Software Quality Report

Measuring code quality (analysis)

To monitor code quality, we are using several techniques.

1. We use Checkstyle and PMD rulesets. We have updated them slightly, to not include some very strange rules (for instance, order of imports), and added some important ones (such as cyclomatic complexity checking).

2. We ensure that we do not leave unsolved warnings in the code. There are exceptions, such as leaving a method to be later implemented, or when warnings clash with the application logic.

3. Furthermore, we use 1 main plugin: "CodeMR". It can calculate almost all metrics. We set the following thresholds for ourselves (the following metrics are exactly Chidamber-Kemerer metrics):

   - CBO - [0; 10]
   - DIT - [0; 3]
   - LCOM - [0; 1]
   - NOC - [0; 3]
   - RFC - [0; 35]
   - WMC - [0; 20]
   - [Additional, not Chidamber-Kemerer] LOC - [0; 200]
   - [Additional, not Chidamber-Kemerer] CC - [0; 6]

   Note that if all default CodeMR metrics (Coupling, Complexity, Lack of Cohesion and Size) are either low or low-medium, then our Chidamber-Kemerer metrics are all met. Therefore, when testing for our code quality, we simply check if all CodeMR metrics are low or low-medium. If they are not, we refactor so that they become that.

   As for LOC and CC metrics, first one we check manually, and the second one we handle by checkstyle.

4. We do not apply metrics to the DTO classes.

Examples of code quality improvements

Since we have been monitoring our code for quality purposes throughout the majority of the project, and most implementations of endpoints specifically wrote code with decent code quality, there were not many large code refactoring issues to do. In fact, the first time CodeMR reported a decent issue (i.e., non-low or non-low-medium metric verdict), was around 10 days before the end of the project. Despite us constantly writing code of good quality, some refactoring still had to be done. We will present some of these cases right now.

Refactoring large controllers

Our group wrote the OpenAPI specification the other way around, compared to the other groups: we wrote our models and our controller method declarations first, and from them, we generated the OpenAPI specification. But for that we needed to add a lot of annotations in our controller classes. That had cluttered our controller classes very much. You can check the state of the controllers at `e7c3b179`. With no functionality yet implemented, some classes had 250+ lines, and every method had 5+ annotations on top of it.

Therefore, after finishing with the OpenAPI specification, we decided to refactor each controller class into a pair of API interface and a controller class that implements that interface. We left all annotations in the API interfaces, and only the method definitions were left inside of the implementing controllers. As a result, our controller class sizes reduced by hundreds of LOCs.

The issue handling this is #22 (closed) and MR is !19 (merged). Notice that if you look at the state before this MR (`e7c3b179`) then there are some classes that have >200 LOC, which violates our metrics. For instance, ReviewsController has 247 LOC, which is way too much, considering that there is no functionality yet implemented in that class. But after refactoring, the class now has 129 lines, which is more than a 100 less.

Fixing technical debt

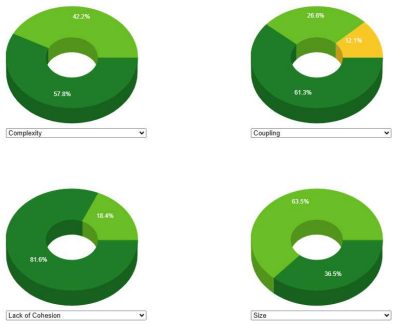
At some point in our project, a decent amount of technical debt was collected: most MRs were not checking for Java warnings, redundant files were left and not removed, there were instances of test duplication. We decided to fix this technical dept as soon as possible. Therefore, the issue #26 (closed) did that (MR !25 (merged)).

For instance, before the merge request was merged (`68afc893`), there were more than 50 warnings in the whole project, such as having redundant classes injected, having unused variables, or not making required fields `final` (most of these fixed in `68afc893` and `cef76699`). Also, there was a redundant test file, that did not do anything, and in general, redundant tests. Finally, this issue also reduced cyclomatic complexity of some methods `a14d2826`.

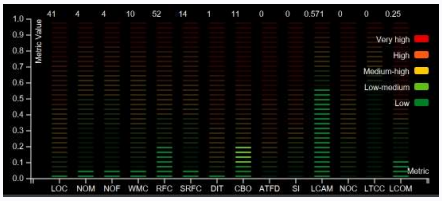
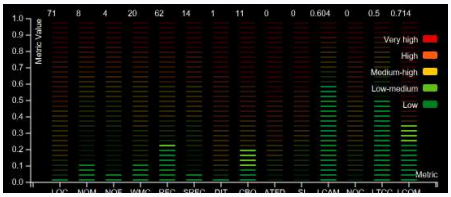
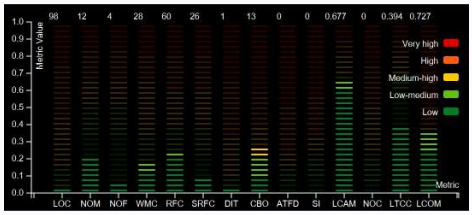
You can find the issue at #26 (closed) and the respective MR at !25 (merged).

Refactoring for code metrics

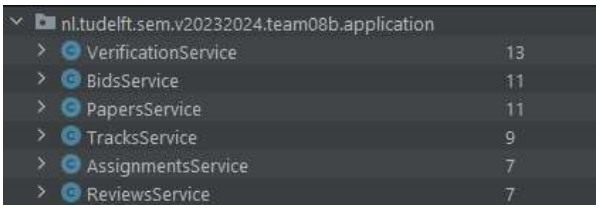
Finally, we get to code quality metrics. The largest code quality issue, that handled most of the refactoring was #29. If we look at what was the CodeMR plugin reported, before this issue was resolved, we see:



The yellow color means that there is a significant number of classes that highly coupled. If we inspect further, we can see the general metrics of these classes:

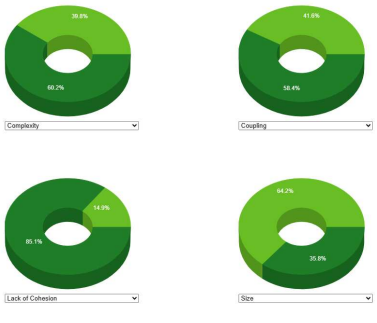
		
BidsService	PapersService	VerificationService

And the Coupling Between Objects (CBO) in particular:

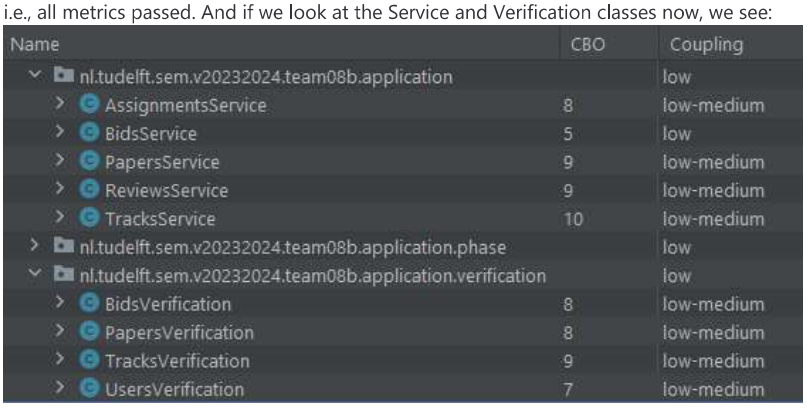


We see that 3 classes have high coupling (CBO > 10). After some inspection, we see noticed the reason for this is that VerificationService has become very large, handling verification for Users, Tracks and Papers, which is too much. Therefore, we decided to split the VerificationService class and also move certain verification methods from Bids and Papers Services into their respective Verification classes.

After the splitting was done (MR !33), we got the following CodeMR report:



i.e., all metrics passed. And if we look at the Service and Verification classes now, we see:



You can see that the VerificationService has been split into 4 specialized verification classes (BidsVerification, TracksVerification, PapersVerification and UsersVerification). You can also notice, that no single class has a CBO of more than 10.

The main commit for the splitting was e26a7d4b. For full commit and change list, see MR !33.

Other refactorings

Throughout the project there were multiple other, smaller refactorings done. We provide some instances:

- `217490d1` refactor a class to inject Clock for testability
- `b15b21c3` refactoring tests for readability and maintanability
- ...