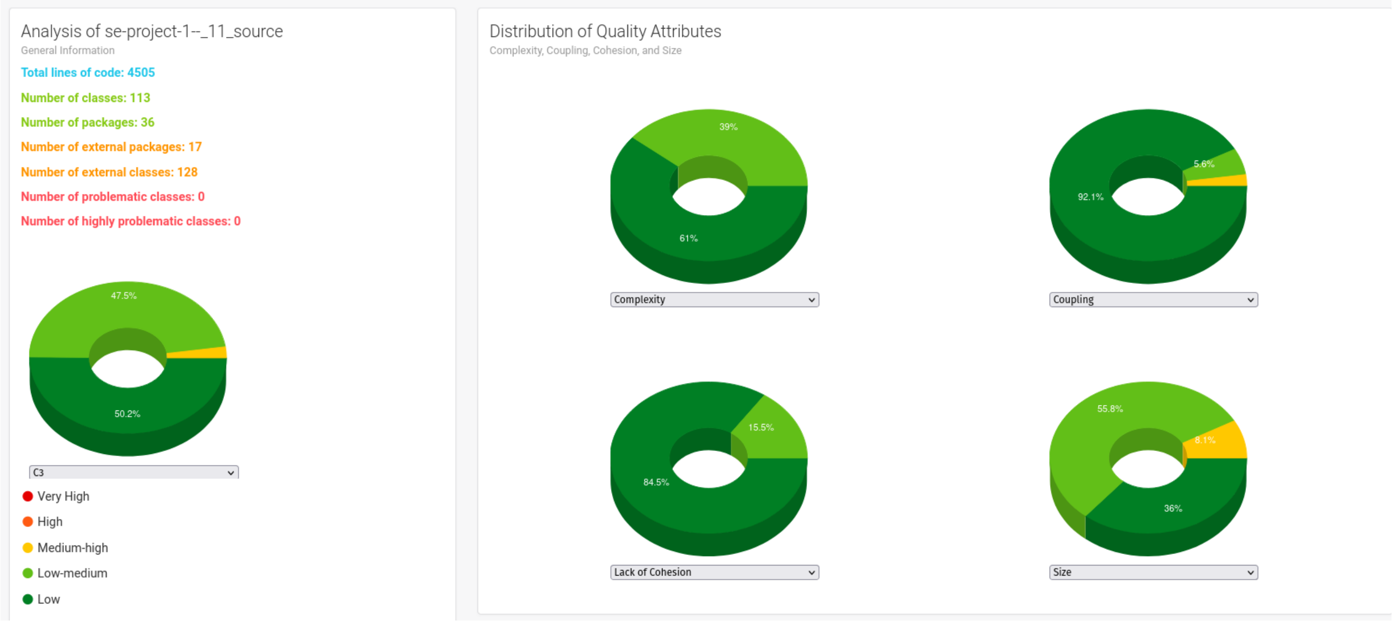
Task 2b: Code Metrics Analysis

Tools Used:

We have used popular tools like **SonarQube, CodeMR** and **Designite Java** to compute the code quality metrics on the Books project.

SonarQube provided a rich set of metrics with associated metrics and visualizations. ***CodeMR*** along with the codemetric, generates comprehensive visualizations on it. ***Designite*** can be used to find out additional code quality metrics like NC (Number of Children of Class), DIT (Depth of Inheritance Tree of Class) etc.

CodeMR Dashboard



Analysis of se-project-1--_11_source

General Information

Total lines of code: 4505

Number of classes: 113

Number of packages: 36

Number of external packages: 17

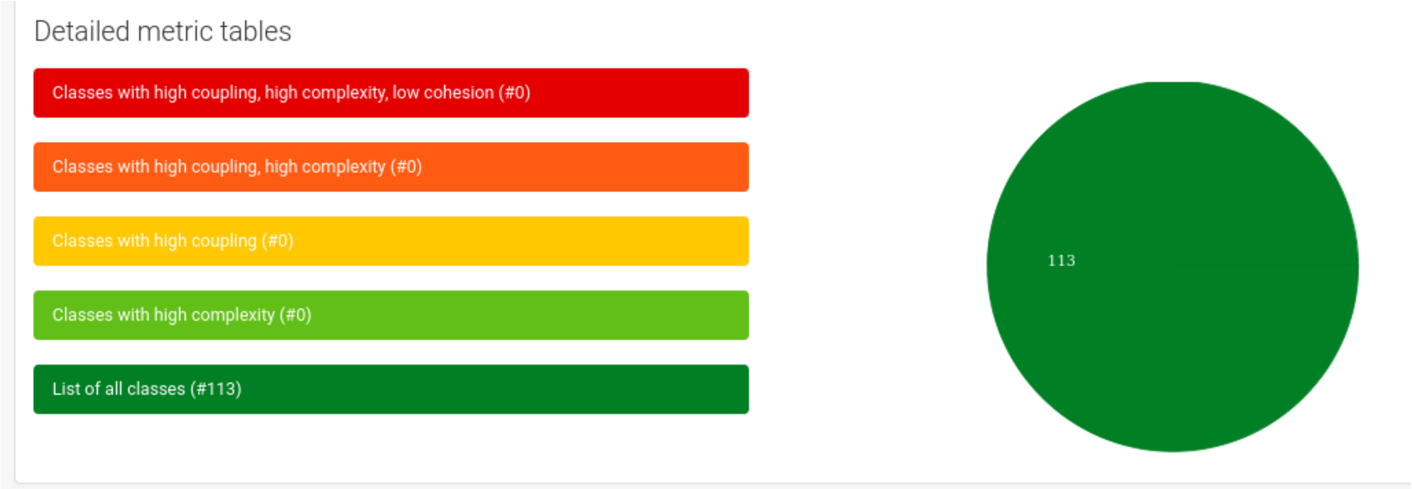
Number of external classes: 128

Number of problematic classes: 0

Number of highly problematic classes: 0

Distribution of Quality Attributes

Complexity, Coupling, Cohesion, and Size



Detailed metric tables

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

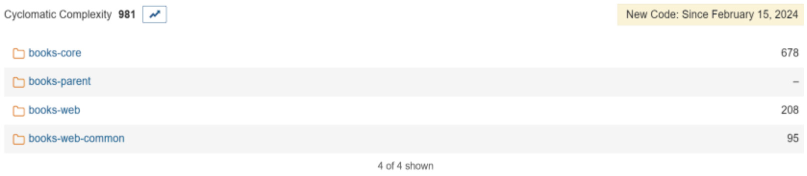
Classes with high coupling, high complexity (#0)

Classes with high coupling (#0)

Classes with high complexity (#0)

List of all classes (#113)

1. **Cyclomatic Complexity:**
   Cyclomatic complexity (CYC) is a software metric used to determine the complexity of a program. Cyclomatic complexity is a count of the number of decisions in the source code. The higher the count, the more complex the code. Programs with lower Cyclomatic complexity are easier to understand and less risky to modify.

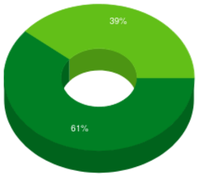


Cyclomatic Complexity **981**

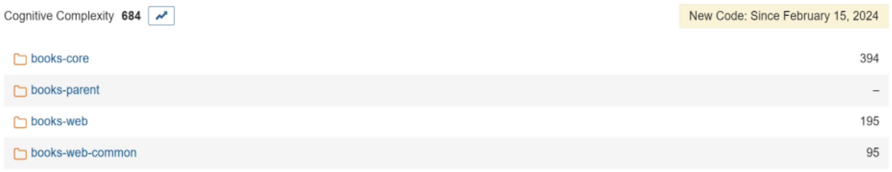
New Code: Since February 15, 2024

books-core	678
books-parent	–
books-web	208
books-web-common	95

4 of 4 shown



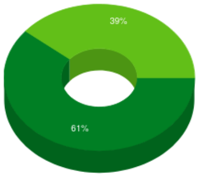
2. **Cognitive Complexity**
   Cognitive Complexity is a measure of how hard it is to understand a given piece of code—e.g., a function, a class, etc. Unlike Cyclomatic Complexity, which determines how difficult your code will be to test, Cognitive Complexity tells you how difficult your code will be to read and understand.



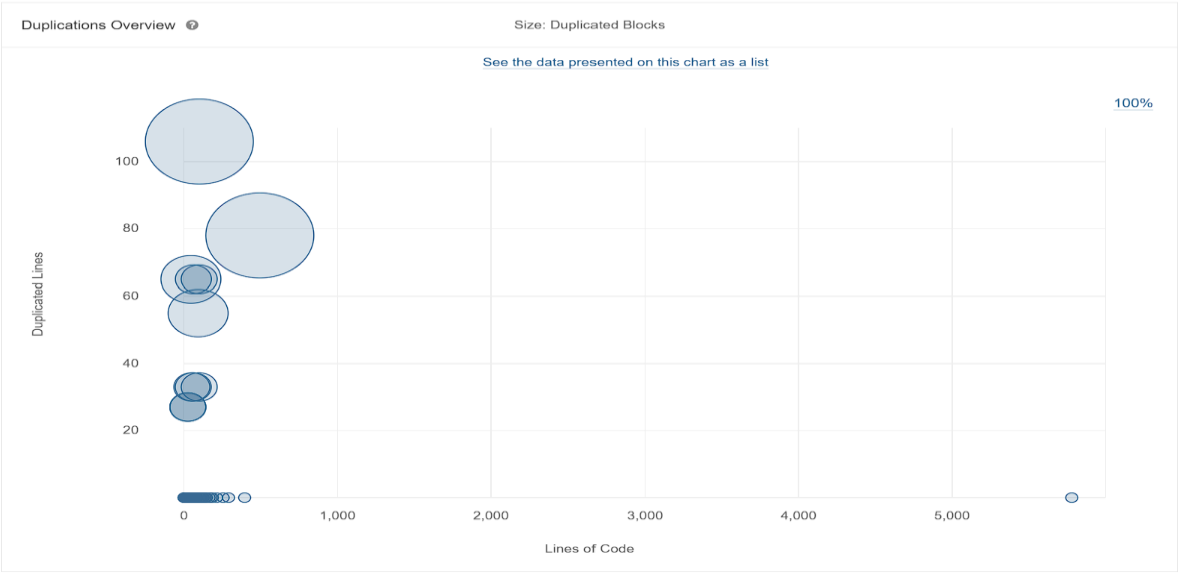
Cognitive Complexity **684**

New Code: Since February 15, 2024

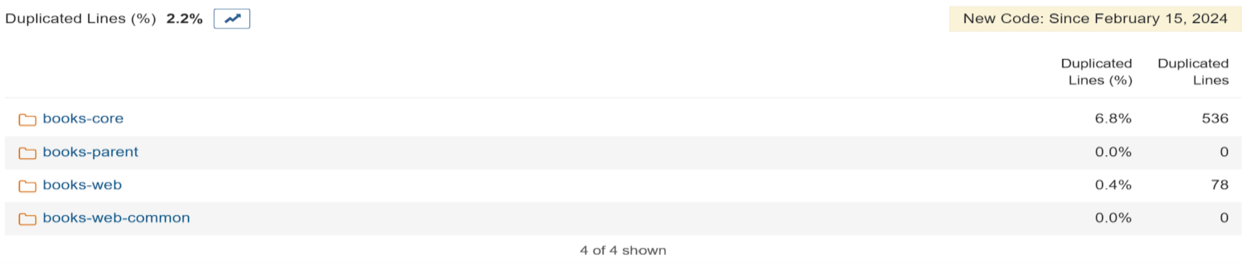
books-core	394
books-parent	–
books-web	195
books-web-common	95



3. **Code Duplication:**
   Same block of code repeating multiple times is the most basic form of Code Duplication.



Duplicated Lines: Number of physical lines duplicated.



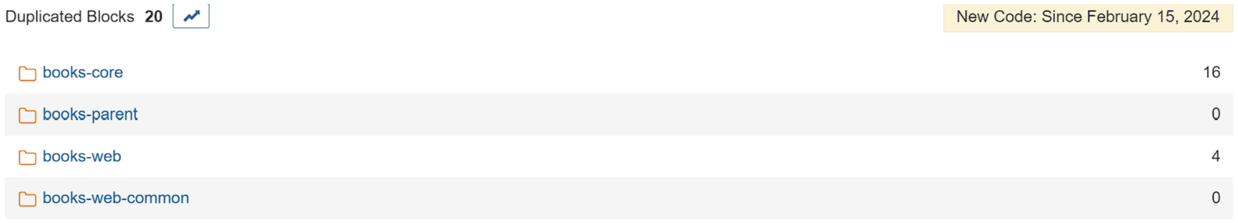
Duplicated Lines (%) **2.2%**

New Code: Since February 15, 2024

	Duplicated Lines (%)	Duplicated Lines
books-core	6.8%	536
books-parent	0.0%	0
books-web	0.4%	78
books-web-common	0.0%	0

4 of 4 shown

Duplicated Blocks - Number of duplicated blocks participating in duplication.

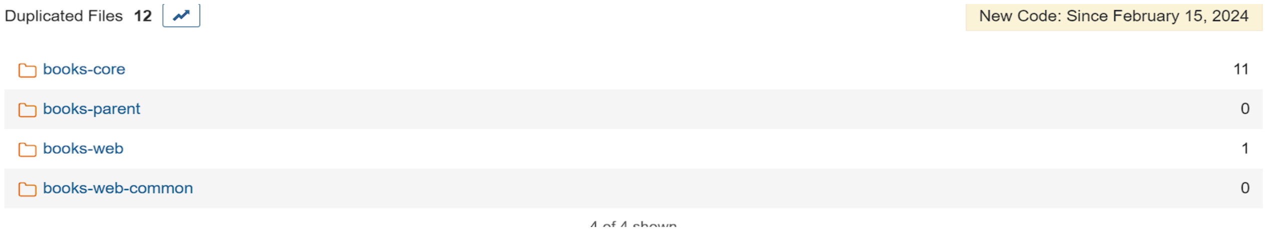


Duplicated Blocks **20**

New Code: Since February 15, 2024

books-core	16
books-parent	0
books-web	4
books-web-common	0

Duplicated Files - Number of files containing duplicated lines or blocks.



Duplicated Files **12**

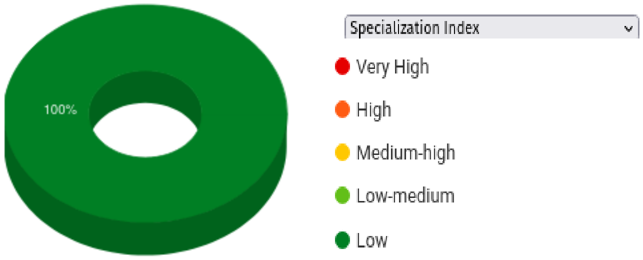
New Code: Since February 15, 2024

books-core	11
books-parent	0
books-web	1
books-web-common	0

4 of 4 shown

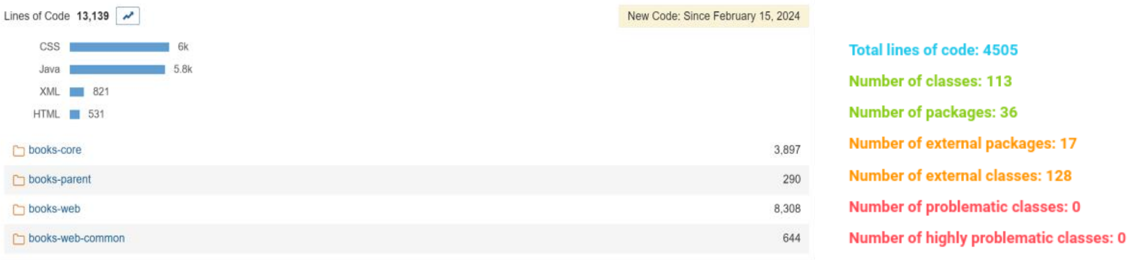
- *Implications:* High code duplication can result in inconsistencies and increase maintenance effort. Identifying and removing duplicate code enhances code quality.

4. **Specialization Index:**
   The Specialization Index metric measures the extent to which subclasses override their ancestors classes. This index is the ratio between the number of overridden methods and total number of methods in a Class, weighted by the depth of inheritance for this class.



5. **Lines of Code:**
   Source lines of code (SLOC), also known as lines of code (LOC), is a software metric used to measure the size of a computer program by counting the number of lines in the text of the program's source code. SLOC is typically used to predict the amount of effort that will

be required to develop a program, as well as to estimate programming productivity or maintainability once the software is produced.


Lines of Code 13,139

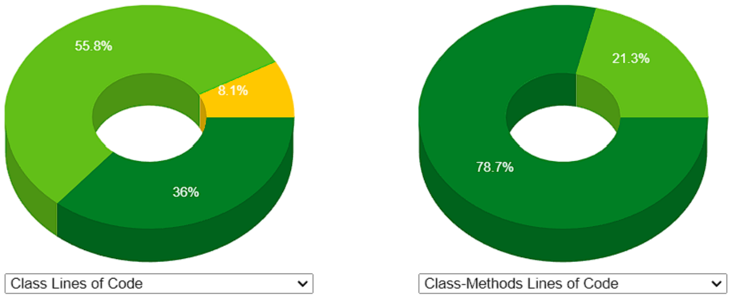
New Code: Since February 15, 2024

CSS	6k
Java	5.8k
XML	821
HTML	531

books-core	3,897
books-parent	290
books-web	8,308
books-web-common	644

Total lines of code: 4505
Number of classes: 113
Number of packages: 36
Number of external packages: 17
Number of external classes: 128
Number of problematic classes: 0
Number of highly problematic classes: 0

Lines of Code report generated by SonarQube

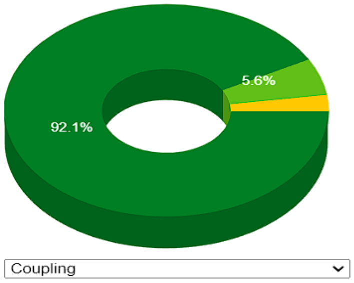


Class and Class-Methods Lines of Code percentage pie chart generated by CodeMR.

- *Implications:* Large codebase can be challenging to manage. While not a direct indicator of issues, it highlights areas that might need further investigation.

6. Coupling Between Objects:

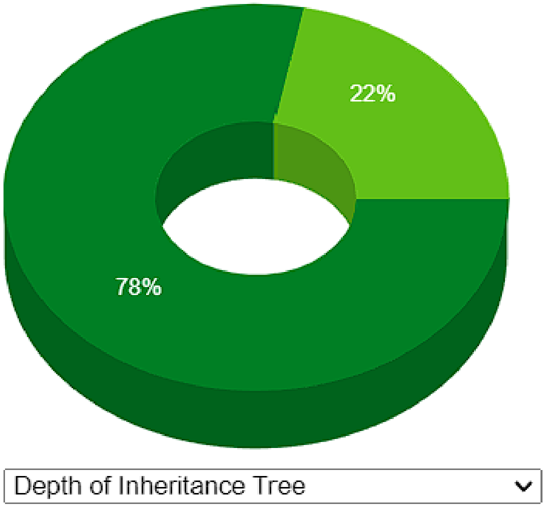
Class coupling is a measure of how many classes a single class uses. A high number is bad and a low number is usually good with this metric. Class coupling has been shown to be an accurate predictor of software failure.



- *Implications:* High coupling signifies increased interdependence between classes, making the system less flexible. A refactoring strategy may be needed to reduce coupling.

7. Depth of Inheritance Tree:

Depth of Inheritance Tree (DIT) is the maximum length of a path from a class to a root class in the inheritance structure of a system. DIT measures how many super-classes can affect a class. DIT is only applicable to object-oriented systems.



- *Implications:* Deep inheritance trees can result in complex hierarchies. Simplifying the inheritance structure might enhance code readability and maintainability.

Detailed Evaluation report by CodeMR and Designite tools can be accessed here.

Task 3a Design Smells Refactoring

In Task 2a, we identified and analyzed 7+ design smells within the existing codebase. Now, the objective is to rectify these issues through code refactoring without fundamentally altering the structure of the code.

2a_1-Design Smell

Where is the design smell?

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

Design Smell Description:
This issue is raised when Sonar considers that a method is a 'Brain Method'.

A Brain Method is a method that tends to centralize its owner's class logic and generally performs too many operations. This can include checking too many conditions, using lots of variables, and ultimately making it difficult to understand, maintain and reuse.

It is characterized by high LOC number, high cyclomatic and cognitive complexity, and a large number of variables being used.

Impact: Brain Methods are often hard to cover with tests, because of their deep nesting, and they are error-prone, because of the many local variables they usually introduce. Such methods will be very hard to read and understand for anyone outside who created them, and therefore hard to maintain and fix if bugs get spotted.

They also enable code duplication since the method itself can hardly be reused anywhere else.

Refactoring Strategy:
The common approach is to identify fragments of the method's code that deal with a specific responsibility and extract them to a new method. This will make each method more readable, easy to understand and maintain, easier to test, and more prone to be reused.
In this paper, the authors describe a systematic procedure to refactor this type of code smell: "Assessing the Refactoring of Brain Methods".

Pull Request/Issue Link: https://github.com/serc-courses/se-project-1--_11/pull/26

2a_2-Design Smell

Where is the design smell?

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

Design Smell Description: The design smell in the code is the inclusion of a hard-coded password in the constants file, which is considered bad practice for security reasons. Storing sensitive information, such as passwords, directly in the source code increases the risk of exposure and compromises secure.

Refactoring Strategy: To address this issue, it is recommended to separate sensitive information like passwords from the codebase and use a more secure approach, such as storing them in a configuration file or using a secure credential management system.

Pull Request/Issue Link: https://github.com/serc-courses/se-project-1--_11/pull/18

2a_3-Design Smell

Where is the design smell?

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

Design Smell Description: Cognitive Complexity is a measure of how hard the control flow of a method is to understand. Methods with high Cognitive Complexity will be difficult to maintain.

Exceptions: equals and hashCode methods are ignored because they might be automatically generated and might end up being difficult to understand, especially in the presence of many fields.

Pull Request Link: https://github.com/serc-courses/se-project-1--_11/issues/14

2a_4-Design Smell

Where is the design smell?

books-core/src/main/java/com/sismics/util/ResourceUtil.java

Design Smell Description: Cognitive Complexity is a measure of how hard the control flow of a method is to understand. Methods with high Cognitive Complexity will be difficult to maintain.

Exceptions: equals and hashCode methods are ignored because they might be automatically generated and might end up being difficult to understand, especially in the presence of many fields.

Pull Request/Issue Link: https://github.com/serc-courses/se-project-1--_11/issues/24

2a_5-Design Smell

Where is the design smell? https://github.com/serc-courses/se-project-1--_11/issues/12

Design Smell Description: Many resources in Java need be closed after they have been used. If they are not, the garbage collector cannot reclaim the resources' memory, and they are still considered to be in use by the operating system. Such resources are considered leaked, which can lead to performance issues.

Refactoring: To address the design smells identified, we can refactor the HttpUtil class as follows:

Try-with-Resources: The BufferedReader is declared within the parentheses of the try statement, which means it will be automatically closed at the end of the block, even if exceptions are thrown. This removes the need for the final block used solely to close the resource.

Pull Request/Issue Link: https://github.com/serc-courses/se-project-1--_11/issues/12

Numerous code smells identified have been addressed through refactoring, and the corresponding pull request (PR) is provided below:

https://github.com/serc-courses/se-project-1--_11/pull/19

https://github.com/serc-courses/se-project-1--_11/pull/22

https://github.com/serc-courses/se-project-1--_11/pull/27

https://github.com/serc-courses/se-project-1--_11/pull/30

https://github.com/serc-courses/se-project-1--_11/pull/31

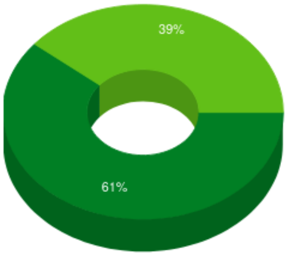
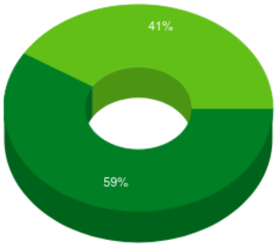
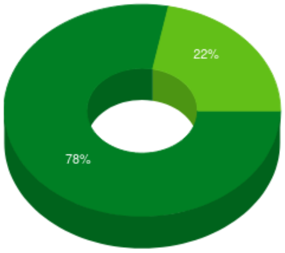
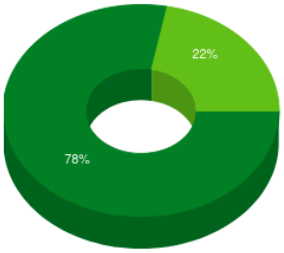
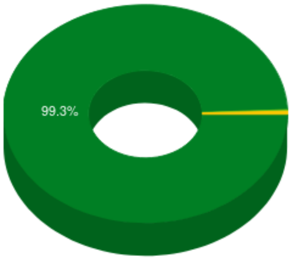
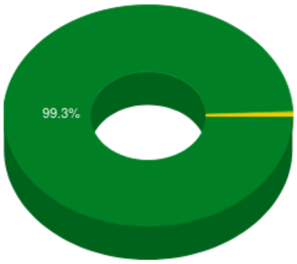
Task 3b Code Metrics

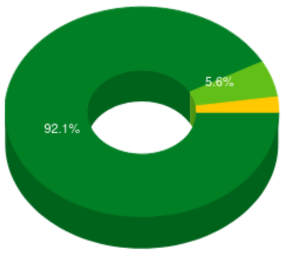
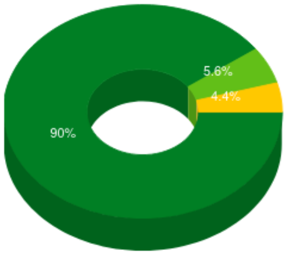
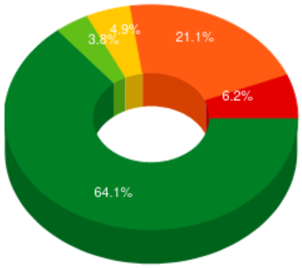
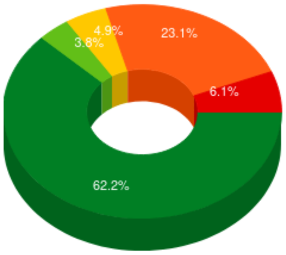
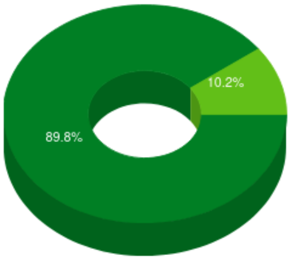
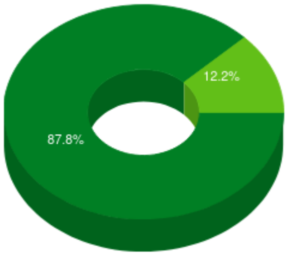
Tools Used:

We have used popular tools like **CodeMR** to compute the code quality metrics on the Books project.

CodeMR along with the code metric, generates comprehensive visualizations on it.

CodeMR Dashboard

S. N.	Metrics	Before Refactoring: Code Metrics	After Refactoring: Code Metrics
01	**Cyclomatic Complexity** ● Very High ● High ● Medium-high ● Low-medium ● Low		
02	**Depth of Inheritance Tree** ● Very High ● High ● Medium-high ● Low-medium ● Low		
03	**Number of Children of a Class** ● Very High ● High ● Medium-high ● Low-medium ● Low		

04	**Coupling Between Object Classes** ● Very High ● High ● Medium-high ● Low-medium ● Low		
05	**Lack of Cohesion on Methods** ● Very High ● High ● Medium-high ● Low-medium ● Low		
06	**Response for a Class** ● Very High ● High ● Medium-high ● Low-medium ● Low		

Detailed Evaluation reports generated by CodeMR tools can be accessed here.

Task 3c:

2a_1-Design Smell

Where is the design smell?

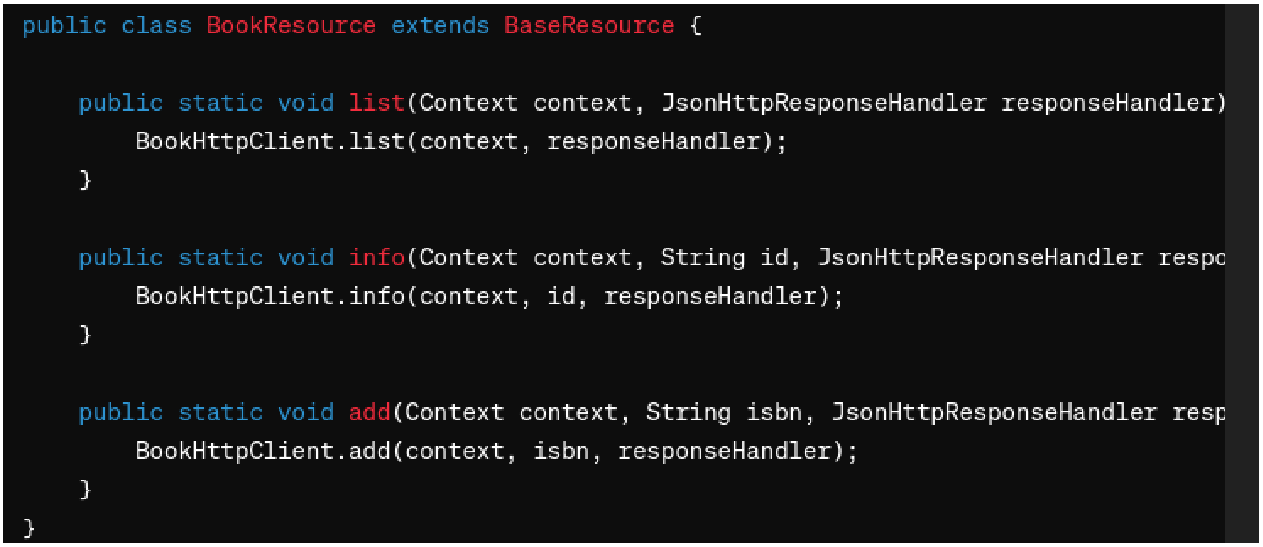
books-web/src/main/java/com/sismics/books/rest/resource/BookResource.java

Link to the ChatGPT chat

Original Smells detected by us: A "Brain Method" was detected. Refactor it to reduce at least one of the following metrics: LOC from 77 to 64, Complexity from 19 to 14, Nesting Level from 3 to 2, Number of Variables from 24 to 6. Please refer to section 2.a for more details.

Smells detected by ChatGPT:

1. **Large Class:** The BaseResource class seems to be doing too much. It handles authentication, checks permissions, and manages the principal object. This violates the Single Responsibility Principle (SRP), which states that a class should have only one reason to change.
2. **Feature Envy:** The methods authenticate(), checkBaseFunction(), and hasBaseFunction() seem to be more interested in the state of other objects than in their own state. They are accessing and manipulating the state of the request and principal objects, which could be seen as a sign of feature envy.
3. **Primitive Obsession:** The appKey field is a String, which is a primitive obsession. It might be better to encapsulate this value within a class that provides behavior related to the application key.
4. **Lack of Encapsulation:** The principal field is protected, which means it can be accessed and potentially modified by subclasses. This breaks encapsulation and could lead to unexpected behavior if not handled carefully.



```
public class BookResource extends BaseResource {

    public static void list(Context context, JsonHttpResponseHandler responseHandler)
        BookHttpClient.list(context, responseHandler);
    }

    public static void info(Context context, String id, JsonHttpResponseHandler respo
        BookHttpClient.info(context, id, responseHandler);
    }

    public static void add(Context context, String isbn, JsonHttpResponseHandler resp
        BookHttpClient.add(context, isbn, responseHandler);
    }
}
```

2a_2-Design Smell

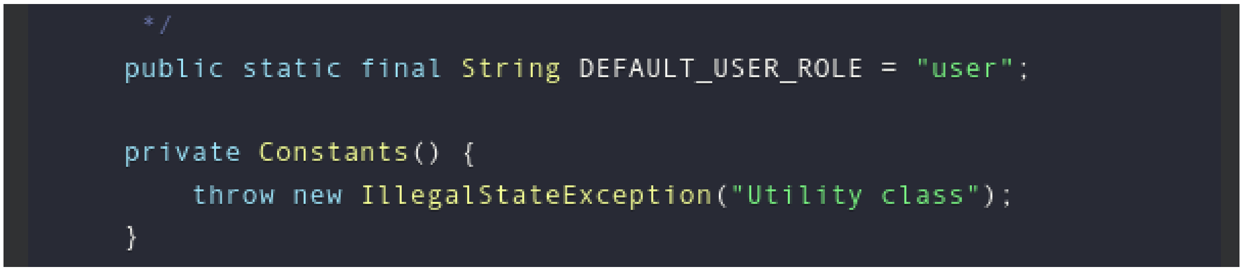
Where is the design smell?

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

Link to the ChatGPT chat

Original Smells detected by us: Hard-coded password in the constants file.
Smells detected by ChatGPT: The design smell in the given code is the inclusion of a hard-coded password in the constants file, which is generally considered bad practice for security reasons. Storing sensitive information, such as passwords, directly in the source code increases the risk of exposure and compromises security.

Observations: To address this issue, it is recommended to separate sensitive information like passwords from the codebase and use a more secure approach, such as storing them in a configuration file or using a secure credential management system.



```
 */
public static final String DEFAULT_USER_ROLE = "user";

private Constants() {
    throw new IllegalStateException("Utility class");
}
```

2a_5-Design Smell

Where is the design smell?

books-core/src/main/java/com/sismics/util/HttpUtil.java

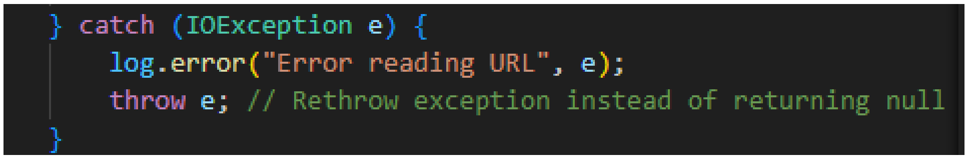
Link to the ChatGPT chat

Original Smells detected by us: Resource Management.

Smells detected by ChatGPT: Resource Management, Error Handling and Duplicate Code and Logging.

- **Error Handling:** Changed to throw IOExceptions to the caller, which allows for better error management and avoids returning null values.
- **Resource Management**: Adopted try-with-resources statements to ensure that resources are automatically closed in all cases, simplifying the code and improving reliability.
- **Duplicate Code and Logging:** This refactor didn't specifically address logging verbosity but improved resource management and error handling. For larger projects, consider implementing a more flexible logging strategy that allows for various levels of log details.

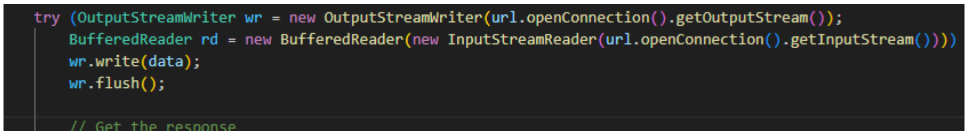
Error Handling: The below error handling of throwing IOExceptions to the caller was done by chatGPT refactored code.



```java
} catch (IOException e) {
    log.error("Error reading URL", e);
    throw e; // Rethrow exception instead of returning null
}
```

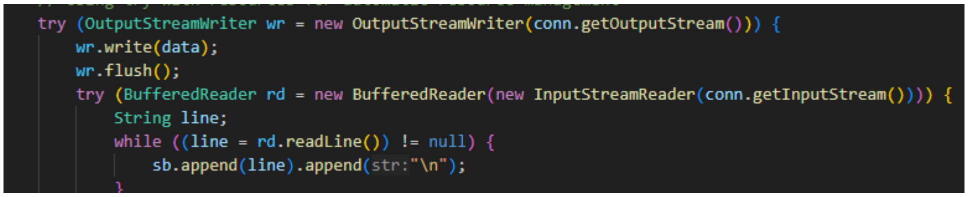
Resource Management: Adopted try-with-resources statements to ensure that resources are automatically closed in all cases, simplifying the code and improving reliability.

Below is the refactoring done by us,



```java
try (OutputStreamWriter wr = new OutputStreamWriter(url.openConnection().getOutputStream());
    BufferedReader rd = new BufferedReader(new InputStreamReader(url.openConnection().getInputStream())))
    wr.write(data);
    wr.flush();

    // Get the response
```

Below is the refactoring done by ChatGPT which uses Nested try-with-resources .



```java
try (OutputStreamWriter wr = new OutputStreamWriter(conn.getOutputStream())) {
    wr.write(data);
    wr.flush();
    try (BufferedReader rd = new BufferedReader(new InputStreamReader(conn.getInputStream()))) {
        String line;
        while ((line = rd.readLine()) != null) {
            sb.append(line).append(str:"\n");
        }
```

Observations: In this ChatGPT could identify different design smells and refactor properly than the manually done refactoring. During the manual refactoring the error handling part was missed out and also nested try-with-resources was done by ChatGPT. ChatGPT had a suggestion on combining the readUrlIntoString() and postUrl() but it did not provide this in output. In manual refactoring combining functions with similar tasks is possible with proper analysis.

2a_8-Design Smell

Where is the design smell?

books-core/src/main/java/com/sismics/books/core/model/context/AppContext.java

Link to the ChatGPT chat

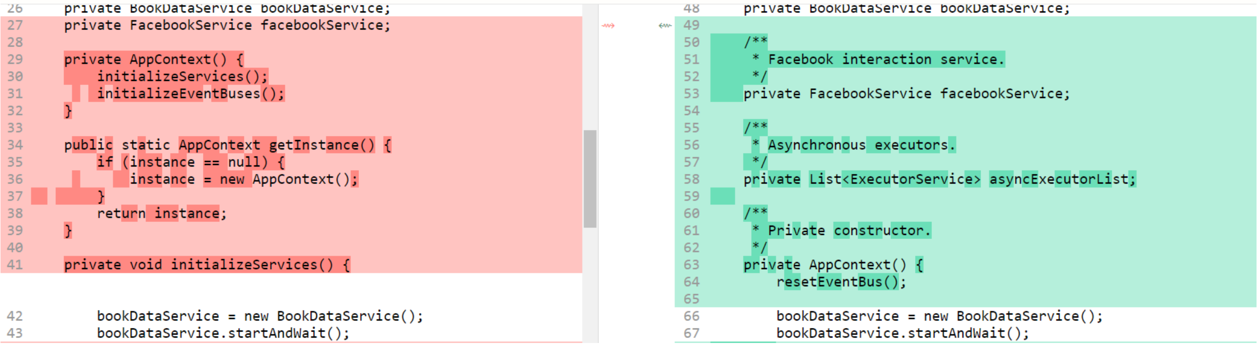
Original Smells detected by us: Singleton Pattern Misuse, God Object, Static Method, Hardcoded Values, Mutable Static State. Please refer to section 2.a for more details.

Smells detected by ChatGPT:

- **Singleton Pattern**: While the singleton pattern ensures there's only one instance of **AppContext**, it also introduces tight coupling and makes it hard to test.
- **Resource Management**: The code directly creates and starts services (**BookDataService**, **FacebookService**) within the constructor, which tightly couples the initialization of services with the **AppContext**.
- **Complex Initialization Logic**: The **resetEventBus()** method has complex logic for initializing event buses and executors, making it hard to understand and maintain.

Changes in the code. Red refers to the code generated by ChatGPT, Green is the refactored code by us. Please see the changes below

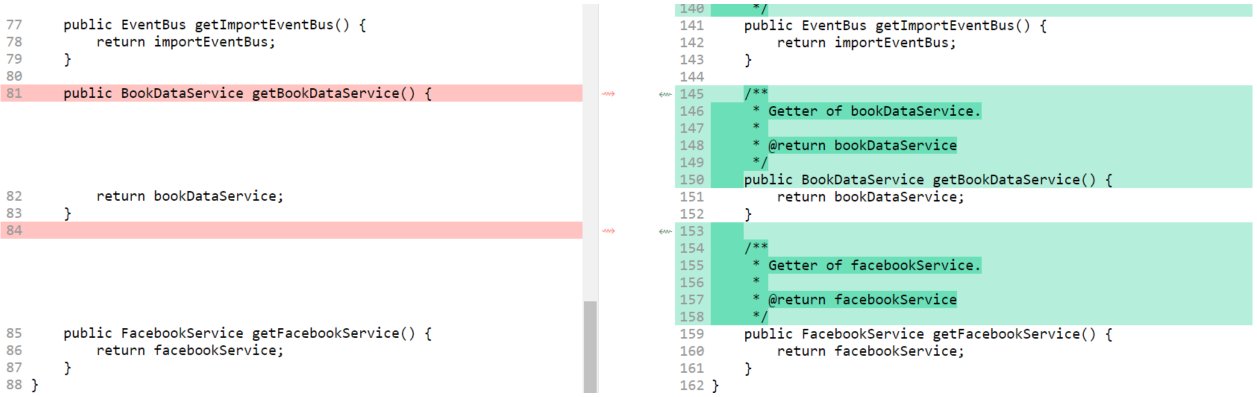
Removed Singleton Instance: While keeping the static **getInstance()** method, I removed the **instance** field and made the constructor public. This allows for easier testing and flexibility in managing instances.



```
    private BookDataService bookDataService;
    private FacebookService facebookService;

    private AppContext() {
        initializeServices();
        initializeEventBuses();
    }

    public static AppContext getInstance() {
        if (instance == null) {
            instance = new AppContext();
        }
        return instance;
    }

    private void initializeServices() {


        bookDataService = new BookDataService();
        bookDataService.startAndWait();
```

```
    private BookDataService bookDataService;

    /**
     * Facebook interaction service.
     */
    private FacebookService facebookService;

    /**
     * Asynchronous executors.
     */
    private List<ExecutorService> asyncExecutorList;

    /**
     * Private constructor.
     */
    private AppContext() {
        resetEventBus();

        bookDataService = new BookDataService();
        bookDataService.startAndWait();
```

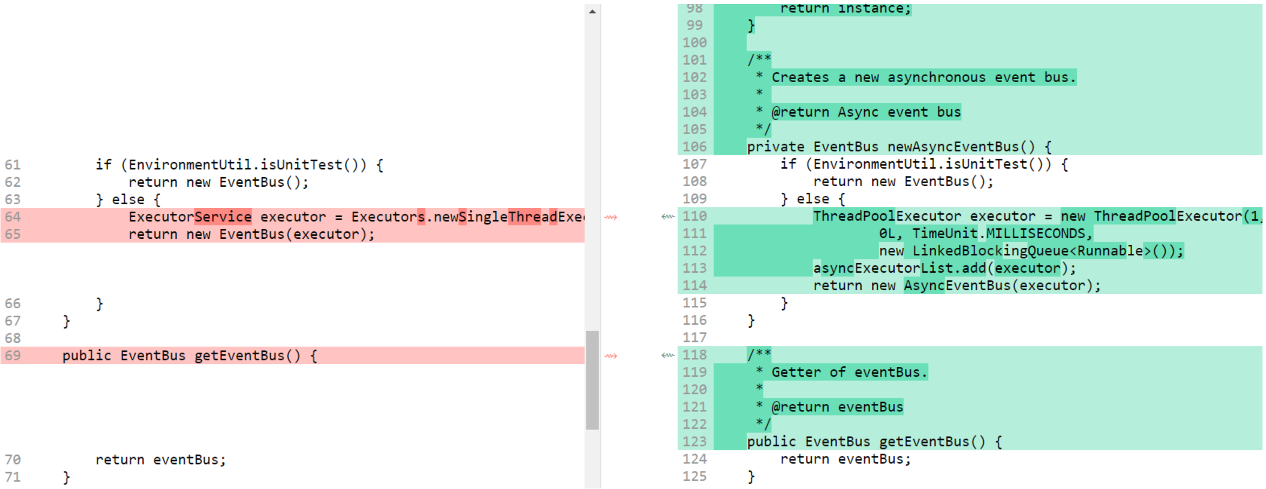
Separated Service Initialization: Moved the initialization of services (**BookDataService**, **FacebookService**) out of the constructor to their own method (**initializeServices()**).



```
    public EventBus getImportEventBus() {
        return importEventBus;
    }

    public BookDataService getBookDataService() {





        return bookDataService;
    }





    public FacebookService getFacebookService() {
        return facebookService;
    }
}
```

```
     */
    public EventBus getImportEventBus() {
        return importEventBus;
    }

    /**
     * Getter of bookDataService.
     *
     * @return bookDataService
     */
    public BookDataService getBookDataService() {
        return bookDataService;
    }

    /**
     * Getter of facebookService.
     *
     * @return facebookService
     */
    public FacebookService getFacebookService() {
        return facebookService;
    }
}
```

Simplified Event Bus Initialization: Removed the **resetEventBus()** method and instead directly initialized event buses (**eventBus**, **asyncEventBus**, **importEventBus**) in the constructor. Simplified the creation of **ExecutorService** for async event buses.



```
        if (EnvironmentUtil.isUnitTest()) {
            return new EventBus();
        } else {
            ExecutorService executor = Executors.newSingleThreadExe
            return new EventBus(executor);
        }
    }

    public EventBus getEventBus() {
        return eventBus;
    }
```

```
        return instance;
    }

    /**
     * Creates a new asynchronous event bus.
     *
     * @return Async event bus
     */
    private EventBus newAsyncEventBus() {
        if (EnvironmentUtil.isUnitTest()) {
            return new EventBus();
        } else {
            ThreadPoolExecutor executor = new ThreadPoolExecutor(1,
                    0L, TimeUnit.MILLISECONDS,
                    new LinkedBlockingQueue<Runnable>());
            asyncExecutorList.add(executor);
            return new AsyncEventBus(executor);
        }
    }

    /**
     * Getter of eventBus.
     *
     * @return eventBus
     */
    public EventBus getEventBus() {
        return eventBus;
    }
```

<u>Observations</u>: ChatGPT could identify the main code smells in the code but failed to explore more inherent code smells. Leveraging ChatGPT would be useful, having said that we observe that it needs more finetuning on detecting code smells data so that it could perform even better. Code given by ChatGPT is workable, but not always. Manual intervention of tweaking in between, suggesting few other code metrics might improve the performance of using large language models in the detecting code smells.

In General Observations | A detailed study

Please find the Chat link with ChatGPT here

Introduction

The objective of this study was to investigate the capabilities of ChatGPT in identifying and refactoring code smells in Java code. We explored various prompts and scenarios to assess the performance of ChatGPT in different contexts and scenarios.

Methodology

We conducted a series of experiments using different prompts and Java code snippets to evaluate ChatGPT's ability to detect and refactor code smells. The observations were recorded systematically to analyze ChatGPT's performance.

Prompts and observations

Prompt 1:

Prompt Given:

Given the prompt and Java code.

I will give you the java code from the repository. You have to identify different kinds of smells and refactor the code.

Please give the output in the below format

1. Detected design smell, code smell in the code

2. Where did you find the code

3. Refactored java complete code with appropriate comments.

Output Received: Initial success in identifying code smells, followed by a decrease in consistency and effectiveness with longer code snippets.

Observations: Noted that ChatGPT performed adequately with shorter code contexts but struggled with maintaining consistency and understanding longer codebases.

Prompt 2:

Prompt Given:

I will give you the java code and pointers where the different type of code smells is found. Please see the documentation and refactor the java code. Let me know if you found anything extra?

Output Received: Improved performance with shorter code contexts and specified code smells, but limited effectiveness in longer contexts.

Observations: Highlighted the dependency on well-defined prompts and noted ChatGPT's difficulty in handling larger codebases without manual intervention.

Prompt 3:

Prompt Given:

Did you find any code smells other than what I have given

Output Received: Some extraneous code smells suggested by ChatGPT, indicating a tendency to generate responses even when not directly relevant.

Observations: Noted instances of hallucinatory responses, where ChatGPT seemed to generate outputs without a clear connection to the task at hand.

Prompt 4:

Prompt Given:

Request to identify code smells independently.

Output Received: Successful identification of code smells in both short and long code contexts.

Observations: Acknowledged ChatGPT's ability to identify code smells effectively when prompted directly, highlighting its potential with the right guidance.

Prompt 5:

Prompt Given:

How do you like to refactor the code? Should I give the whole code at once or break into parts and give?

Output Received: Recommendation to provide code snippets in manageable parts for efficient discussion and refactoring.

Observations: Noted the importance of breaking down code snippets for effective collaboration and refining the refactoring process.

Prompt 6:

Prompt Given:

Refactore the java code, detect smells, I will break down a java code and give it to you in 3 instances. If there is any smell in the given part. please refactore or else say there is no smell. At last we can combine whole refactored code.

Output Received: Initial success followed by a decline in performance and relevance as iterations progressed.

Observations: Emphasized the need for continuous prompting and manual intervention, particularly with larger codebases.

Findings

1. Performance with Code Context Length:

- ChatGPT demonstrated proficiency with shorter code contexts but struggled with longer code snippets.
- Longer code contexts resulted in inconsistency and reduced effectiveness in identifying and refactoring code smells.

2. Prompt Dependency:

- ChatGPT heavily relies on the initial prompt provided.
- Forgetting the initial prompt led to a deviation from the intended task, affecting the quality of generated outputs.

3. Effectiveness with Small Code Contexts:

- In scenarios with smaller code contexts, ChatGPT exhibited better performance.
- It was particularly effective when pre-identified code smells were specified, showing excellent refactoring capabilities.

4. Challenges with Database Refactoring:

- Refactoring larger codebases posed significant challenges due to ChatGPT's tendency to forget context catastrophically.
- Continuous prompting after each interaction proved to be a viable workaround for maintaining context.

5. Prompt-Driven Output:

- The prompt's nature significantly impacted the output generated by ChatGPT.
- Prompt-driven interactions were crucial for steering ChatGPT towards the desired outcomes.

6. Manual Intervention Requirement:

- While ChatGPT showed promise, manual intervention was often necessary to ensure the quality of refactored code.
- Smaller codebases required less manual intervention and yielded excellent results.

7. Performance Enhancement through Fine-Tuning:

- Focusing on training ChatGPT-like models specifically tailored for code smell detection and refactoring could enhance performance and reduce dependency on manual intervention.

Conclusion

In conclusion, ChatGPT exhibits potential in code refactoring tasks, especially with smaller code contexts and well-defined prompts. However, challenges such as forgetting context and inconsistency in longer code contexts highlight the need for further research and development. Fine-tuning models specifically for code refactoring could address these challenges and unlock ChatGPT's full potential in automated code improvement tasks.