Report
Project - 1
Team - 20

Adyansh Kakran
2021111020

Meghana Tedla
2021102002

Prakhar Jain
2022121008

Rudra Dhar
2022801002

Shrikara A
2021101058

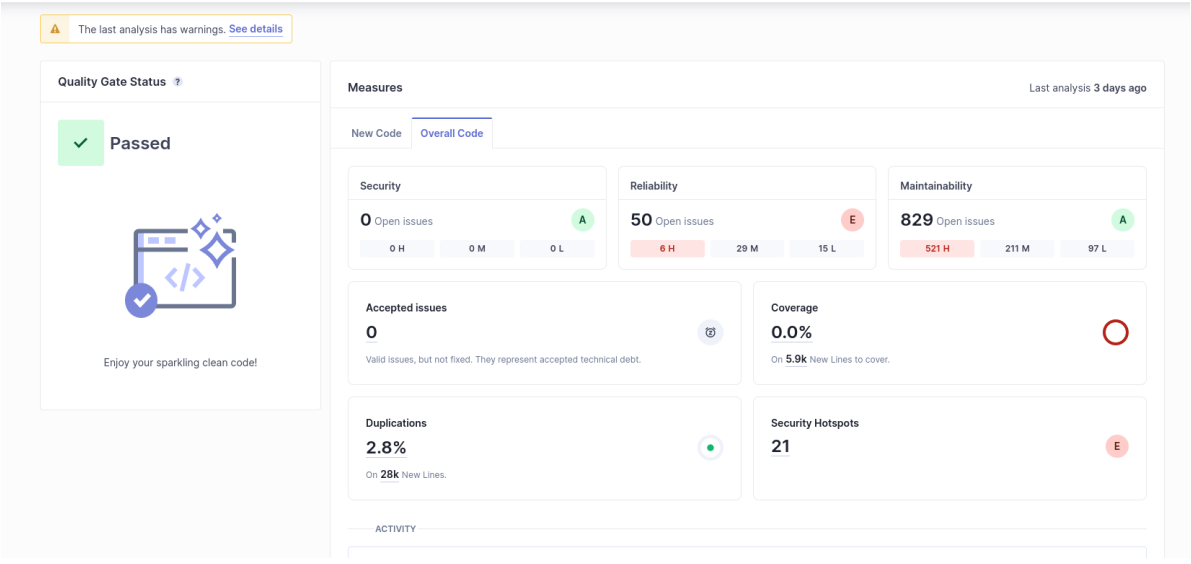
February 15, 2024

Analysis

Design Smells and methods to Refactor them

Writing clean code is essential to maintaining a healthy codebase. We define clean code as code that meets a certain defined standard, i.e. code that is reliable, secure, maintainable, readable, and modular, in addition to having other key attributes. After going through the code, we found a few code smells that were repetitive in nature and they resulted in design smells. As the code base was huge and we can't go through all the directories and code we used external resources. So for doing large scale analysis to find the code smells that can result into design smells, we used Sonarqube. SonarQube integrates into our existing workflow and detects issues in our code to help us perform continuous code inspections of our project. But since Sonarqube analysis is sometimes not upto mark and the results of Sonarqube is also not perfect. So, we have also used some other plug-ins like Designite in IntelliJ IDEA which helps us find design smells where Sonarqube helped us find the low level code smells.

Sonarqube gave us a massive 879 code smells in the whole codebase which could not be fixed in the entirety, but we found that majority of them were very small ones like usage of a string literal in multiple places without using a final static constant.



This analysis led us towards a lot of code smells which in turn pointed to design smells. The design smells we found were as follows -

1. **Violation of Encapsulation**
   In software design, a violation of encapsulation is a design smell that occurs when the principles of encapsulation are not properly followed. Encapsulation is a fundamental object-oriented programming concept that emphasizes hiding internal implementation details of a class while providing

a well-defined public interface for interacting with it. A few key characteristics of the same are Public data members, Direct manipulation of internal state, etc.

All over the codebase we found that the EntityManager was being directly feteched from the `ThreadLocalContext`. This means that other classes will have direct access to the `EntityManager` which is a design smell. To refactor it, we created a function that fetches the `EntityManager` and returns it.

```
1        private EntityManager getEntityManager() {
2            return ThreadLocalContext.get().getEntityManager();
3        }
```

2. **Broken Hierarchy**
In software design, a "broken hierarchy" is a type of design smell that indicates a potential problem with how inheritance is being used. It occurs when a "subtype" class doesn't actually have a true "is-a" relationship with its "supertype" class. This can lead to a number of issues such as Broken substitutability, Misunderstood relationships, etc.

   We found this design smell in mainly the Resources folder where each and every class extended the `BaseResource` class even though it was needed. To refactor this, we tried to use Composition in place of Inheritance but that did not work out due to a varied amount of usage of the functions between classes. This led us to create another abstract class called `AuthenticatedResource`. `BaseResource` extended `AuthenticatedResource` and all classes that required only the functionality of authentication inherited from `AuthenticatedResource` instead of `BaseResource`.

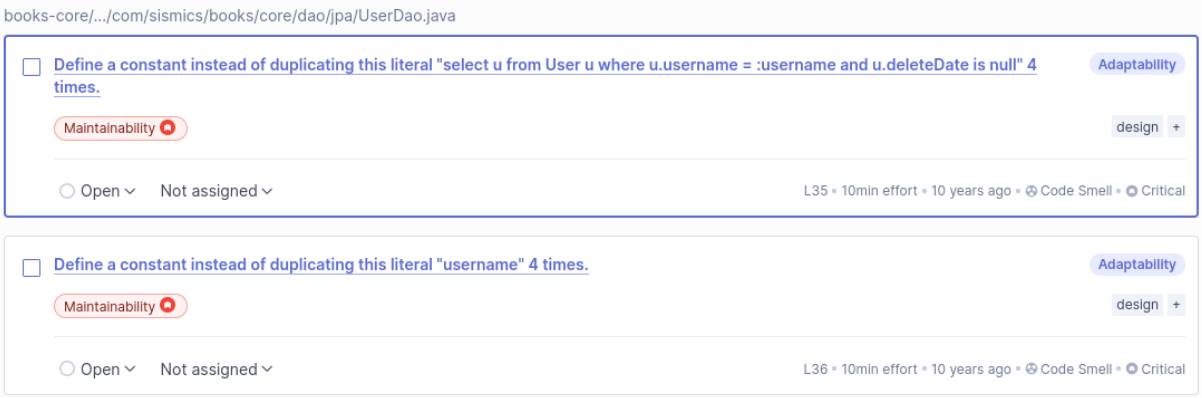
Project Name	Package Name	Type Name	Code Smell
se-project-1--_20	com.sismics.books.rest.resource	AppResource	Broken Hierarchy
se-project-1--_20	com.sismics.books.resource	BookResource	Broken Hierarchy
se-project-1--_20	com.sismics.books.rest.resource	BookResource	Broken Hierarchy
se-project-1--_20	com.sismics.books.rest.resource	ConnectResource	Broken Hierarchy
se-project-1--_20	com.sismics.books.rest.resource	LocaleResource	Broken Hierarchy
se-project-1--_20	com.sismics.books.rest.resource	TagResource	Broken Hierarchy
se-project-1--_20	com.sismics.books.rest.resource	ThemeResource	Broken Hierarchy
se-project-1--_20	com.sismics.books.resource	UserResource	Broken Hierarchy
se-project-1--_20	com.sismics.books.rest.resource	UserResource	Broken Hierarchy

3. **Missing Abstraction**
In software design, a "missing abstraction" is a type of design smell that occurs when code lacks proper abstraction mechanisms like classes, functions, or interfaces. This results in scattered data or logic, making the code harder to understand, maintain, and reuse. A few characterestics of this design smell are Primitive Obsession, Magic Numbers, Repetitive Code, etc.

   We found multiple instances of this in various files such as `UserBookDao` and `UserDao` where the function `getUserBook()` was repeated twice with a change of parameters. Code to fetch User from Database was also repeated in multiple places which led to the creation of the function `getQueryForUserFromDb()`. Magic strings were also found in multiple classes which were fixed by using `private static final` variables.

4. **Violation of Single Responsibility Principle**
The Single Responsibility Principle (SRP) is one of the fundamental principles of software design, especially in object-oriented programming. It states that a module, class, or function should have one, and only one, reason to change. In simpler terms, each piece of code should have a clearly defined purpose and avoid doing too many things at once. It is very important as it makes sure that you know what to edit as soon as you figure out what the error is.

   Another instance of this was seen in multiple functions where they were trying to access the database as well as validate and process the data. These kinds of functions were split into multiple



to follow this principle and improve maintainability. A few examples of these *long methods* are the `findByCriteria()` and `on()` functions of `UserBookDao` and `BookImportAsyncListener` classes respectively.

5. **Insufficient Modularization**
   Insufficient modularization, also known as the "God class" or "Blob" smell, is a design issue in software development. It arises when a single unit of code (like a class, function, or module) becomes overly large and complex, encompassing too many responsibilities without proper organization. A few key characteristics of this are Large class size, multiple functionalities, Scattered concerns, etc.

   We found multiple code smells that followed this design smell. One of these was in `UserAppDao` where both Business Logic and Data Access Logic were in the same place. To refactor this, we made another class called `UserAppMapper` which handled the Business Logic being handled originally in the wrong place.

LLM Refactoring

Note - All the code mentioned in the chats has not been tested with the original code, but a general overview of the generated code has been taken. GPT-3.5 has been used for all LLM testing.

BookImportAsyncListener

The link to the conversation is here.
We found that the LLM did a pretty good job of refactoring the long function in the `BookImportAsyncListener` file. It handled the long function `on()` very nicely and split it into multiple functions each with a single responsibility. It even gave better exception handling using a dedicated funtion which handles the error handling.

UserResource

The link to the conversations for this file are given below:

- This chat gave Design smells. Though some of the smells suggested by ChatGpt were right, some others were wrong. For example, it gave the design smell of Primitive Obsession, but on manual inspection, we found that it is not the case.

- This chat suggested code refactorings. Though it was able to refactor smaller code segments, like a function, it didn't consider the whole codebase. But since the codebase is multi-modular with imports from different packages and modules, the GPT refactored code gave errors with respect to imports and dependencies.

UserDao

The link to the conversation for this file is here.
We found that the LLM did not efficiently refactor the code in this case. It found the Missing Encapsulation design smell correctly but could not refactor the other major design smells like Repitive Code, etc. The refactored code was not according to the suggestions that it gave. It identified a majority of the code smells that were identified by us but was unable to refactor them properly.

UserAppDao

The link to the conversation for this file is here.
The quality of the refactored code in this case was much more efficient than the previous conversations. It gave code with very obvious bugs such as passing the `EntityManager` inside the constructor, which will not work with the given codebase and would require huge changes in multiple files. The identification of general code smells was still nice but the refactored code would have to be worked upon for some time even though it got rid of the majority of the code smells.

TagResource

The link to the conversation for this file is here.
The code smells and the alterate refactoring suggested were pretty accurate. The only limitation that we found was that it did not identify one of the most important design smells which is **Broken Hierarchy** because it did not have the required information of the whole codebase.

Summary

The suggestions and the detected code smells were pretty good in majority of the cases but the refactored code may or may not be integrable with the entire codebase. This makes us learn that you still cannot blindly trust LLMs for code refactoring and human intervention is still very important in this domain.

- **Readability -** The code is readable and is very well documented as well as explained by the LLM.
- **Maintainability -** The code seemed maintainable but considering that the LLM cannot get information about the whole codebase in one go due to the limitation of context length, it cannot produce the best results.
- **Efficiency -** The refactored code suggested are not integrable with the codebase which makes it less efficient. For a single file given to the LLM, it does make the code more efficient but requires human intervention to make it work.

Code Metrics Overview

Before Refactoring

Class Level

Name of Class	LOC	Cyclomatic Complexity	CBO	RFC	DIT	LCAM
BookDao	19	low	2	11	1	0.333
BookResource	366	medium-high	29	120	2	0.662
AuthenticationTokenDao	48	low	3	27	1	0.333
TagDao	84	low	4	34	1	0.472
PaginatedList	20	low	0	7	1	0.524
UserBookDao	81	low-medium	10	40	1	0.611
PaginatedLists	38	low	4	29	1	0.467
BaseResource	21	low	5	8	1	0.167
UserResource	297	medium-high	21	90	2	0.631
SortCriteria	12	low	0	3	1	0.444
UserDao	98	low-medium	10	54	1	0.58
TagResource	85	low-medium	8	31	2	0.125

Package Level

Name of Package	LOC	Coupling	Complexity	Lack of Cohesion
com.sismics.books.core.constant	2	low	low	low
com.sismics.books.rest.resource	1020	low-medium	low-medium	low
com.sismics.books.core.dao.jpa	554	low-medium	low	low
com.sismics.books.core.util.jpa	87	low	low	low

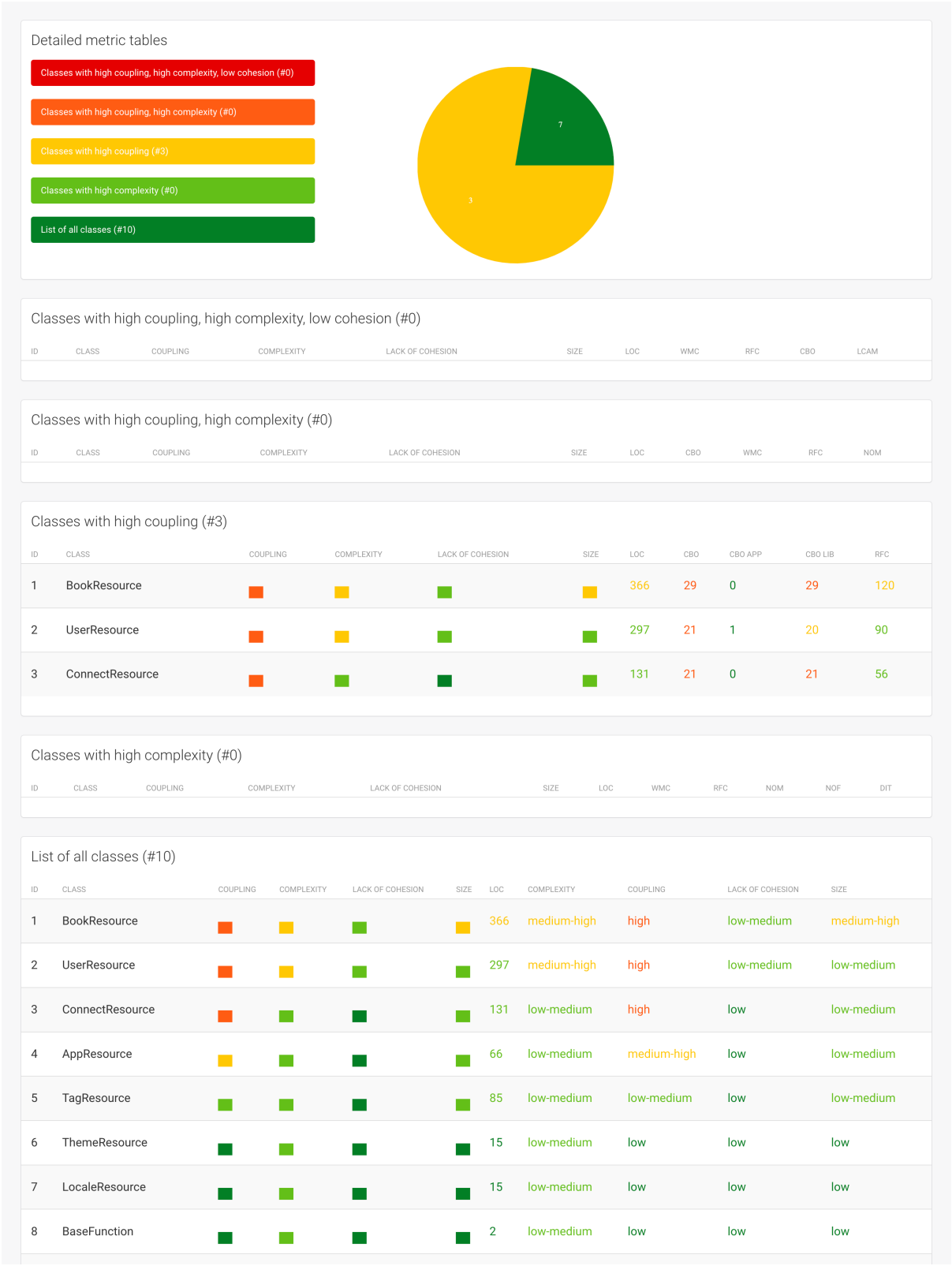


Figure 1: Books Web

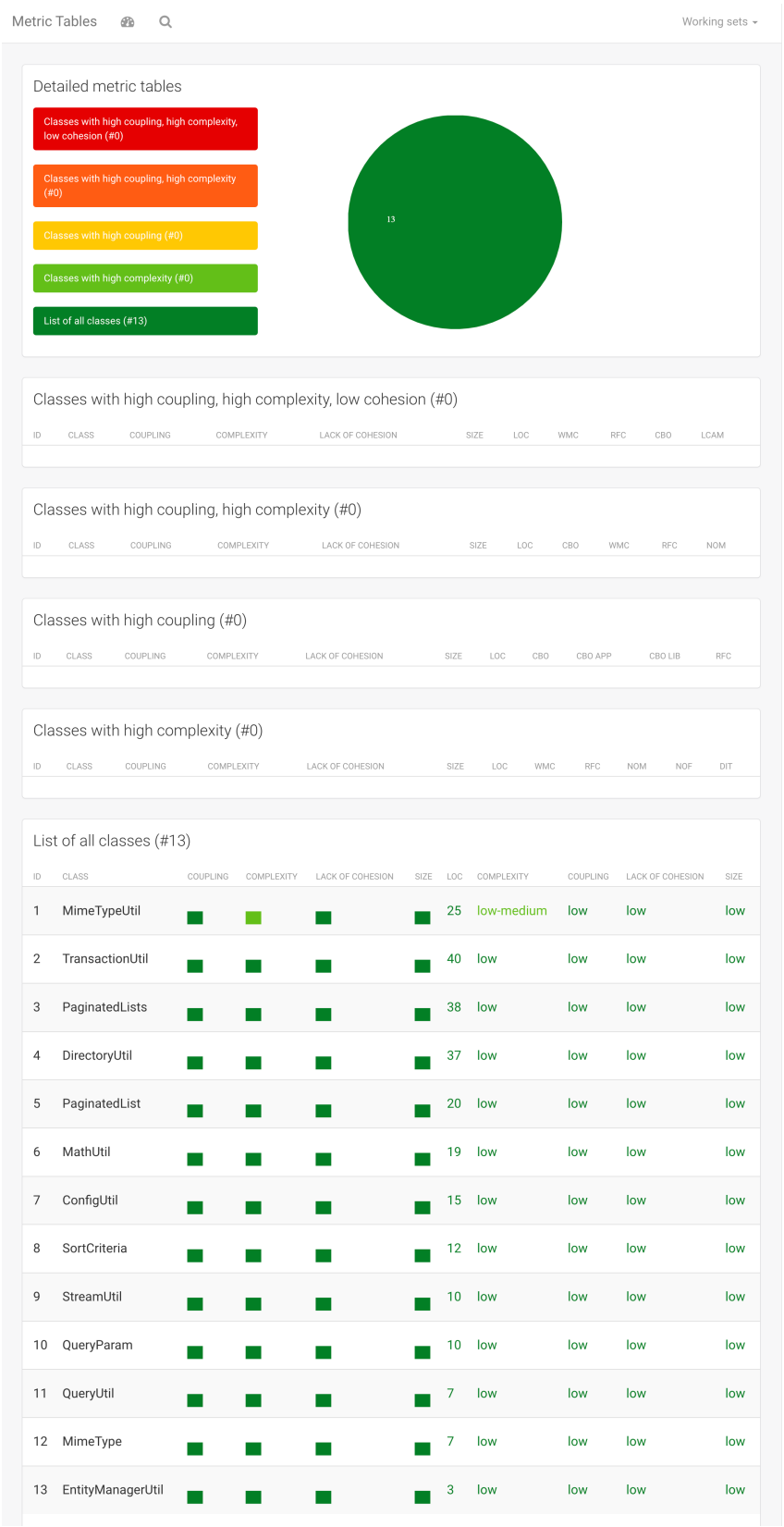


Figure 2: Books Core Utils

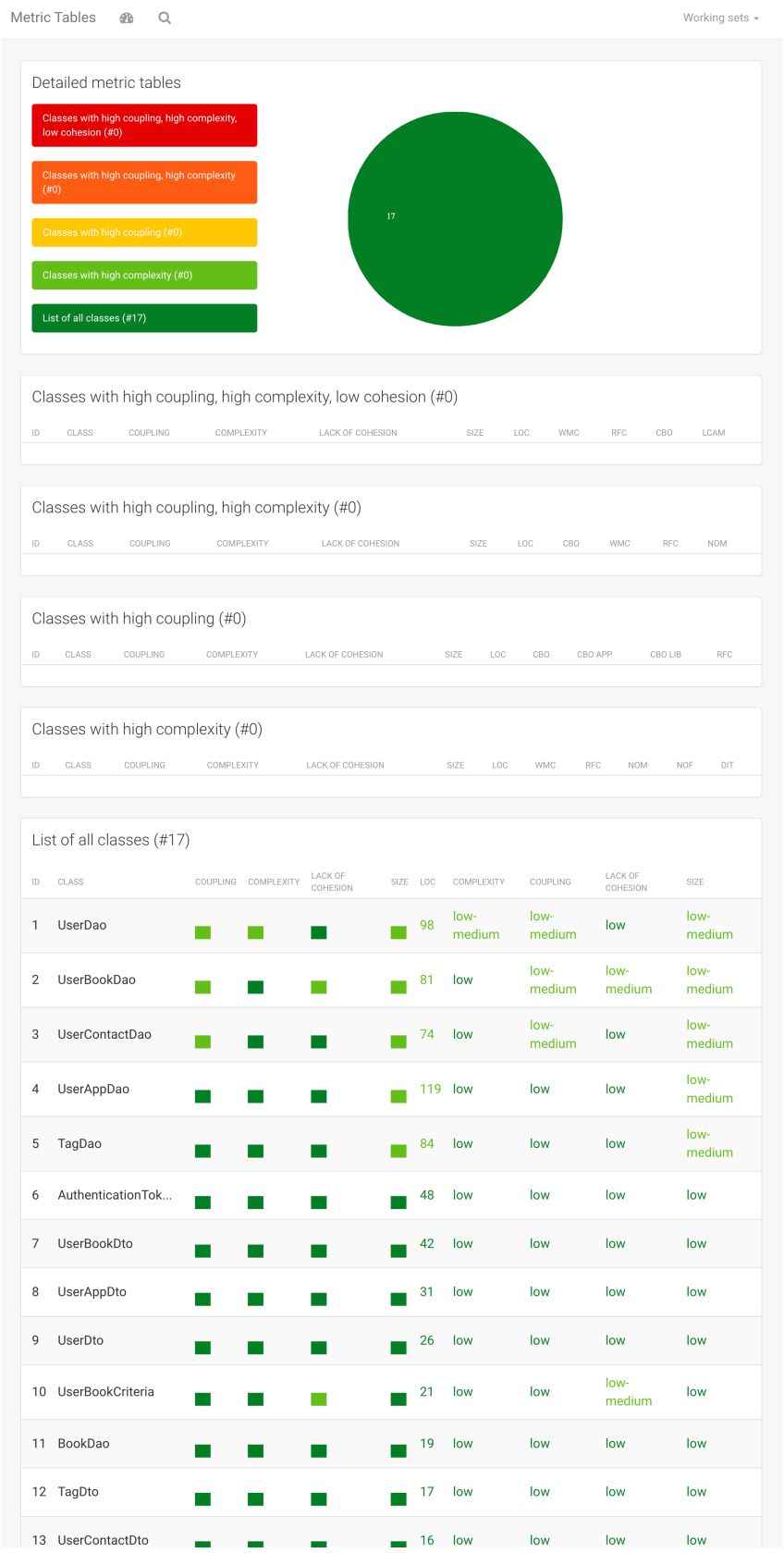


Figure 3: Books Core Dao Jpa

After Refactoring

Class Level

Name of Class	LOC	Cyclomatic Complexity	CBO	RFC	DIT	LCAM
BookDao	19	low	2	11	1	0.333
BookResource	359	medium-high	32	126	2	0.723
AuthenticationTokenDao	48	low	3	27	1	0.333
TagDao	96	low	4	41	1	0.571
PaginatedList	20	low	0	7	1	0.524
UserBookDao	87	low-medium	10	44	1	0.698
PaginatedLists	13	low	1	2	1	0.25
BaseResource	11	low-medium	3	5	2	0.000
UserResource	250	medium-high	24	95	3	0.64
SortCriteria	12	low	0	3	1	0.444
UserDao	102	low-medium	11	58	1	0.705
TagResource	90	low-medium	9	38	2	0.364

Package Level

Name of Package	LOC	Coupling	Complexity	Lack of Cohesion
com.sismics.books.core.constant	37	low	low	low
com.sismics.books.rest.resource	972	low-medium	low	low
com.sismics.books.core.dao.jpa	538	low-medium	low	low
com.sismics.books.core.util.jpa	87	low	low	low

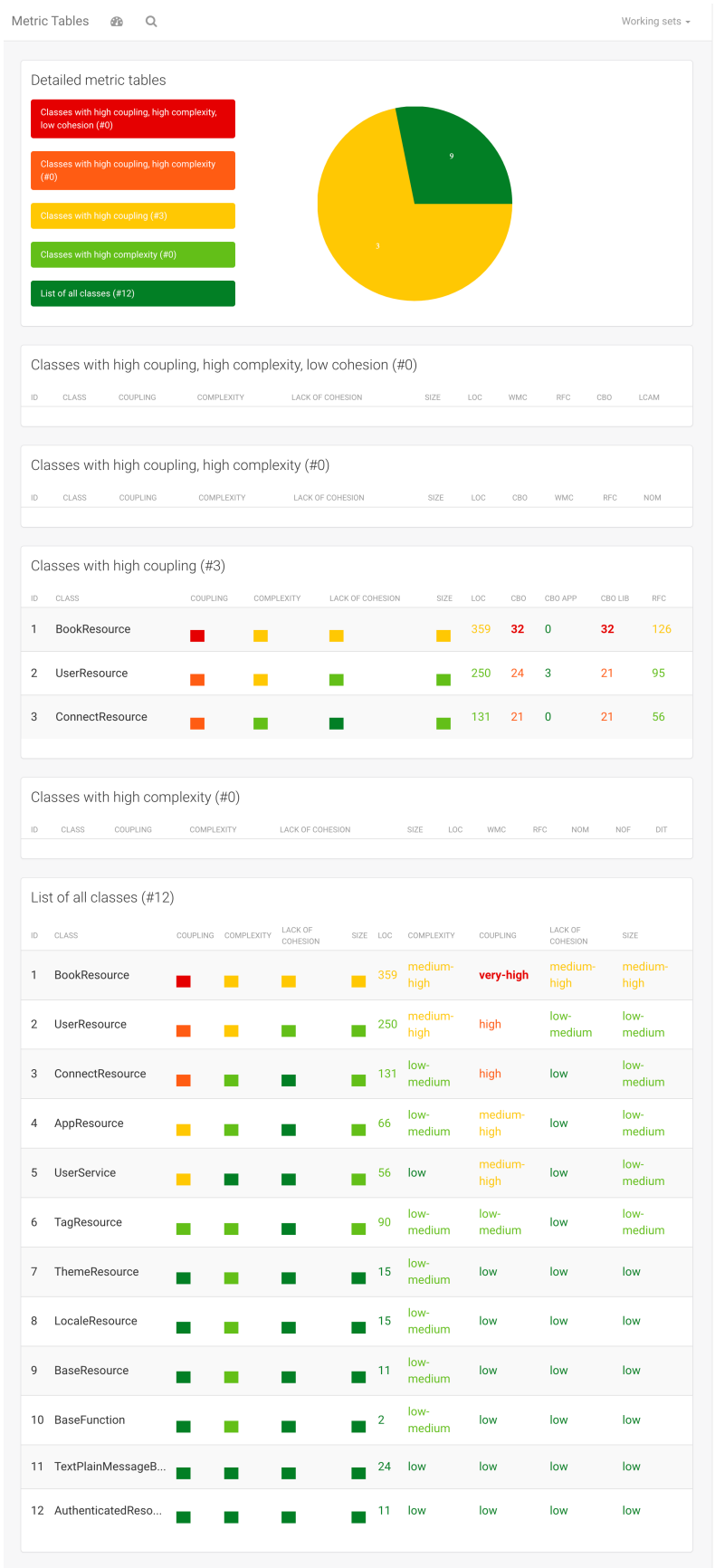


Figure 4: Books Web

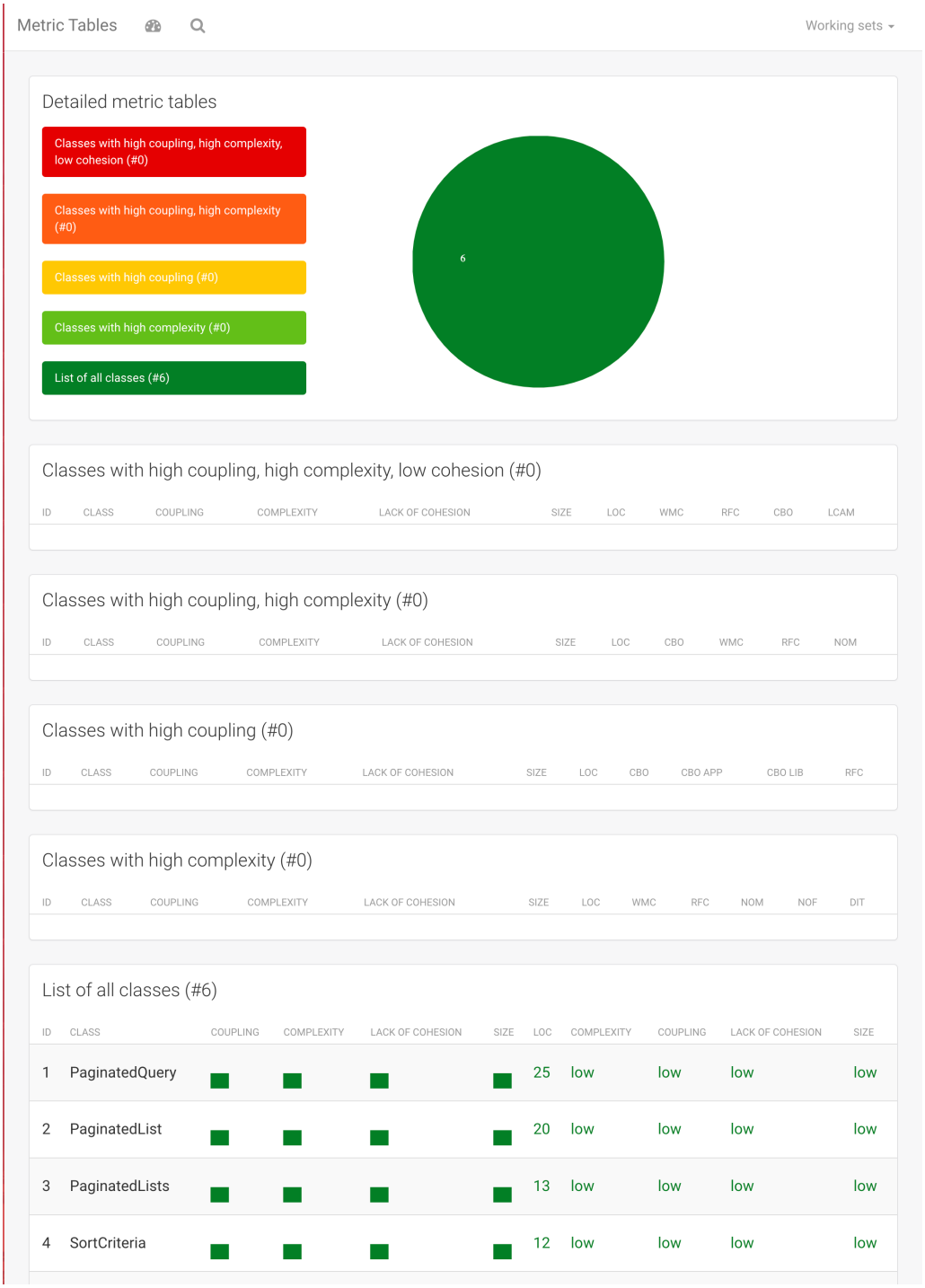


Figure 5: Books Core Utils

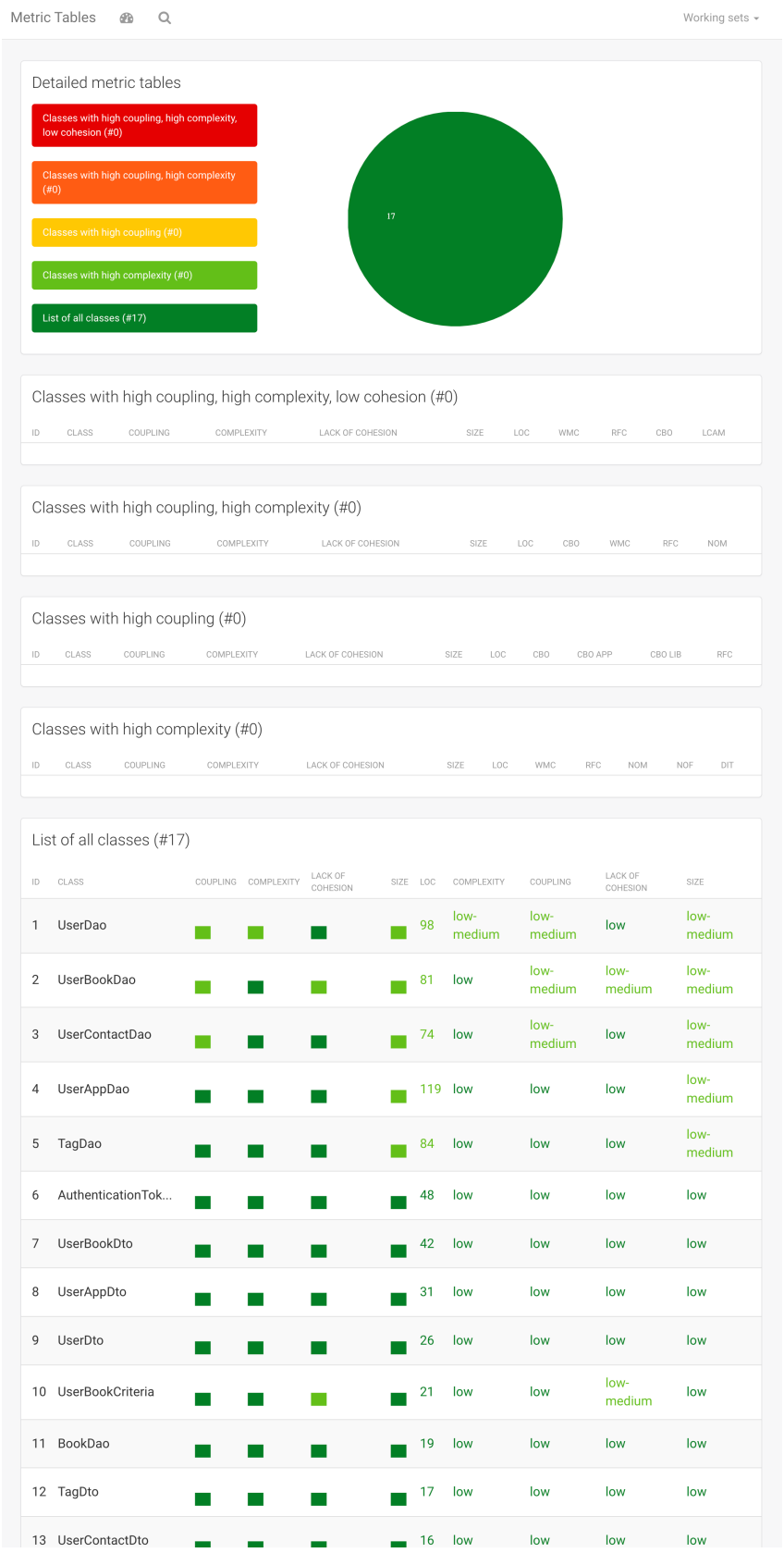


Figure 6: Books Core Dao Jpa

Explanation of Metrics

1 Lines of Code (LOC)

A class's line count is determined by the **LOC**. Excessive **LOC** could be a sign of bloated or overly complex classes, even though it is not directly connected with quality.

Impact on software quality: Excessively high **LOC** can impact readability, leading to decreased maintainability and increased likelihood of bugs.

Impact on maintainability: Classes with high **LOC** are harder to understand and maintain, as developers need to read through more code. Refactoring and making changes become more challenging, potentially introducing errors.

Potential performance issues: High **LOC** can lead to longer load times and increased memory usage, especially if the code includes unnecessary operations or inefficient algorithms.

2 Cycomatic Complexity

Cyclomatic Complexity Based on the quantity of linearly independent paths through the code, cyclomatic complexity calculates the complexity of a class or method.

Impact on software quality: High cyclomatic complexity can indicate code that is difficult to understand and prone to errors. It lowers the overall quality of the software by increasing its complexity.

Impact on maintainability: Classes with high cyclomatic complexity are harder to maintain and test as they have multiple execution paths. Changes become riskier, and understanding the behavior of such code becomes challenging.

Potential performance issues: Code with high cyclomatic complexity may have performance issues, as complex control flow can lead to inefficient execution paths and increased computational overhead.

3 Coupling Between Object Classes (CBO)

CBO measures the number of other classes a class depends on.

Impact on software quality: High coupling can lead to a lack of encapsulation and increase the likelihood of unintended side effects when modifying code.

Impact on maintainability: Classes with high coupling are harder to maintain as changes in one class may require modifications in many others. This increases the risk of introducing bugs and decreases the modularity of the codebase.

Potential performance issues: High coupling can impact performance by increasing the complexity of interactions between classes, leading to slower execution times and increased resource consumption.

4 Response for a class (RFC)

RFC counts the number of methods that can be executed in response to a message received by an object of a class. High **RFC** values can indicate classes with many responsibilities, violating the single responsibility principle.

Impact on software quality: High **RFC** values can indicate classes with multiple responsibilities, which can lower the overall quality of the software by violating the single responsibility principle.

Impact on maintainability: Classes with high **RFC** may be harder to maintain as they have multiple entry points and responsibilities. Changes in one part of the class can inadvertently affect other parts, making it harder to reason about the code.

Potential performance issues: Classes with high **RFC** may suffer from performance issues as they may perform multiple tasks, leading to increased complexity and resource consumption.

5 Depth of Inheritance Tree (DIT)

DIT measures how many levels of inheritance a class has.

Impact on software quality: Deep inheritance hierarchies can decrease software quality by making the code harder to understand and maintain. They can lead to a complex and fragile design.

Impact on maintainability: Classes with deep inheritance hierarchies are harder to maintain as changes in a base class can affect many derived classes. It increases the risk of unintended side effects and decreases the flexibility of the codebase.

Potential performance issues0: Deep inheritance hierarchies can impact performance by increasing the overhead of method lookup and dispatch, potentially leading to slower execution times.

6 Lack of Cohesion Among Methods (LCAM)

CAM metric is the measure of cohesion based on parameter types of methods.

$$LCAM = 1 - CAM$$

Impact on software quality: High **LCAM** values indicate poor cohesion among methods, which can decrease the overall quality of the software by making it harder to understand and reason about.

Impact on maintainability: Classes with high **LCAM** are harder to maintain as the methods may not be logically related to each other. This can lead to difficulties in understanding the behavior of the class and identifying appropriate changes.

Potential performance issues: Classes with high **LCAM** may suffer from performance issues as they may execute unrelated tasks within the same class, leading to increased complexity and potential inefficiencies.

Tools Used: CodeMR was used for all metrics mentioned above, and Sonarqube was used for CC. Checkstyle was used for NPath complexity but gave method level and a TA told us to report class level or above metrics, so it has been omitted from this report.

Analysis

1. **Lines of Code**: As seen from tables presented above, a significant decrease in lines of code per class, as well as per package. Since no functionality was lost, which was checked through manual testing (as the included tests did not pass for the original code base, and instructions given were to skip tests), we can conclude that there was a lot of duplicate code and unoptimal coding practice manifested in the form of code smells and design smells. This affected the readability and maintainability of files, especially large files with hundreds of lines of code. Through extracting methods and changing the inheritance tree, we can reduce the lines of code. We reduce the number of files with > 250 lines of code by half. This allows the files to be more readable, maintainable and understandable, with reduced chance of developer error.

2. **Cyclomatic Complexity (CC)**: As expected, most classes have the same amount of Cyclomatic Complexity since the general structure of methods within the classes were not egregiously written. We do see that the cyclomatic complexity for most classes remain same, for very few it increases, but the trend on the package level is that it either remains the same or reduces.

3. **Coupling Between object Classes (CBO)**: We see that a lot of classes have reduced CBO, indicating higher independence and promotion of the single responsibility principle. It also suggests improved flexibility and maintainability since classes do not depend on each other as much. It also improves testability by increasing frequency and importance of unit tests as opposed to integration tests. There are instances, however, where the coupling increases. This is expected when there is a class that had multiple smells relating to magic literals or was violating encapsulation by directly accessing deeper methods, such as in BookResource and TagResource. This is parallel to a shift from a point-to-point model, with lots of redundancy among classes to a hub and spoke model where there is a separation of responsibilities and centralisation of important business logic where necessary.

4. **Response for a Class (RFC)**: Similarly to above, we see that most classes have the same or lower level of RFC, with a couple showing a small increase. Reducing RFC focuses responsibility and reduces complexity. This increases testability, maintainability and promotes reuse.

5. **Depth of Inheritance Tree (DIT)**: Shallow inheritance is not always ideal, and we see in the Design Smells section that there were issues with the inheritance, like Broken Hierarchy, among others. Classes that are not the victims of this design smell have unchanged DIT, and those that have undergone refactoring might have 1 added to their DIT. We see that we follow the rule of thumb of being below 5 DIT.

6. **Lack of Cohesion Among Classes (LCAM)**: We see small changes in LCAM, mostly of the order of ± 0.1. A lower indicates focused methods, with low coupling and high reusability. However, some classes have higher LCAM, and this is expected for utility classes and data classes, which can have numerous helper functions and methods manipulating the data structure respectively.