SOFTWARE ENGINEERING METHODS

Assignment 2

Group OP5-SEM10:

Arjun Vilakathara

Stijn Coppens

Adriaan Brands

Gideon Bot

Ethan Keller

Noyan Toksoy

Table of Contents

Side note:

In this report we document how we refactored 5 methods and 5 classes. To find code smells we used a static analysis tool to compute metrics about the code. If these metrics surpassed some threshold we would refactor the respective code. An important note is that we had very good evaluations of the code by the static analysis tool in the beginning. The high code quality, at least as per codeMR metrics, made it 'harder' to actually find 10 fixes that were needed for this assignment. That is why this matter was discussed with the TA and it was concluded that valid fixes/refactorings done in the past could be used as well.

Tool used

The tool used throughout the refactoring process is CodeMR. This is a software quality and static code analysis tool which allows for easy visualisations of any major quality metric on both method and class level. This tool was chosen due to the ease of use both for generating reports and for integrated use in the IDE used, IntelliJ.

Code metrics

Class-level metrics:

Lack of Tight Class Cohesion (LoTCC):

Tight Class Cohesion measures the level of cohesion between public methods in a class. Thus if a class suffers from lack of tight class cohesion this means that the public methods of said class are poorly coupled and there are some design mistakes. Of course this is sometimes unavoidable in entity classes where getters, setters and other generic methods are very often not coupled at all yet necessary for the proper functioning of that class.

Coupling:

Coupling measures the degree of independence between different parts of the source code. As one might suspect this is basically the opposite from cohesion. Therefore high cohesion often correlates to low coupling (which is to be desired). When there is a high amount of coupling there is also a low amount of cohesion (relatively speaking), this situation is to be avoided. In this case it basically boils down to too many methods depending on each other thus if one has a bug none of the others will behave as expected/desired.

Lack of Cohesion of Methods (LCOM):

LCOM measures how classes are related to each other. This is a measure to see whether the class implements one or more responsibility. low cohesion means classes that are not very related and thus means that the class implements more than one responsibility. This lack of cohesion also implies that the classes should probably be separated into multiple classes. Ideally you would want to have classes with high LCOM which means that they implement one functionality.

Method-level metrics:

Number of Methods Called:

Number of methods measures as expected the number of methods called in a certain method. This can be an issue as the method quickly becomes very hard to comprehend and debug due to the large amount of deeper, nested function calls and is thus to be avoided if at all possible.

Nested Block Depth (NBD):

Nested block depth measures how many blocks “deep” the code goes. A high nested block depth becomes an issue due to the immense time and space complexities this creates. With multiple nested loops and/or if statements the complexity of the method skyrockets in any metric possible. It stands to reason that this leads to issues on many fronts such as performance, maintainability, debugging, …

Thresholds Chosen:

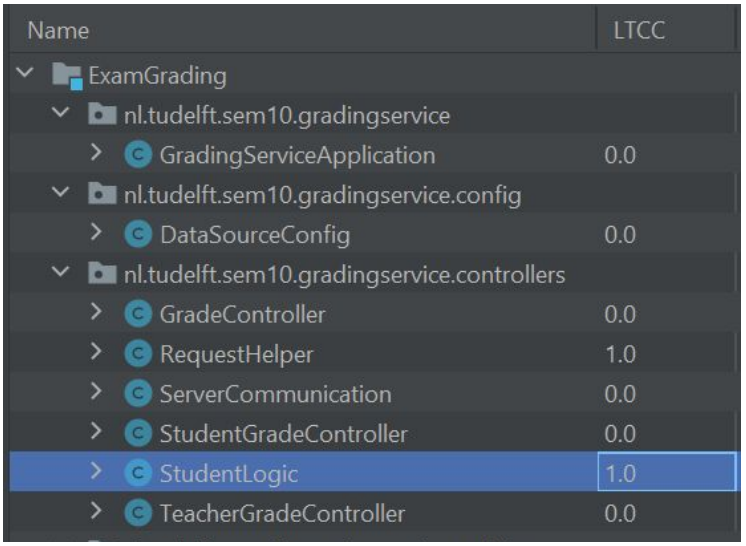
The thresholds chosen throughout the entire process were the default ones suggested by CodeMR. These already gave some warnings about certain classes having issues thus tinkering about with said thresholds was not entirely necessary. The aim of this refactor process is to get everything from a “red/orange/yellow” value to a “green” value by improving the metric used.

Class and method level fixes

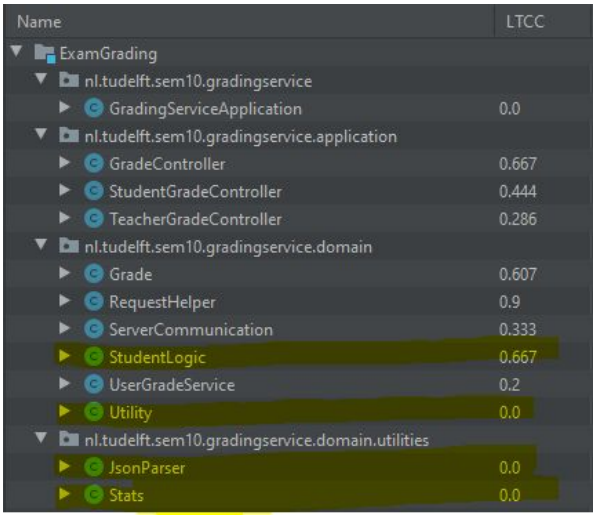
Class Level Fixes

Grade Service StudentLogic

In the grading service, the class StudentLogic initially has a LOTCC (Lack of tight class cohesion score) of 1, which is very high. This meant that the public methods within this class had very little correlation to each other, as they did not have any real connection between them.
These are the original metrics (LOTCC in the images is titiles LTCC):
Before



(The highlighted one is the class being considered)
These are the improved metrics:
After



(The highlighted classes are those discussed below)

The refactoring done here was further splitting up of the public methods in the class. It was understood that the reason the cohesion was so low was because the methods in the class

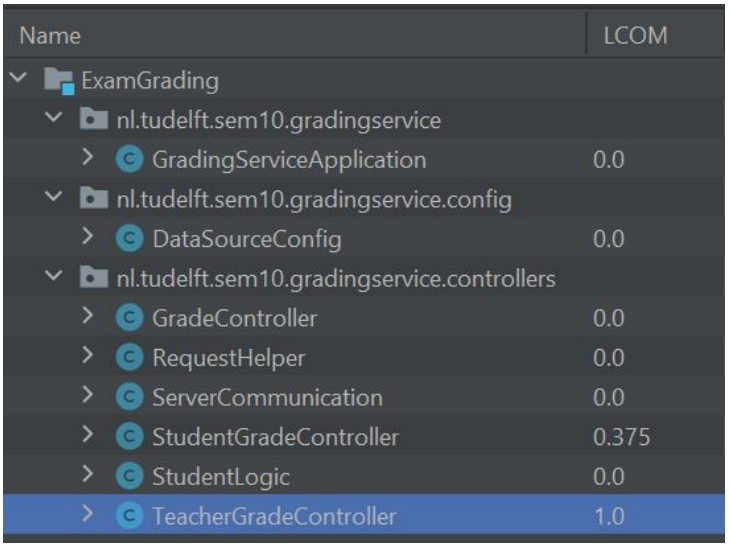
had different functionalities. Some of the methods such as getMean() or getVariance() were methods that calculated values in a list whereas the other method getGrade() was one that did interserver communication, parsed json, accessed the database and calculated based on that info. Based on what was discovered, it was decided that it would be best to implement the following:

- StudentLogic only contains getGrade() and setServerCommunication(). Get grade now only handles inter service communication and forwards the values to another class.
- A Stats class that holds all public calculation methods such as getMean, Weighted Average etc.
- A JsonParser class that handles parsing a Json string to a Json object.
- A utilities (Utility) method that used both JsonParser and Stats class and returned the calculated grade using an inputted jsonString (representing the weights of each exam) and a collection of grades.

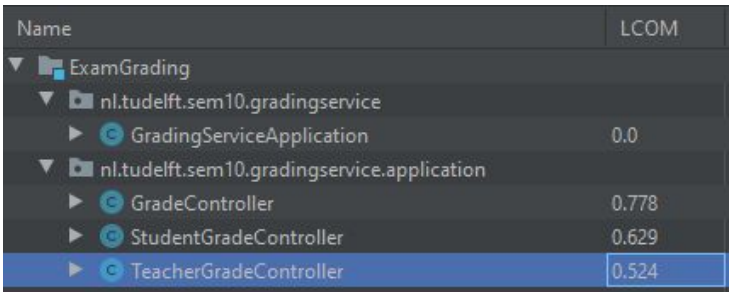
This refactor can be seen in the latest commit. The old version was *before on commit 5226d524915b2164cdf795ec5a1880435267dc52*. The idea was that by doing this methods that don't have any common ground i.e. are not cohesive are split into smaller more cohesive classes. As a result LOTCC value has dropped from 1.0 (Very high) to 0.667 (low-medium). Furthermore, the LOTCC values for the Stats, JsonPareser and Utilities classes are all 0. This would mean that indeed the way it was decided to split up the methods makes sense as now the new classes have a very low LOTCC score. However, StudentLogic still has a 0.677 but this can be due to the setter in the method and the getGrade method having very little in common. But this cannot be improved as the reason the setter is there is to make testing easier and removing it doesn't really improve the code in any beneficial way.

Grading Service TeacherGradeController

Before (before commit 5226d524915b2164cdf795ec5a1880435267dc52)



After

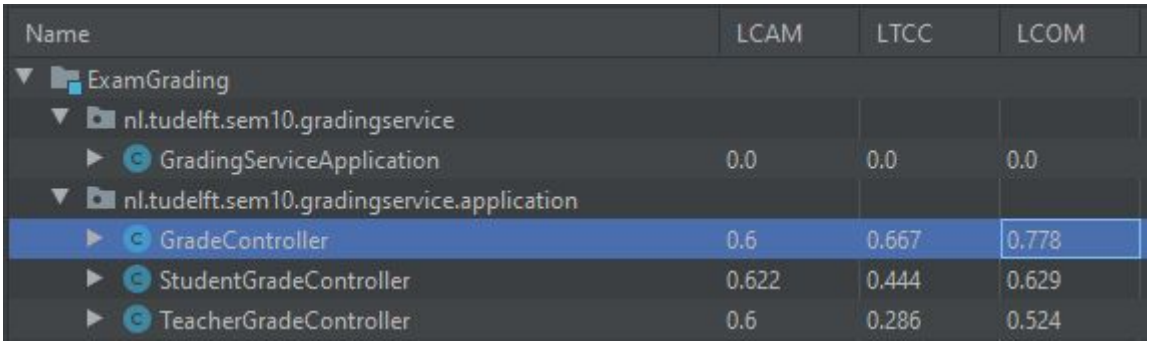


The lack of cohesion of methods in the *before* picture was at 1.0. This was because all methods updating, creating, and removing grades were not yet in the TeacherGradeController, but in the *GradeController*. On the 14th of December, it was decided that they were better suited for the *TeacherGradeController*, which improved the lack of cohesion to 0.524, as can be seen in the *after* picture.

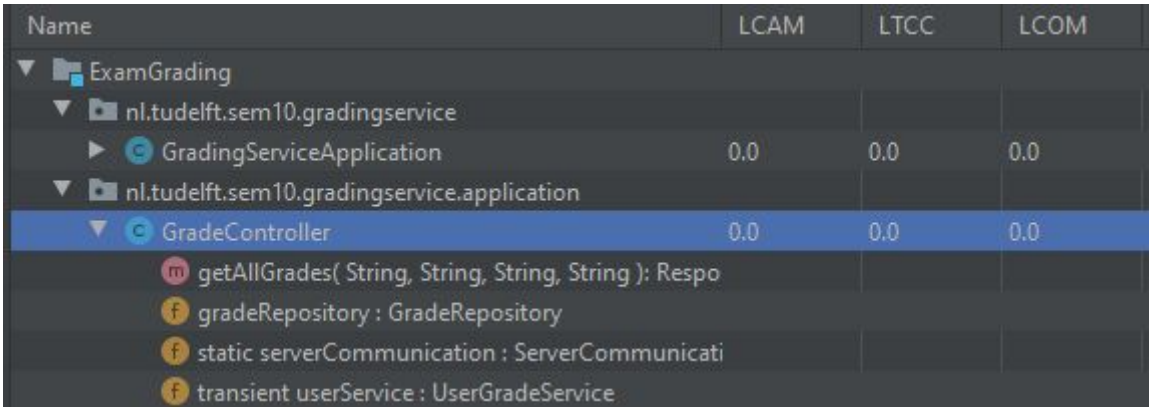
Grading Service GradeController

The GradeController class originally had a lack of tight class cohesion. This was mainly caused by three methods completely unrelated to the rest of the class or the application for that matter. Removing these methods then of course improves class cohesion, which is what was done to fix the issue.
As mentioned, these methods did not have anything to do with the application as they existed only for testing purposes. The methods in question were setters for some of the class's fields, but they were only used for testing to use mocked instances. This means that although the original application still works, removing the methods broke some tests. To fix these, the field values were set using reflection rather than using setters in our tests.

Before



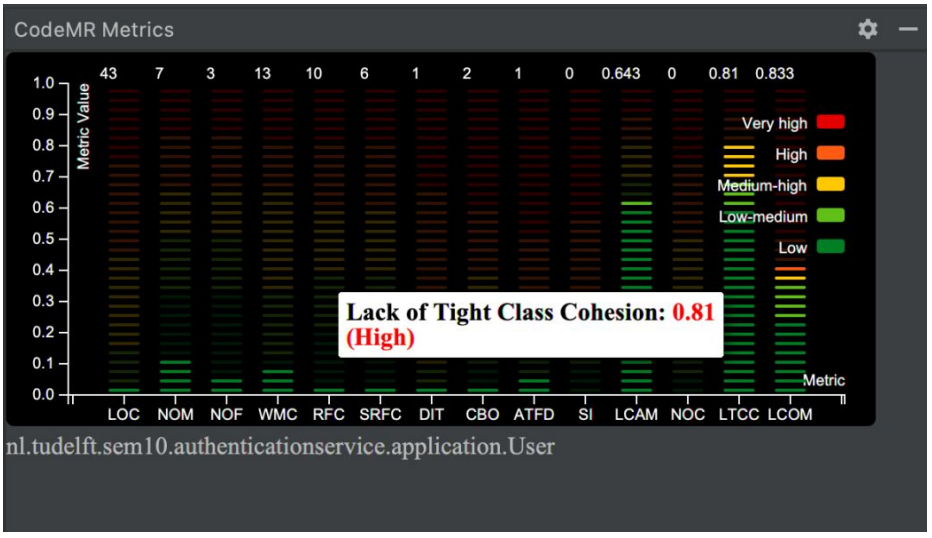
After



Authentication Service User

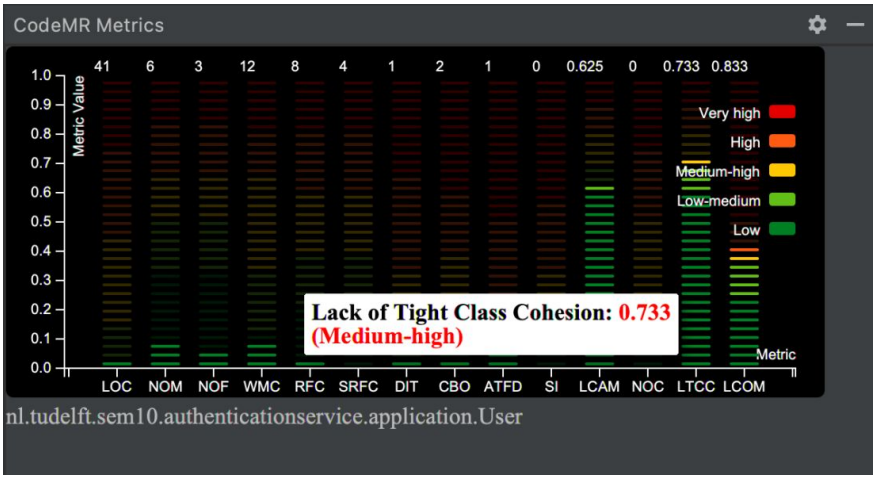
The ‘User’ class in the authentication service serves as an entity class to model the information that is needed on a user interacting with the microservice. CodeMR complained about this class having a lack of tight class cohesion (LoTCC of 0.81). Meaning the public methods do not interact much with each other.

Before



Of course in entity classes this is the case most of the time because simple getters and setters don’t call each other. It was still thought the ‘User’ entity class was due to some refactoring and thus decided to remove the hashCode method. This method was never used and thus safe to remove. Removing this method ensured that there were less methods that did not interact with each other resulting in a lower LoTCC (0.733).

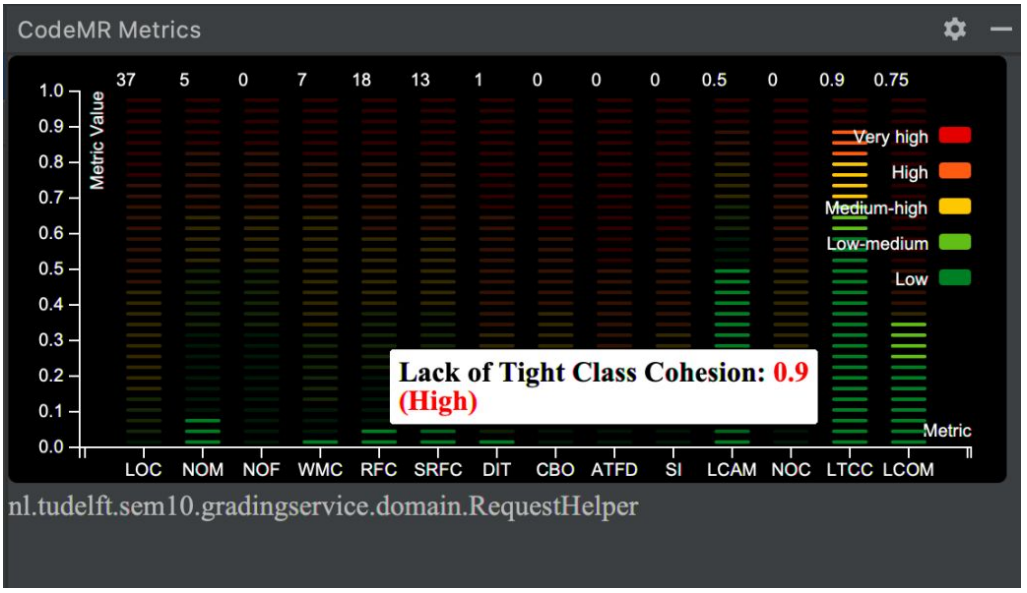
After



Grade Service RequestHelper

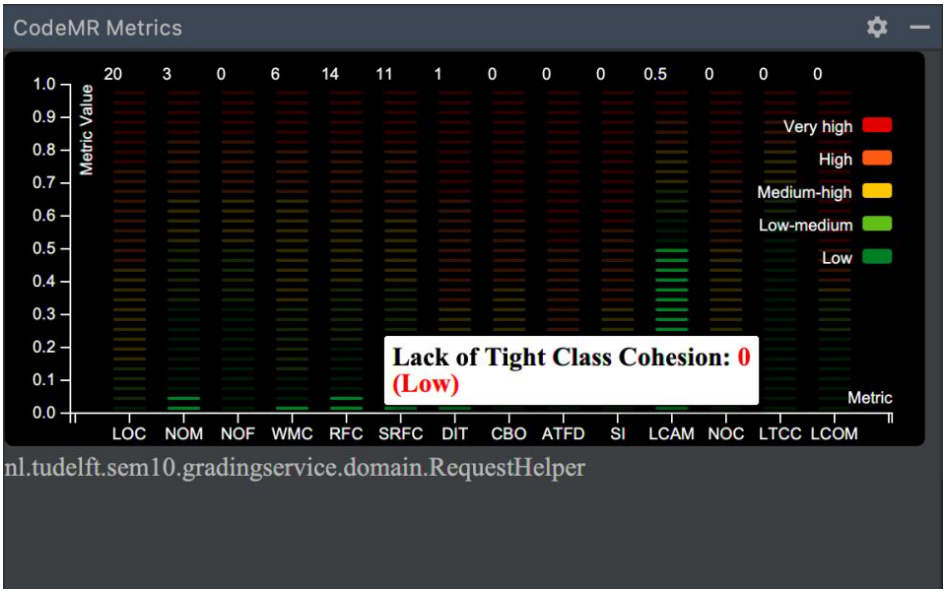
The ‘RequestHelper’ class eases the use of REST requests. This class received a high risk for the Lack of Tight Class Cohesion metric (0.9)

Before



It was decided to refactor the class to lower the metric value and because some duplicate code was present. The class was refactored by generalizing a request builder method and thus making it possible to remove other methods. This made it possible to get rid of the duplicate code and lower the lack of tight cohesion in the class.

After



Method Level Fixes

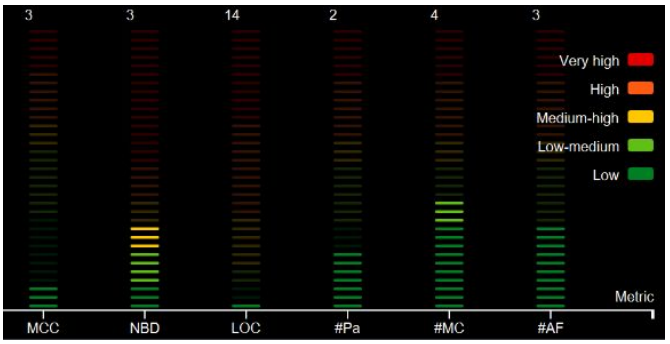
Course Service CategoryServiceImpl#getWeights(String)

This method looks up all course categories for a given course and returns all category name and weight pairs.
Originally, this method requested all known entries from the repository and manually filtered and formatted the categories. Doing so resulted in a high amount of methods called inside the method, which CodeMR rightfully complained about.
To fix this, part of the processing got moved elsewhere. Using a dedicated query, all categories from a given course can be requested at once from the repository, meaning this part of the method logic (filtering) is handled by the database. As a result, the method itself just needs to format the entries, requiring less logic and thus less method calls. Additionally, having the database filter entries improves performance as database drivers are designed to handle filters efficiently and less data needs to be sent to the application.

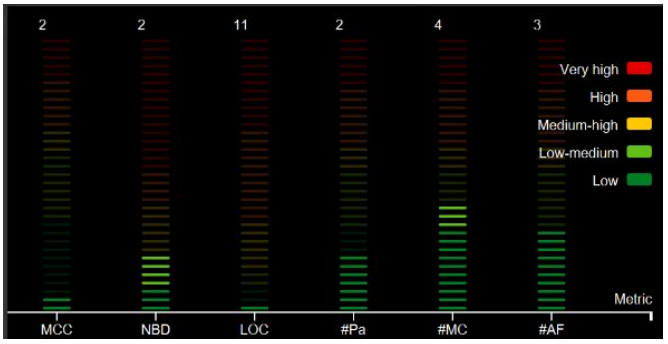
Grade Service StudentGradeController#passingRate(String, String)

This method looks up the passing rate for a given course code. Originally, this method had a try-catch block and an if statement within. This resulted in a relatively high nested block depth of 3. CodeMR classified this as medium-high severity. After the simple fix (removing the try-catch block), a nested block depth of 2 was achieved, which was classified as low-medium.

Before



After

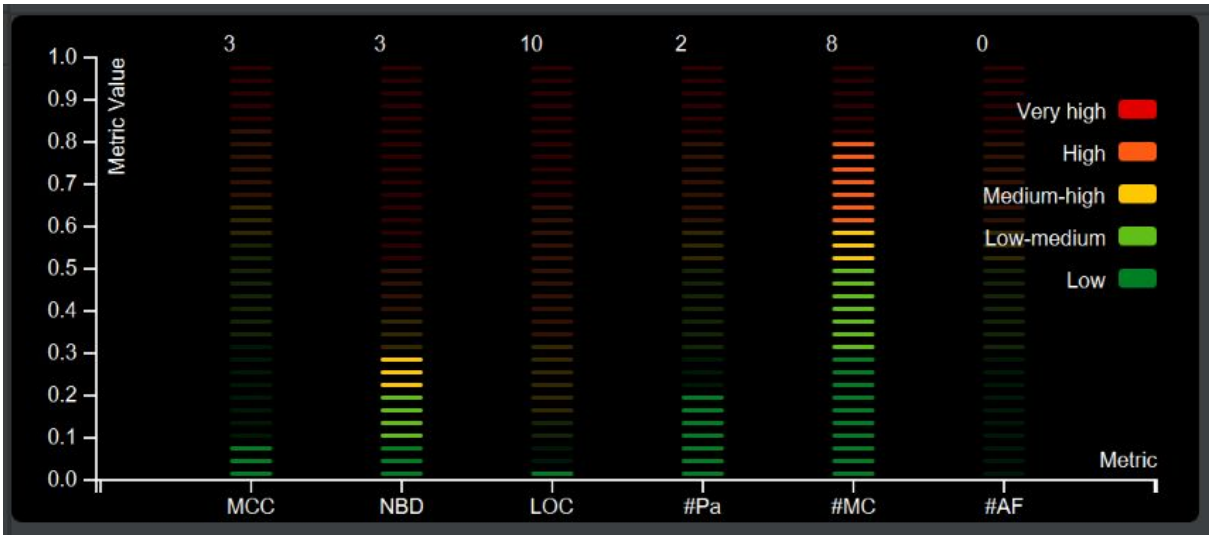


Grade Service RequestHelper#sendRequest(HttpRequest, HttpClient)

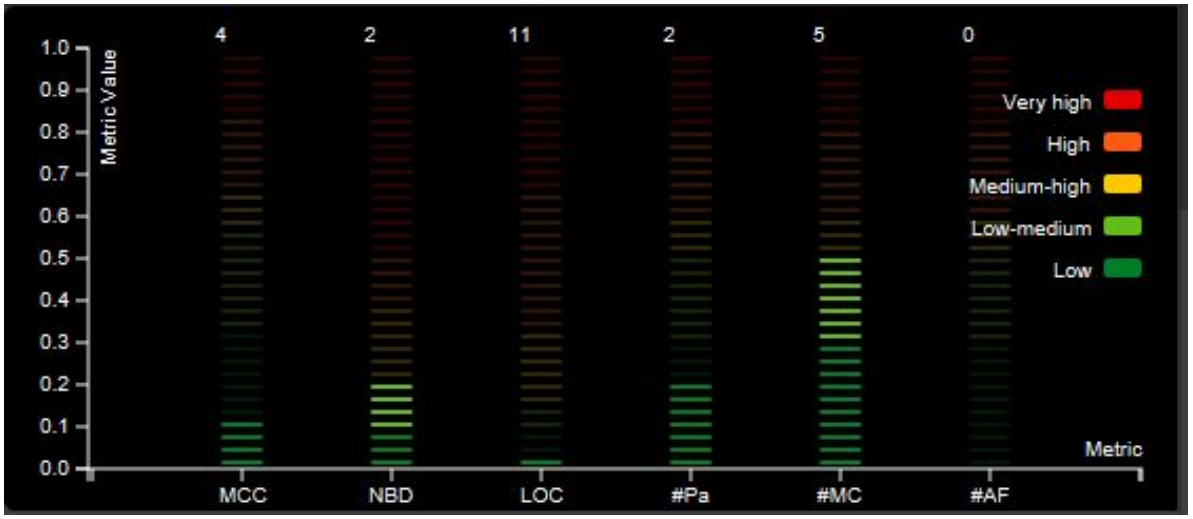
SendRequest is a method in RequestHelper class that actually sends the HttpRequests input as its parameter and receives a response. It would then return the contents of the response body in the case that it was not null or empty.

As can be seen below the before metrics of the method show that there is a high number of methods called being 8, and a medium high value for NBD being 3.

Before



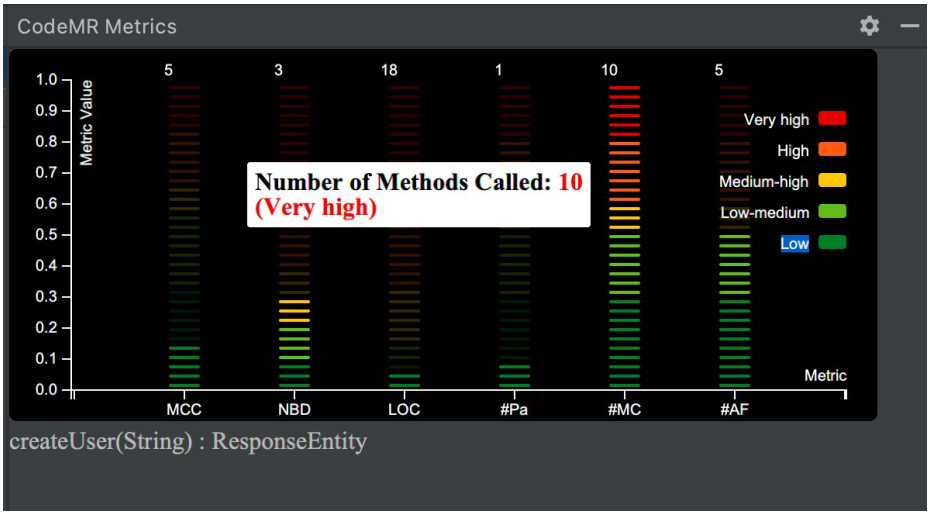
The improvement made here was getting rid of unnecessary method calls to response.body(), which was called two times and instead could just be called once and stored in a variable. The method call e.printStackTrace() was also removed since debugging here was not necessary. Finally the last method call fix was removing the call to Integer.asString() and instead just doing String str = “” + int.
The next change made was moving the if statement outside of the try catch block. By doing this it improved the NBD, and was a good change because it was unnecessary to run the actual code inside of the try catch block since its purpose was to catch any exceptions during cross server communications and not exceptions within the code. The result of the refactoring is as below:
After



To summarize, MC, number of methods called reduced from high at 8 to a low medium at 5 and NBD, nested block depth was reduced from medium-high at 3 to a low medium at 2.

User Service UserController#createUser(String)

The createUser method is used to insert a new User into the database. It takes a JSON String as input, turns it into a User object while applying necessary error handling and finally makes a call to the UserRepository to insert the user. Because there is a conversion from a JSON-formatted String to a User object and the error-handling between, the number of methods called was very high. This was the result of the metric before refactoring:
Before



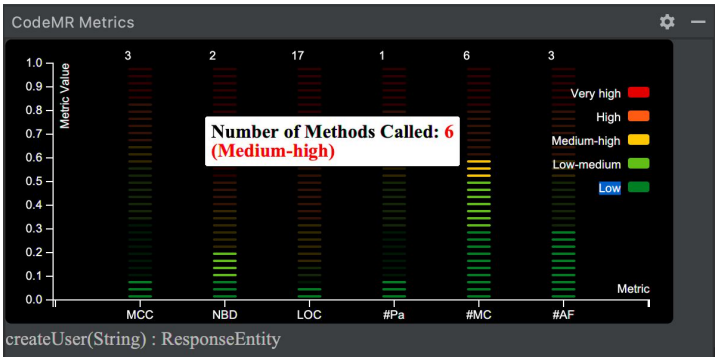
As can be seen from the metric, the result is very high. The cause of this code smell was mainly from JSON operations like these:

```
if (json.has(netIdStr) && json.has( key: "password") && json.has( key: "type")) {
```

```
String password = json.getString( key: "password");
String encrypted = Utility.getEncryptedPassword(password, restTemplate);
int type = json.getInt( key: "type");
```

So we decided to add a method that deals with these JSON operations to the Utility class. This method converts a JSON String to an array of User fields, of type String. This utility method can be found in this directory: UserService/domain/Utility/jsonStringToFields. So the code was refactored to make a call to this method in the beginning and receive all the fields with just one call. The group also refrained from calling other methods where it's not necessary. As a result the following metrics were achieved:

After

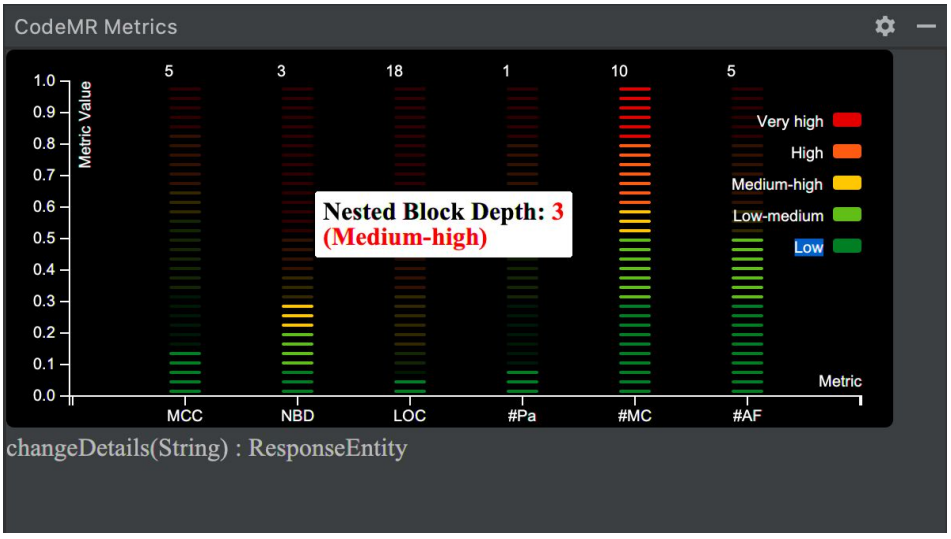


It's still medium-high but there's quite an improvement from 10 methods called to 6. The remaining calls to other methods are necessary to preserve the functionality of the createUser method. Splitting the method into several other methods that don't serve any meaningful purpose or use, will not further improve the code quality. Also the threshold of the metric can be adjusted to reflect the code quality better. So with the use of this new utility method, the code quality of the createUser improved. This also prevented code duplication since another method inside the same class, changeDetails, makes use of the utility method as well.

User Service UserController#changeDetails(String)

The change details method is used to update an existing user in the database. It takes a JSON String as input, which should contain the necessary information about a user to change its password or role, does error handling in the cases where the user doesn't exist or the provided JSON format is invalid, e.g. the necessary fields do not match. Before refactoring, the 'if clauses' to handle these cases were nested. Which increased the nested block depth, which in result decreased the code quality. Here's the result of the CodeMR metric applied to the method before refactoring:

Before



As can be observed from this report, both the nested block and the number of methods called metrics are problematic. The "number of methods called" issue was fixed with the same methodology discussed above in the createUser method level fix. In this section the focus will be on the Nested Block Depth metric.

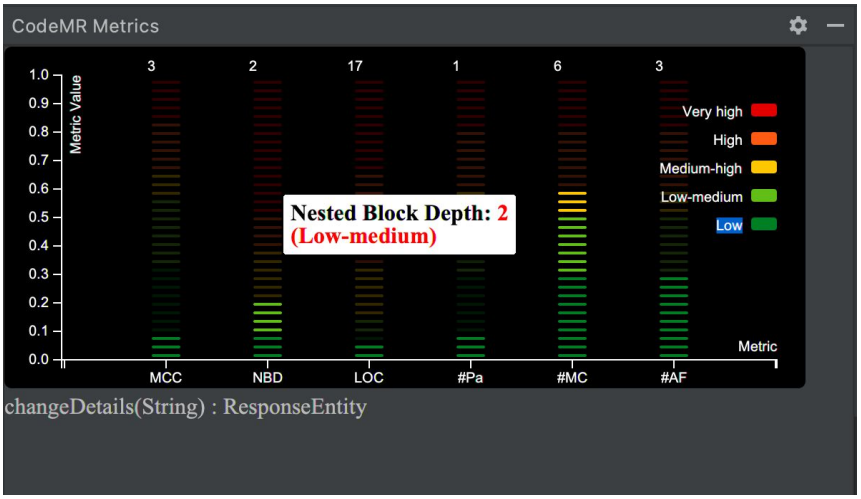
Here's a partial image of the changeDetails method before refactoring:

```
158        if (json.has(netIdStr) && json.has( key: "password") && json.has( key: "type")) {
159            String netId = json.getString(netIdStr);
160            User u = userRepository.getUserByNetId(netId);
161            if (u == null) {
162                return new ResponseEntity<>(HttpStatus.NOT_FOUND);
163            }
```

As can be seen from this picture, line 161 was the main problem regarding nested block depth. So in order to fix it, the 'if clauses' were extracted outside, the checks inside were altered to be able to execute them serially, one after the other. In order to prevent this document from getting too long, it was decided not to include the final images after refactoring. You can find the final version of the method by visiting the same directory mentioned above in our final commit.

Here's the result of the metric after refactoring:

After



As can be observed from this image, the Nested block depth is reduced to low-medium after refactoring operations. This improvement contributed to achieving better code quality.

Conclusion

Upon the first inspection through the CodeMR tool it was noticed that most of the metrics were in the low or low-medium range which presented some issues.The few issues noticed in the current master branch were fixed but these weren't nearly enough meaningful non trivial refactoring operations. After some discussion with the TA it was decided to look through previous versions of the code to discover if the past refactoring had actually improved any metrics without having used codeMR. This provided a lot more options to choose from, however even using the past code a few more class level changes needed to be made in order to finish the report. It was then decided to do some refactoring even if this did not improve any of the metrics provided by CodeMR.