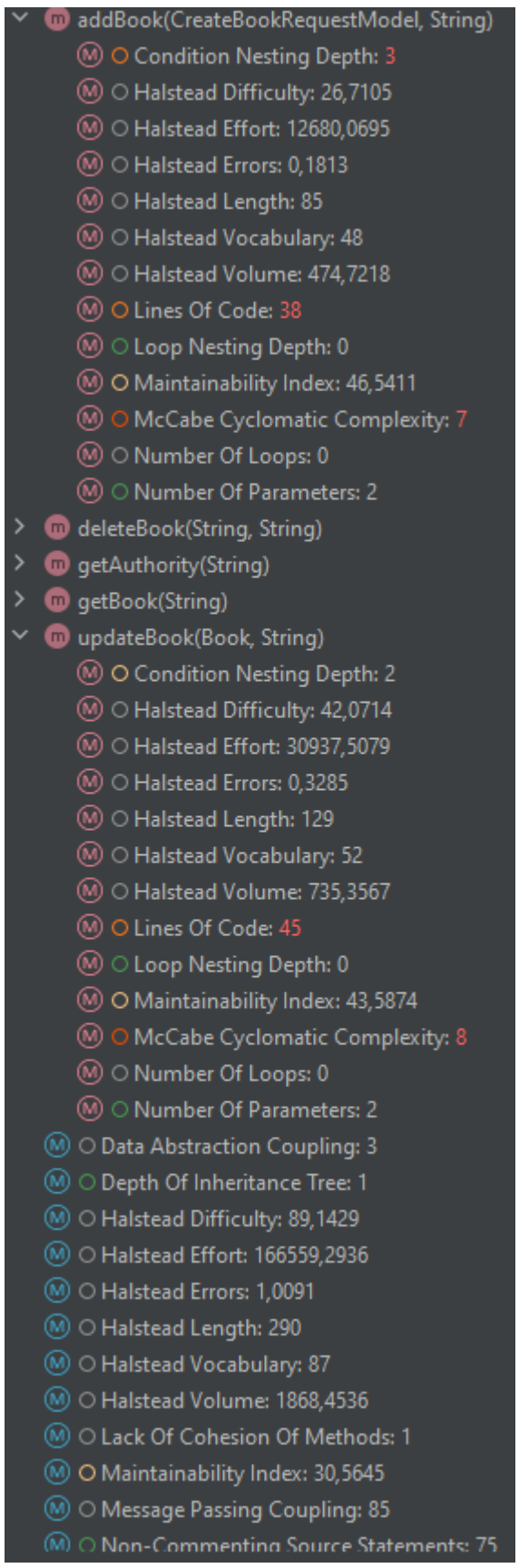
3. Software Quality Report

We noticed that the `BookService` class was violating the single responsibility principle as it was checking whether a user had the appropriate authority and then it was doing operations on the Book database. The methods with the most problems were `addBook` and `updateBook`, but `deleteBook` also had an unnecessary check. Below, in the left image, there is a screenshot from the MetricsTree IntelliJ plug-in, before addressing the problems:

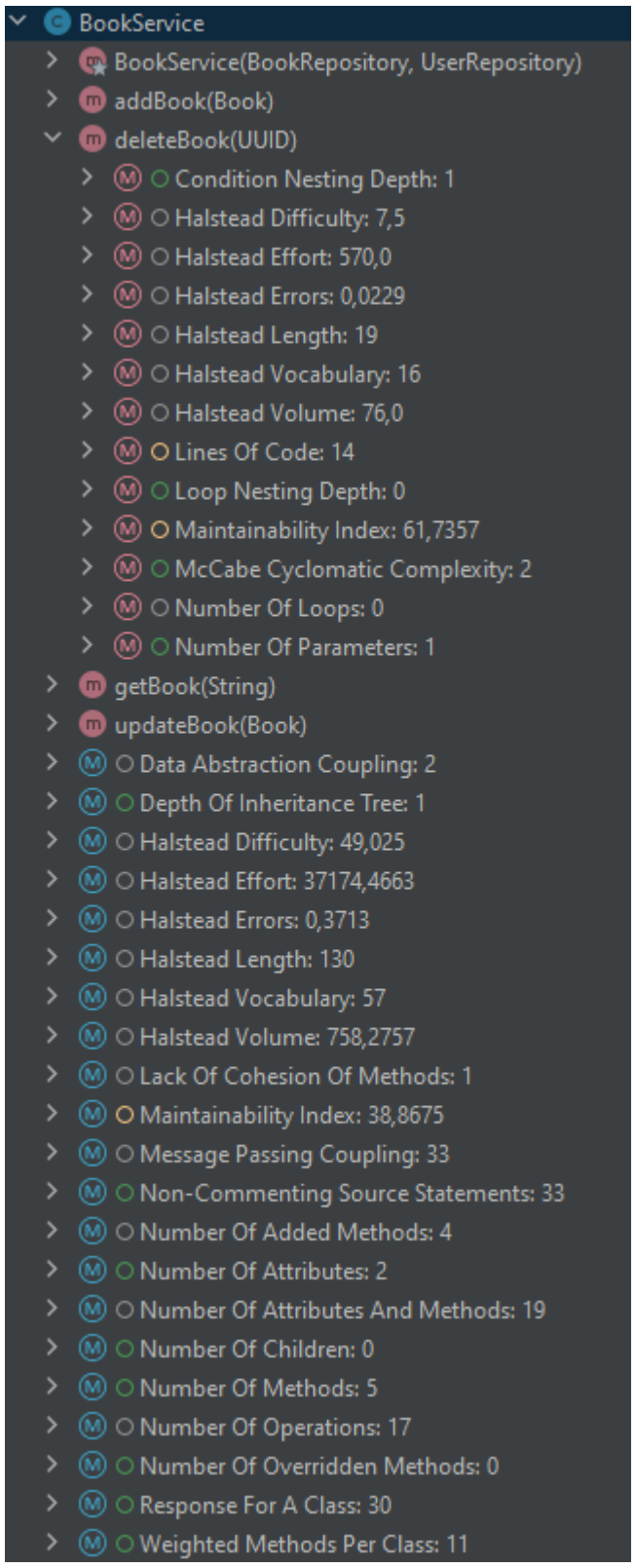


As can be seen, `addBook` and `updateBook` methods have a big Cyclomatic Complexity (7 respectively 8) and have over 20 lines of code, and big condition nesting (3 respectively 2), this being a bad sign. We decided to implement the Chain of Responsibility design pattern and to do the checks there, in the chained filters, instead of doing them inside the methods. This significantly reduced the complexity of the 2 methods as can be seen in the above image, on the right.

Now, lines of code, Cyclomatic Complexity and nested conditions are all green, having good values and increasing the maintainability of the class (right-hand side image). The changes were mainly done in the following commits:

- `Commit 1`
- `Commit 2`

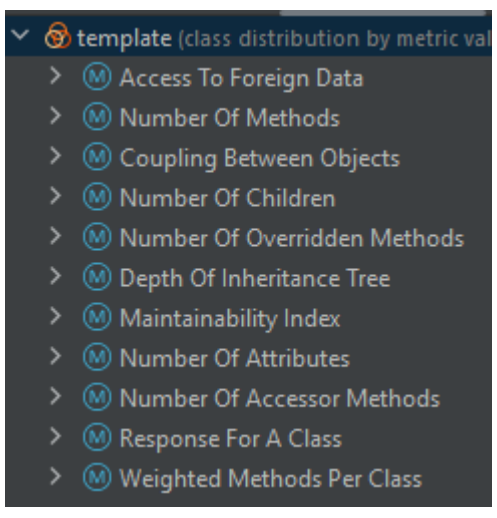
Then, we also adapted the chain of responsibility to work with the deleteBook method. This allowed us to completely remove the jwtService from the BookService class and to make the class adhere to the Single Responsibility principle, as it now interacts only with the bookRepository for performing CRUD operations regarding books.



This was done in `this commit`. It improved the lines of code metric and reduced the Cyclomatic Complexity.

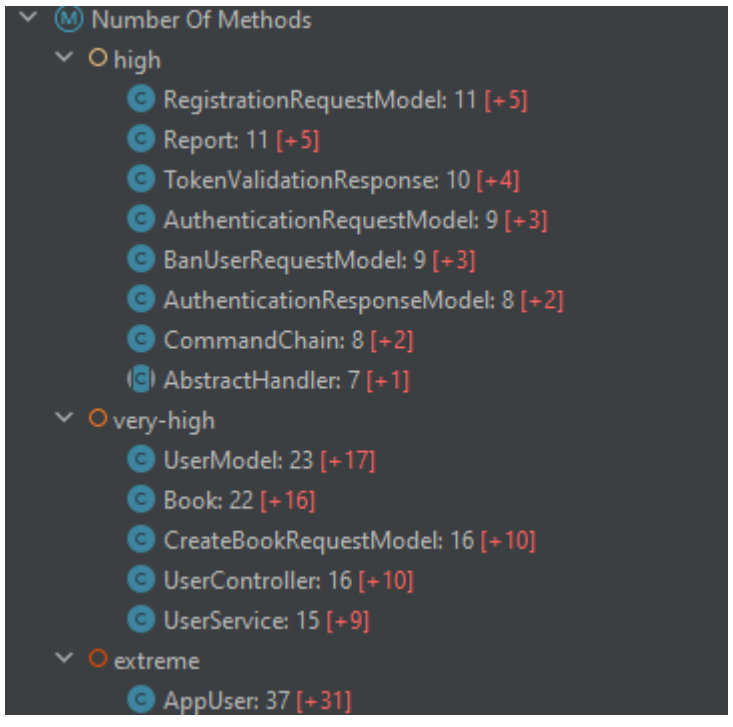
Metrics Analysis

After doing this change, we rerun the code metrics, using the `Project Metrics` section in the MetricsTree plug-in, and then selecting `Sort Classes By Metrics Value`. The following metrics were identified as having problems:



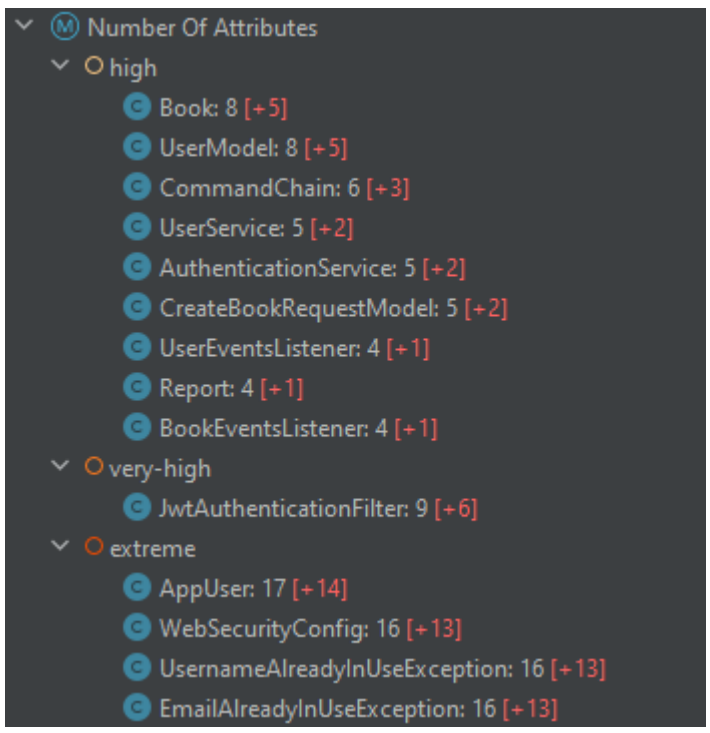
In the following section, we will discuss most of them (mostly the ones taught during class).

1. number of methods



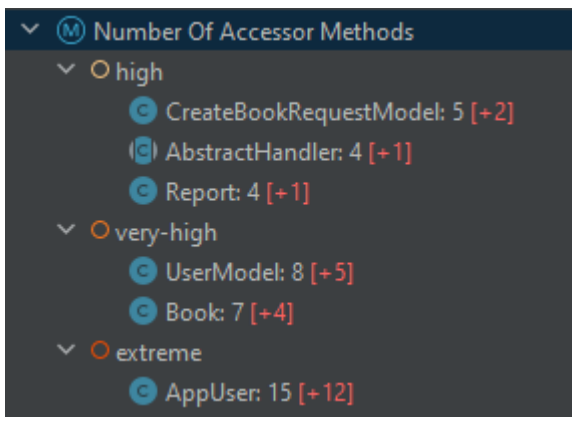
As can be seen, most of the classes which were identified with problems in respect with this metric are data objects or entities which by their nature need a lot of getters and setters, constructors, equals and hash code methods. This is why we decided not to address these problems. The `AbstractHandler` suffers from the same problem, as it exposes some setters to be used while testing. The `UserController` offers a lot of methods, as each field from the AppUser entity can be updated individually, so they are needed. This of course, propagates to the `UserService` as it needs to provide an update method for each field that is updated in the UserController. This is the reason they appear with a lot of methods, and we cannot fix this issue, except for doing the updates all together, which we opted against, as we think that the fields should be updated separately, as usually only a small subset of them will be updated.

2. number of attributes



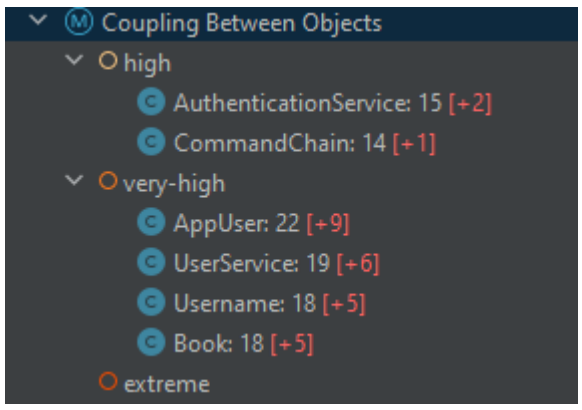
The `AuthenticationFilter` extends `OncePerRequestFilter`, 7 of the 9 attributes being inherited. This is why we cannot address this issue. For the `UserService`, we will try to reduce the number of attributes by creating a static class that holds all constants. The `CommandChain` needs to store the strategies and the contexts, and 6 is the minimum number of these (3 strategies and 3 contexts), so we cannot reduce this anymore. The 2 exceptions suffer from the fact that they inherit the Exception class, which has a lot of attributes, so it cannot be reduced anymore. The rest of the objects are data objects or entities, and decreasing the number of attributes could be done only by grouping some of them into smaller objects, which would increase the coupling, so it would introduce another problem. Additionally, we believe that the attributes belong together, so the cohesion is at good levels, in our opinion.

3. accessor methods (number of getters and setters)



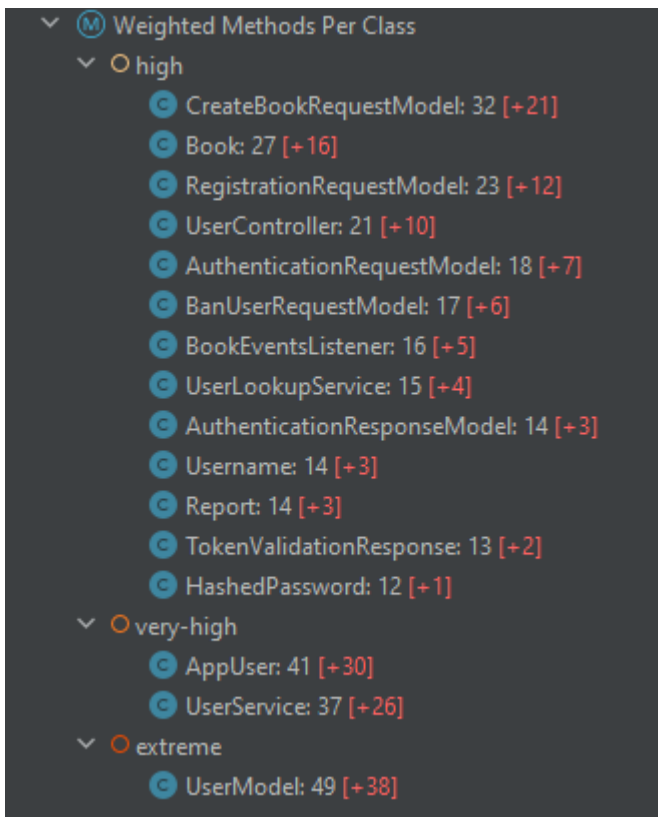
Except for the `AbstractHandler` which needs the getters for testing purposes, all the classes are data or entity objects, which need the getters and setters for proper functionality.

4. coupling between objects



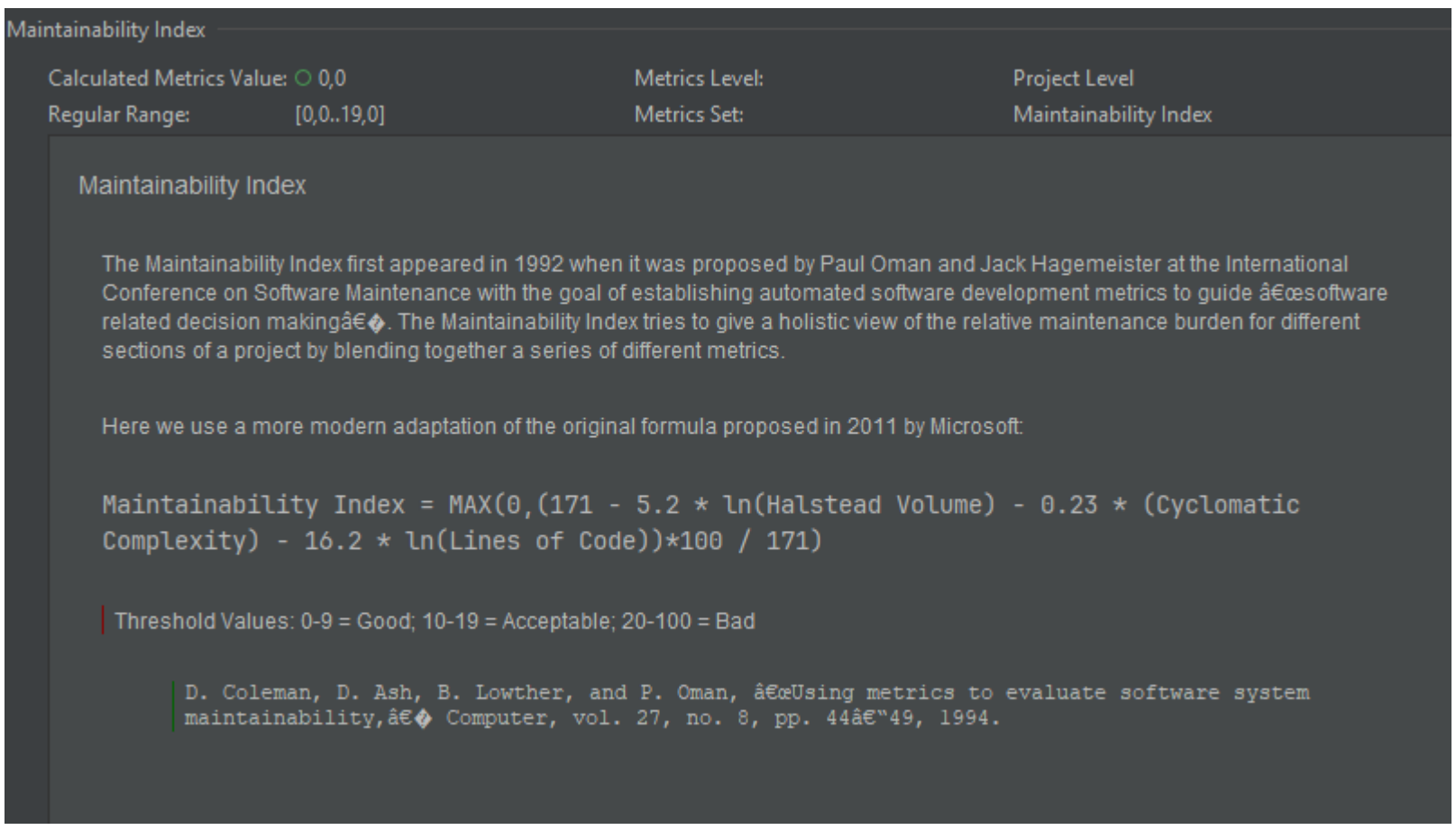
The Book and AppUser were expected to have high coupling as they are used in many parts of the application, being entities. The Username is also used in a lot of cases, as many requests receive a string which needs to be converted to a Username. The UserService has a high CBO because of the many fields it needs to be update, each of the update methods being from the UserController, increasing the coupling. The AuthenticationService and CommandChain have highcoupling as they depend on other classes, but this cannot be reduced, due to the nature of these classes.

5. weighted methods per class



The Cyclomatic Complexity of the methods is in normal parameters. The high values for this metric is correlated with the high metric for number of methods metric. The same classes which had problems there are found here again. Most of them are data or entity classes, which have a high number of getters, setters, constructors. We cannot improve these.

6. maintainability index



The maintainability index at the project level is 0.0, the best possible value. We believe this is a good indicative that the code is well written in general, and the problems are minor.

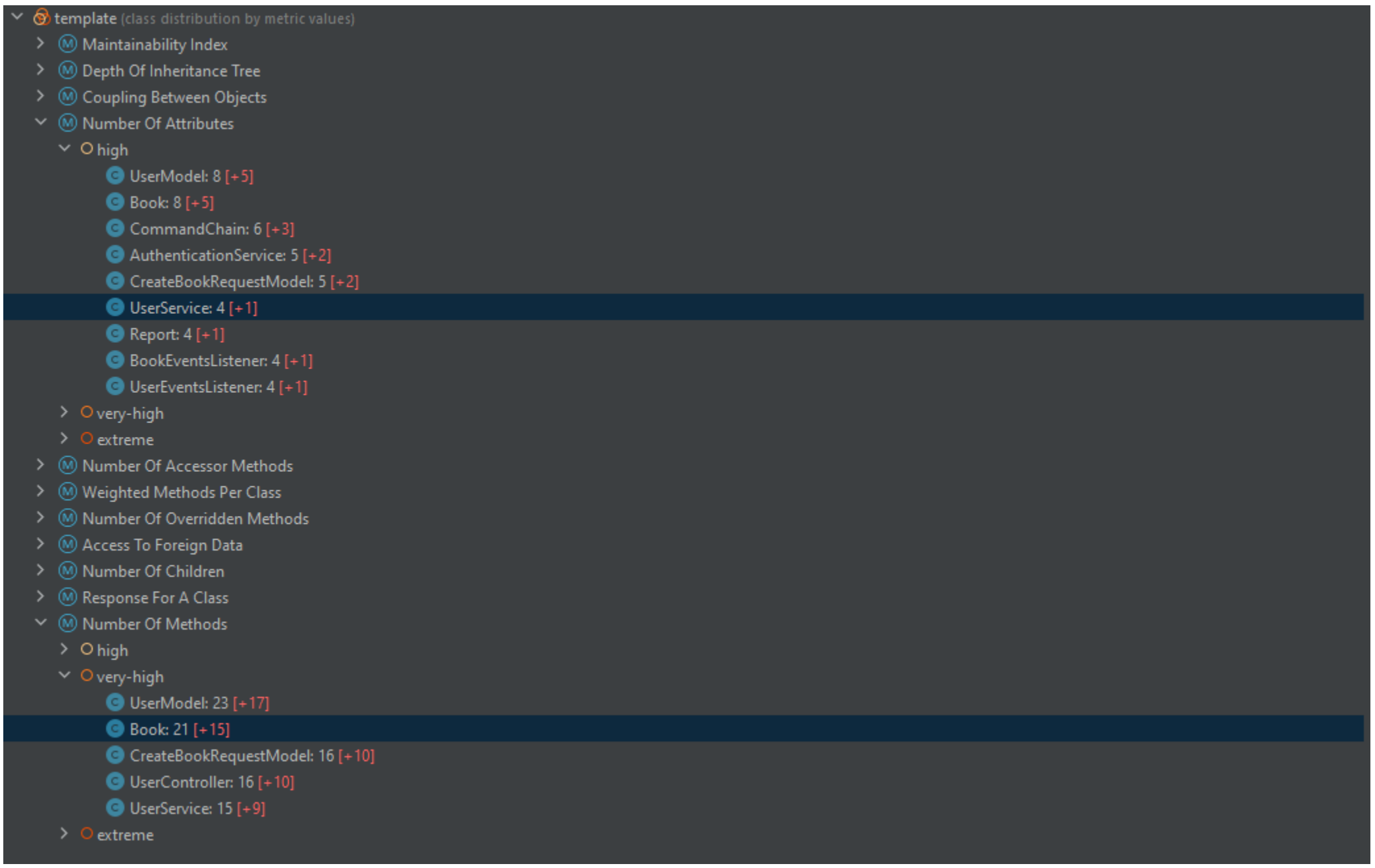
Addressed Problems

A couple of improvements were made:

- all constants were extracted in the Constants class for easier access and easier modification (for the other microservices' URL for example). This change reduces the number of attributes in the UserService class.
- remove unused method from the Book entity. This changes reduced the number of methods in the Book class.
- remove unused methods in JwtService class.
- removed all unused imports for better dependency management.

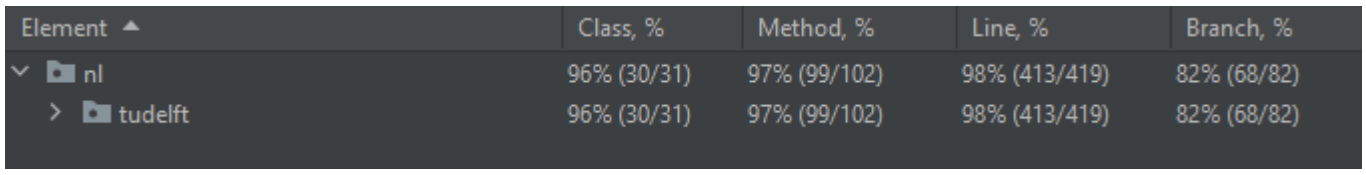
The changes were made in the following commits:

- `refactor constants`
- `remove unused code 1`
- `remove unused code 2`



4. Mutation Testing Report

From the start of the project, we agreed that all merge requests should be properly tested (a merge request should not decrease the overall line coverage). This meant that we were oriented on the tests from the start of the project, so we had a quite good test suite already. Below, there is an image from the IntelliJ coverage tool, whioch shows the line and branch coverage of the test suite at `this stage` :



Element	Class, %	Method, %	Line, %	Branch, %
nl	96% (30/31)	97% (99/102)	98% (413/419)	82% (68/82)
tudelft	96% (30/31)	97% (99/102)	98% (413/419)	82% (68/82)

This was reflected in the first run of the PIT tool, which gave us the following results:

pitest run 1.txt

As can be seen, we had a 95% mutation coverage and 82% branch coverage even without doing any improvements. We then did the first improvement based on the PIT report results. The goal was to improve the branch coverage to at least 90% and the mutation to 98%, which was documented in this issue.

There were some other branches merged on main, which were also merged into the branch of the issue, such that it better addresses the problem. The commits where most tests were improved, based on the PIT report results are:

- `Improve the branch coverage`
- `Remove unused files to kkill mutants`
- `Kill all mutants in UserController`
- `100% mutation coverage`
- `Improve tests after merge from main`
- `100% mutation coverage after merge from main`

During this improvement, the following reports were generated:

Pit Test Coverage Report

Project Summary

Number of Classes	Line Coverage		Mutation Coverage	
27	100%	522/522	100%	193/193

Breakdown by Package

Name	Number of Classes	Line Coverage		Mutation Coverage	
nl.tudelft.sem.template.authentication.application.user	1	100%	3/3	100%	1/1
nl.tudelft.sem.template.authentication.authentication	5	100%	68/68	100%	28/28
nl.tudelft.sem.template.authentication.config	2	100%	31/31	100%	8/8
nl.tudelft.sem.template.authentication.controllers	4	100%	88/88	100%	41/41
nl.tudelft.sem.template.authentication.domain	1	100%	7/7	100%	2/2
nl.tudelft.sem.template.authentication.domain.book	2	100%	103/103	100%	42/42
nl.tudelft.sem.template.authentication.domain.providers.implementations	1	100%	2/2	100%	1/1
nl.tudelft.sem.template.authentication.domain.user	11	100%	220/220	100%	70/70

Pit Test Coverage Report

Project Summary

Number of Classes	Line Coverage		Mutation Coverage	
30	100%	674/674	100%	244/244

Breakdown by Package

Name	Number of Classes	Line Coverage		Mutation Coverage	
nl.tudelft.sem.template.authentication.application.user	1	100%	3/3	100%	1/1
nl.tudelft.sem.template.authentication.authentication	5	100%	68/68	100%	28/28
nl.tudelft.sem.template.authentication.config	2	100%	31/31	100%	8/8
nl.tudelft.sem.template.authentication.controllers	5	100%	137/137	100%	58/58
nl.tudelft.sem.template.authentication.domain	1	100%	7/7	100%	2/2
nl.tudelft.sem.template.authentication.domain.book	2	100%	103/103	100%	42/42
nl.tudelft.sem.template.authentication.domain.providers.implementations	1	100%	2/2	100%	1/1
nl.tudelft.sem.template.authentication.domain.report	2	100%	41/41	100%	15/15
nl.tudelft.sem.template.authentication.domain.user	11	100%	282/282	100%	89/89

Intermediate reports were generated for analysing the files with problems, understanding the mutant that survived, analysing the test that covered that area, and changing the assertions or deriving new test cases. By the end of this merge request, we had 100% branch and mutation coverage.