SEM Assignment 2 - group 33A

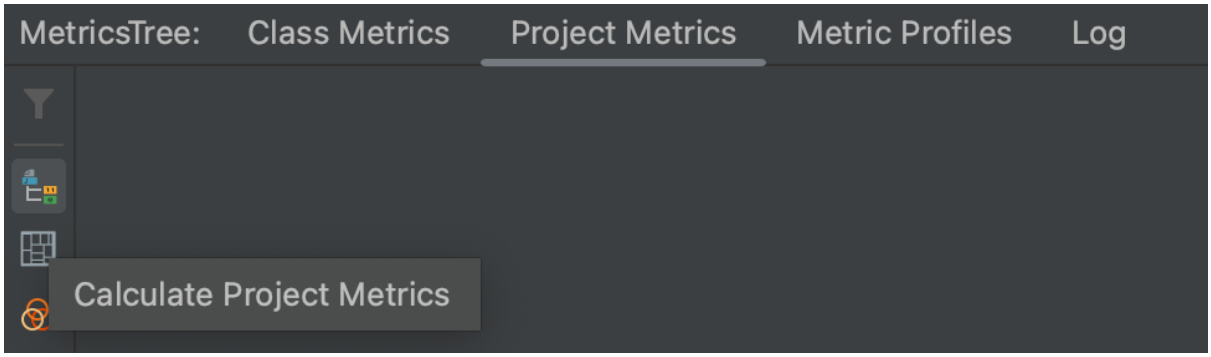
Laimonas Lipinskas (5559375)
Lotte Kremer (4861957)
Nicolae Radu (5527104)
Razvan Loghin (5480620)
Dawid Plona (5205018)
Vlad Nitu (5529557)

January 18, 2023

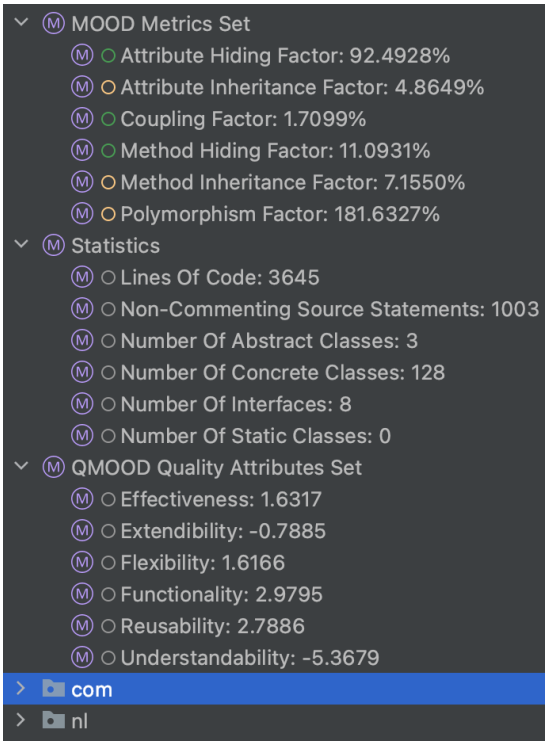
Task 1

To solve this assignment, we have decided to compute our project's code metrics using the 'MetricsTree' IntelliJ plugin. We played around with the tool in our project, computing code metrics at both the class and method levels.

We started our investigations in MetricsTree by computing the Project Metrics:



The results of this gave us an insight about what project metrics we should follow along the refactoring process and what were the strengths and the weaknesses of our system before starting refactoring (for example: The coupling factor seemed pretty low, being marked as 'green', representing that this is a strength, while the Method Inheritance factor was higher than expected, which meant that we should consider focusing on the Number of Children (NoC) metric and other problems related to inheritance):

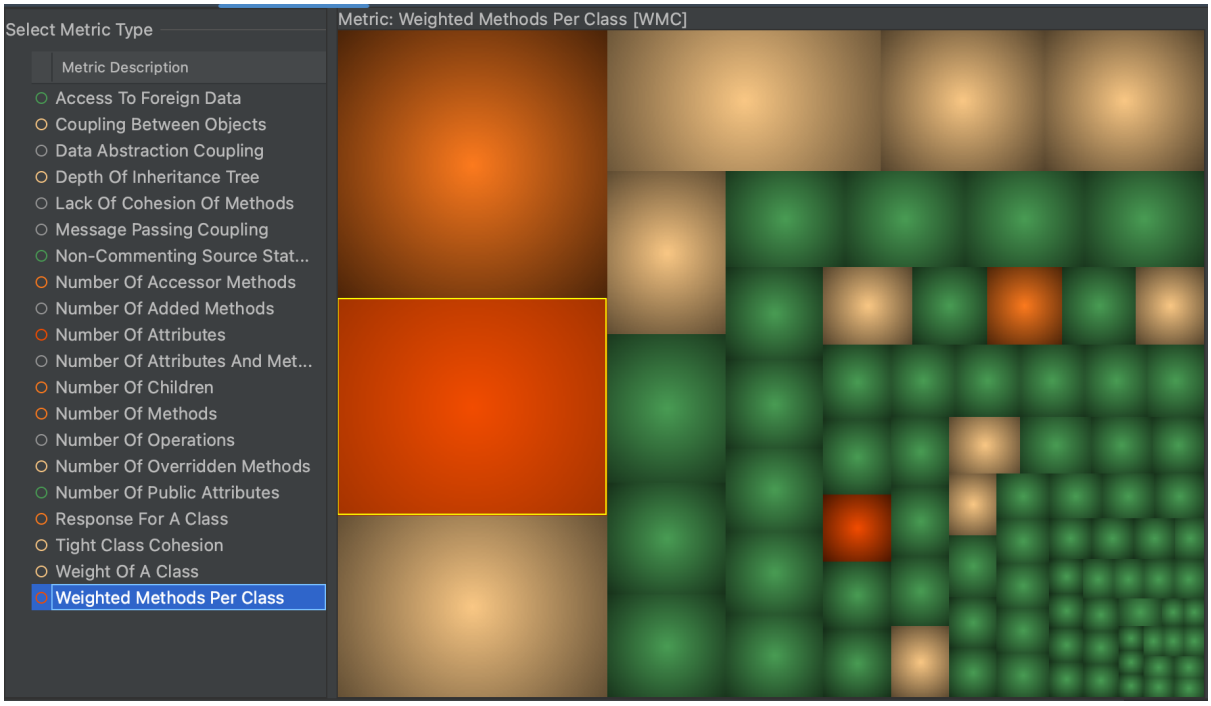


This gave us the opportunity to read detailed descriptions aboute each software metric we were intereseted in improving or that we did not know about before, and how each score is computed

(for example: $Extendibility = 0.5 * Abstraction - 0.5 * Coupling + 0.5 * Inheritance + 0.5 * Polymorphism$), and what these scores explain (for example: *Extendibility* refers to the presence and usage of properties in an existing design)

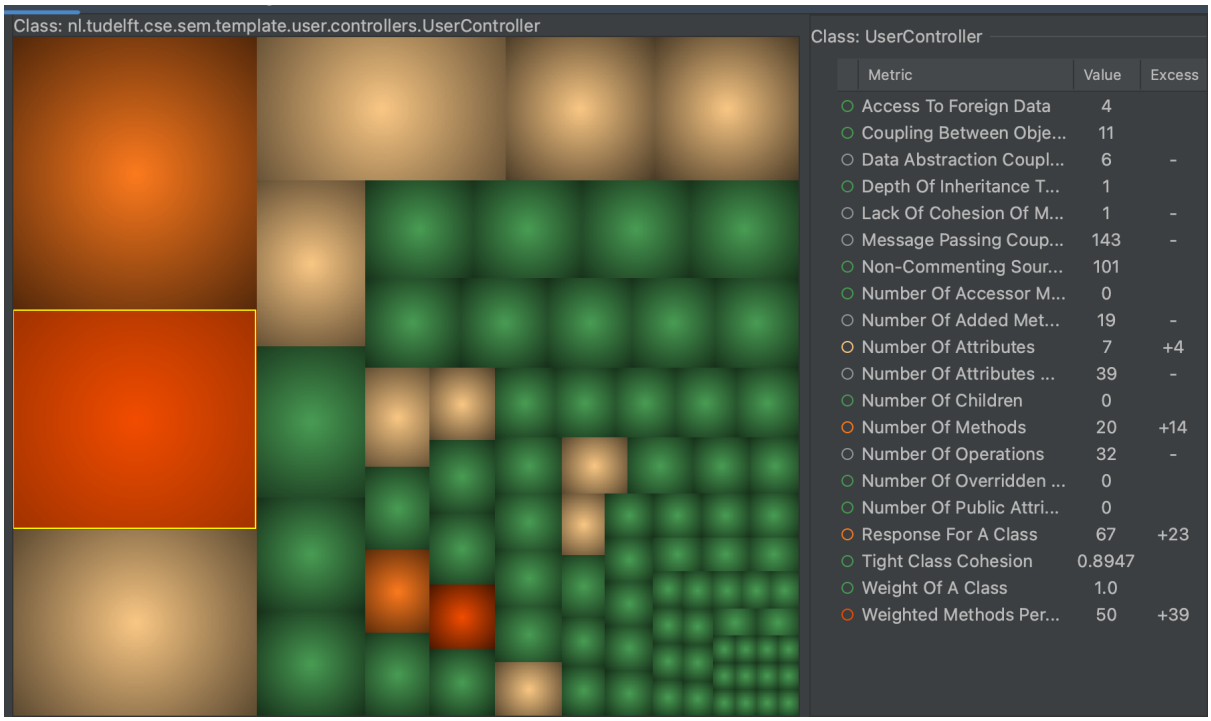
Moreover, we could access each package and check dedicated analysis conducted on the classes we are interested in (for example: analyzing some metrics related to the 'MatchingController' class from the Matching microservice):

In order to ***visualise*** the code smells that were distributed around classes of all our 5 microservices, we made use of the "Build TreeMap with Metrics Values Distribution" feature of MetricsTree (the button that is directly under the Calculate Project Metrics in the first screenshot we provided). The following treemap was outputted (note that these give ***software metrics related to classes***, the ones related to methods will be described later):



Basically, we were provided a treemap for each Metric Type that MetricsTree is able to analyze, such as "Number of Attributes", "Weighted Methods Per Class (WMPC)", "Response for a Class", etc. For a given treemap, there were different colored boxes per class of our system that indicated whether further improvements were necessary (or possible) when it comes to the metric type we were analyzing (for example, for the 'UserController' class, that is represented by the orange-red box that we have our cursor on (surrounded by the yellow margins), MetrisTree suggests that it has a high "Weighted Methods per Class" score, so we should consider refactoring this class).

Additionaly, clicking on a given box (that represents a certain class from our system) gave us even more insights about all the software metrics that were computed for it, which allowed us to plan the refactoring process, by having a wider view over each component of the system, as we targeted multiple metrics per class, thus increasing our efficiency throughout this extensive process:



In this specific example, we can see that not only the WMPC was in excess, but also the "Response for a Class" and the "Number of Methods". Thus, we will focus on restructuring the "UserController" class, such that all these three metrics will be reduced to state-of-the-art values (having metric scores depicted in "green" or "pale yellow").

We thoroughly checked all metric descriptions and the class-specific metrics scores, but as there was a large number of values that we had to work on improving ($metricTypes * classes$), we had to prioritize the way we conduct our refactoring process. That being said, it was infeasible to refactor all the classes and methods, and to (blindly) achieve green metric values for each type, so we decided on the following thresholds in order to identify the room for improvement.

As a general rule, we are going to target reducing the metric score for the **red** and **orange marked metrics**, so those that were the most smelliness. However, we always want to reason whether the smell indicates a problem or not, as it may represent a false positive that the Machine Learning algorithm of MetricsTree suggests, or it may be something that we are not interested in refactoring / something that is impossible to refactor (such as a change that would reduce the number of attributes requiring a lot of other changes in different parts of the system, that may break our functionality, and we are not willing to take such a risk). Nevertheless, we will continue checking the **pale yellow metrics** and if there is an elegant way to refactor such a class or method, that does not necessitate a lot of effort and we see the relevance of such a refactoring, we will definitely go for it, and afterward argue about our decision. Our golden rule during the refactoring process was: never blindly check the numbers/scores of the metrics, always try to reason about them and come up with an argument about whether this

refactoring is necessary or not, whether the trade-off between the advantages resulting from this re-engineering step overcomes the effort, time spent and risk of introducing new bugs

We uniformly split the refactoring tasks among all 6 teammates, each of us having to refactor at least one method and at least one class (even if the method was inside that class, we consider class refactoring and method refactoring separately, as we will describe how we tackled this two problems in the next paragraph). Moreover, we tried to stick to refactoring the microservices that we have worked on, that is each member should try to refactor the classes and methods from the microservice he/she was assigned to.

Besides class metrics that we have seen so far (in the previous screenshots), there are also ***software metrics related to methods*** that MetricTree is able to provide, and we are going to analyze each of them while refactoring a specific class. In order to obtain these, after choosing the class we want to refactor, we can use the 'Class Metrics' feature from MetricsTree, which returns us both the classes problems, and also each method problem from that class. The ones annotated with the small 'm's are method refactoring problems (black m in a red circle) as depicted in figure 1, while the blue capital 'm's are the class refactoring problems, depicted in figure 2.

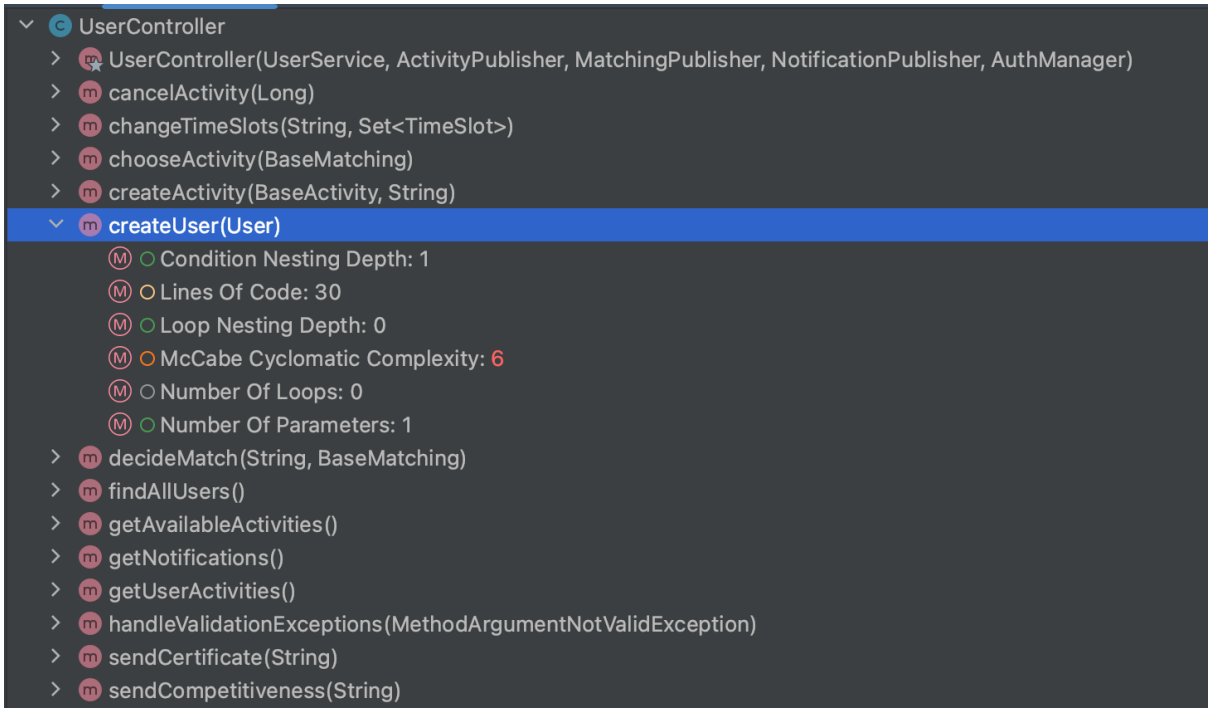


Figure 1: Method-level refactoring metrics

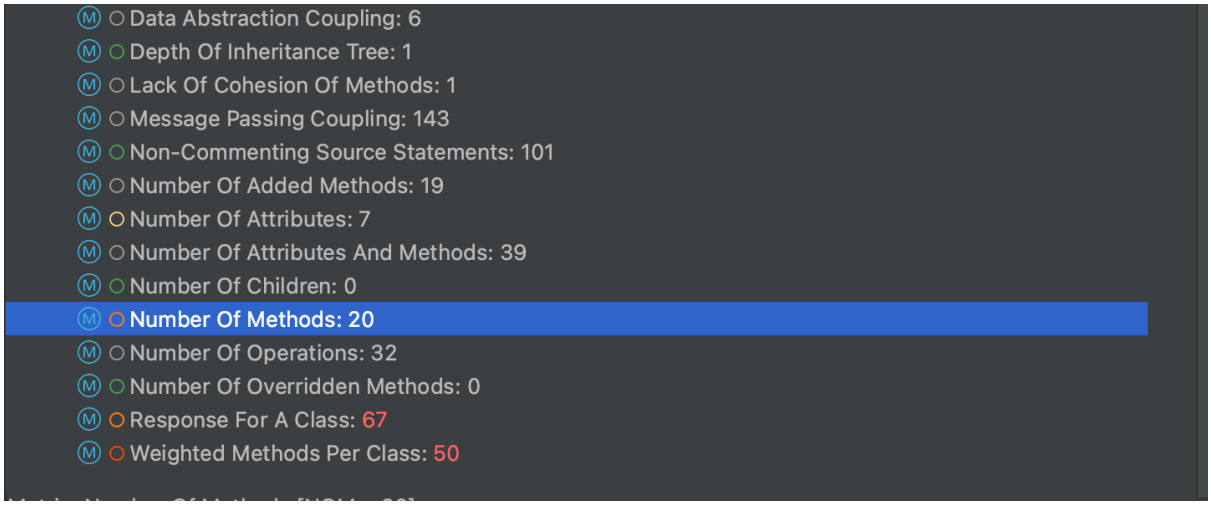


Figure 2: Class-level refactoring metrics

Task 2

Classes impossible to refactor

Besides all the methods and classes that we have refactored and we are going to explain our reasoning for taking those decisions later in this chapter, we were unable to refactor:

1. Equals method provided by @EqualsAndHashCode from Project Lombok or those auto-generated by IntelliJ. Those have a high Cyclomatic Complexity (CC) metric score, due to all the if-statement checkings that are conducted in the source code. That being said, some classes that have a reasonable number of parameters (for example: 5) will get a red-score for the CC metric, this being something that we could not reduce (while also keeping the entire functionality intact).

2. Number of attributes of classes that extended the "Exception" class (such as *UserIdAlready-InUseException* Exception class that we've created in the Authentication microservice, situated in the *domain* package). MetricsTree counts as attributes all the attributes present up in the inheritance tree. Thus, our class even though it defines a single attribute, there are 16 atttributes considered by our refactoring tool.

3. Number of children of *BaseValidator* class from the Activity microservice, situated in the *validators* package is considered by MetricsTree large (thus getting an orange metric score). We could not reduce this, as the BaseValidator is part of the Chain of Responsibility design pattern that we decided to implement, and it has 5 "stages" of validating, thus resulting in 5 classes extending this base validator. Therefore, there is no way we can achieve a smaller number of children classes for BaseValidator. Note that we implemented the *Chain of Responsability* design pattern as presented in the lecture slides, so by following the course guidelines.

4. The authentication and security configuration classes (such as *RequestAuthenticationConfig* and *WebSecurityConfig*) are considered to have too many attributes by our refactoring tool. We were unable to refactor these classes, as these were mainly implemented in the template project (and later we came up with some tweaks). The impossibility of refactoring came from the fact that each of these two classes was extending the *WebSecurityConfigurerAdapter* Spring

Security class, and the number of attributes considered by MetricsTree was also taking into consideration this Spring Security class number of attributes (same as in 2. explained above).

User microservice

We refactored the whole *UserController* class in the *Controller* package. The reason for this was that this class was too large. This was indicated by multiple metrics computed by MetricsTree, as can be seen in Figure 3. There were 20 methods and 'Weighted Methods Per Class' was indicated with 50, which means that the complexity was quite high. Also the Response For A Class, with 67, was indicated as orange by MetricsTree.

To find a solution for this we considered all the functions in UserController and divided them based on their functionality. All the requests / methods that were associated with the Activity microservice were put together in UserActivityController, everything with the Matching microservice in UserMatchingController and same for UserNotificationController. All the requests that did no use any publisher and thus didn't have an interaction with another microservice afterwards are still in UserController. This way the classes are all easier in use and they now all have a clearer purpose. This also resulted in better results in the MetricsTree as can be seen in Figure 4. The three new classes all had a score on MetricsTree of green or yellow as well.

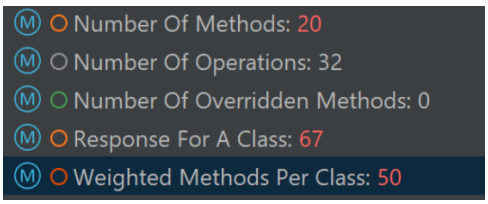


Figure 3: Before refactoring UserController

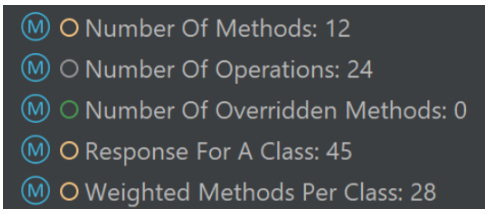


Figure 4: After refactoring UserController

For the *BaseActivity* class that can be found in the *utils* package, we have improved the class-level metrics by applying the following refactorings: reduced the *WMPC* (weighted Methods Per Class) by getting rid of redundant methods (such as unused getters, setters, equals and hashcode methods). This also implied reducing the *Number of Methods* metric, as it can be depicted in figures 5 and 6:

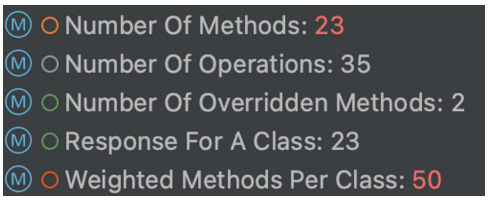


Figure 5: Before refactoring BaseActivity

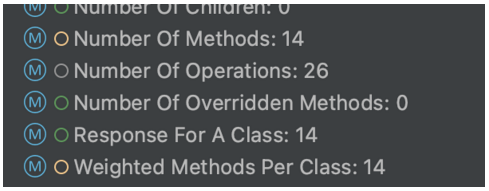


Figure 6: After refactoring BaseActivity

One key method from the *UserController* class we refactored is *createUser*. We have improved the method-level metrics by applying the following refactorings: reduced the *CC* (Cyclomatic Complexity) by delegating a part of the validation step to a newly created *validateId* helper method. Additionally, we managed to reduce the *LOC* (Lines of Code) metric, as shown in figures 7 and 8:

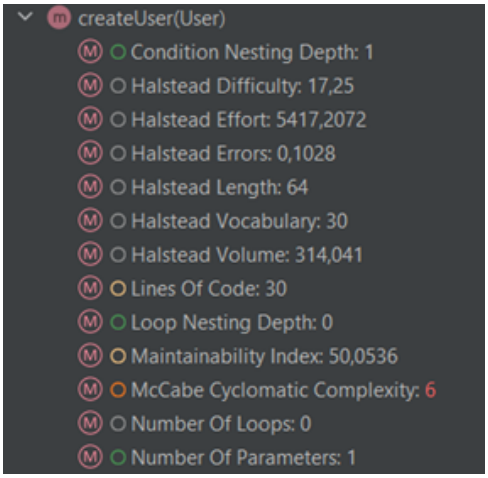


Figure 7: Before refactoring createUser

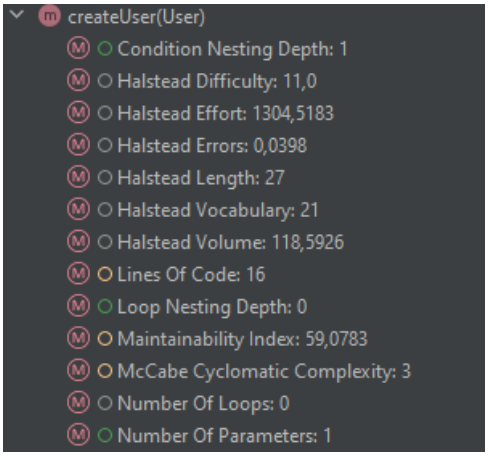


Figure 8: After refactoring createUser

Simultaneously, it appeared sensible to refactor the *InputValidation* class from the *utils* package, so that it can be utilised more effectively with the previously mentioned new validation method, and allow for more seem-less extensibility of the validation step in the future. The metric-wise improvements include lowered Number of Attributes and Methods, lowered Number of Operations, and lowered *WMPC*, as shown in figures 9 and 10:

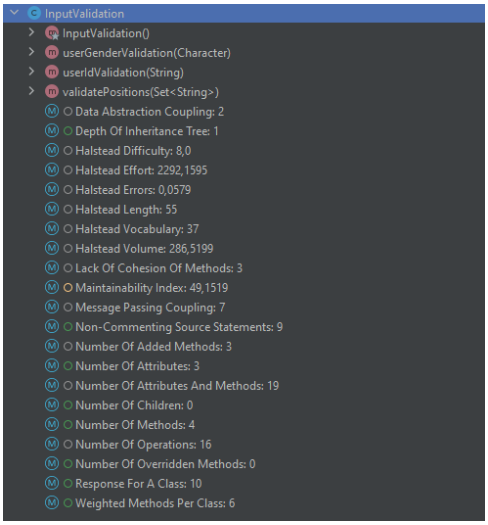


Figure 9: Before refactoring InputValidation

Figure 10: After refactoring InputValidation

Moreover we also made a small change in the *createActivity* in the new *UserActivityController* class. The cyclomatic complexity was 5 and could thus be improved. One of the checks that was performed in this method was if the input was either 'training' or 'competition'. We made a new method for this and thus seperating the check from the createActivity method. The cyclomatic complexity was lowered by one by this refactoring and is now 4.

Matching microservice

For the *MatchingContorller* class that can be found in the *controllers* package, we have improved both class-level and method-level metrics by applying the following refactoring methods: the all arguments constructor (that was improved by reasoning about method-level metrics) now contains a smaller number of parameters (so we decreased the Number of Parameters (NoP), thus improving the NoP metric).

This was achieved by grouping all the Publisher class attributes used by the Matching subsystem (ActivityPublisher and NotificationPublisher) into a "CollectionPublisher" class. By creating this additional *Parameter Object*, we obtained an acceptable number of arguments for our constructor method. Take a look at the comparison beetween figures 11 and 12

Figure 11: Before refactoring MatchingController constructor

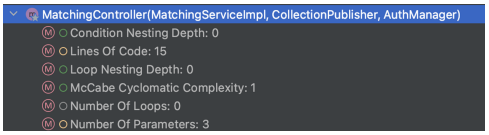


Figure 12: After refactoring MatchingController constructor

In terms of class-level refactoring, by applying this "Extract into class" refactoring technique we managed to decrease the number of attributes of the class under refactoring, as it can be depicted from figures 13 and 14

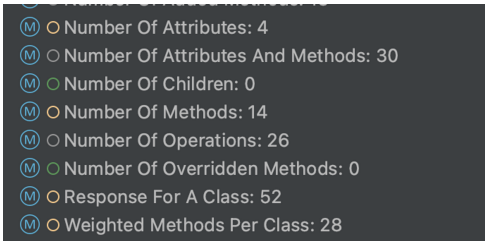


Figure 13: Before refactoring MatchingController class

Figure 14: After refactoring MatchingController class

Another change in the MatchingController class was to the chooseMatch method. The reason for this change was due to Cyclomatic Complexity being too high. In order to remedy the issue a new method was created named securityCheck which handles the responses to unauthorized calls to the first method.

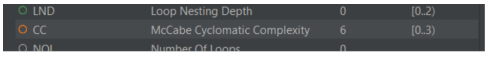


Figure 15: Before refactoring chooseMatch method

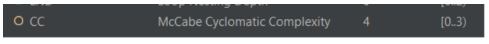


Figure 16: After refactoring chooseMatch method

For the *BaseNotification* class that can be found in the *utils* package, we have improved the class-level metrics by applying similar refactoring techniques as for *User/utils/BaseActivity*. Thus, we managed to reduce the WMCP and the Number of Methods, as depicted in figures 17 and 18 :

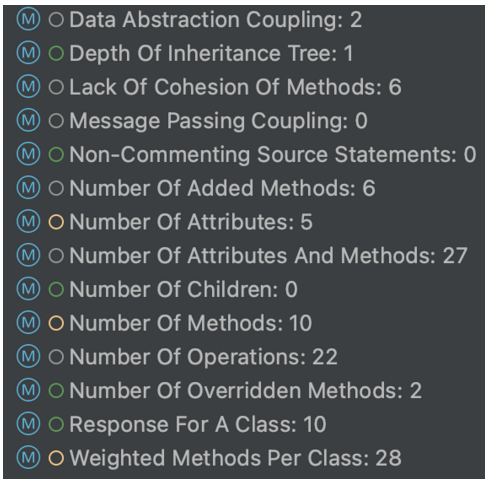


Figure 17: Before refactoring BaseNotification

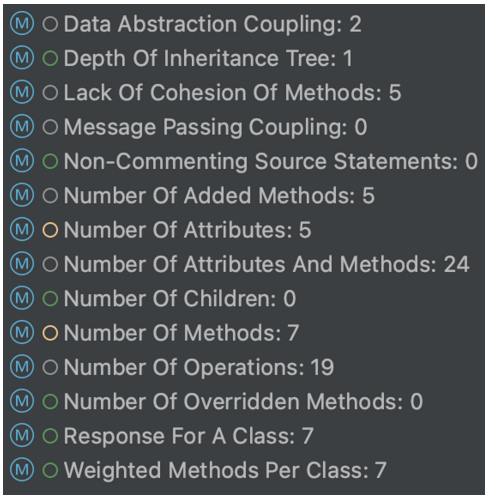


Figure 18: After refactoring BaseNotification

We also came up with a small class-level refactoring for the *Pair* class that can be also found in the*utils* package, for which we removed unused methods (such as *@ToString* Lombok's annotation that we've used for debugging purposes), which resulted in improving the *Number Of Methods* metric.

Another change was done to the notifyUser method inside the NotificationPublisher class. A method-related metric that stood out was the number of parameters which was totaling up to five. To resolve this issue, an instance of the BaseNotification class was used which encapsulates all the previous parameters.

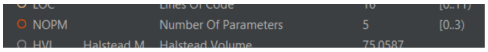


Figure 19: Before refactoring notifyUser method

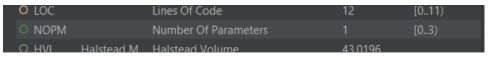


Figure 20: After refactoring notifyUser method

Finally, the last method modified in the Matching microservice was the method filterTimeSlots in the FunctionUtils class. The primary problem related to this function was the Cyclomatic Complexity metric. In order to reduce it, a binary operation was used instead of an if statement while maintaining the same behaviour.

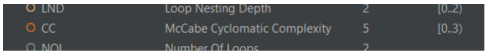


Figure 21: Before refactoring filterTimeSlots method

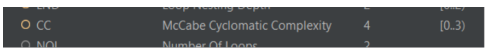


Figure 22: After refactoring filterTimeSlots method

Activity microservice

The first problem that was solved by refactoring in the *Activity Microservice* was the *McCabe Cyclomatic Complexity (CC)* of the **"check"** and **"sendAvailableActivities"** methods, which were too big: 10 and 9. We succeeded in lowering it to the value of 5 by creating a helper method named **"getValidator"** that takes care of choosing the right Validator with reference to a given Activity and Position. This was the process that was building most of the CC.

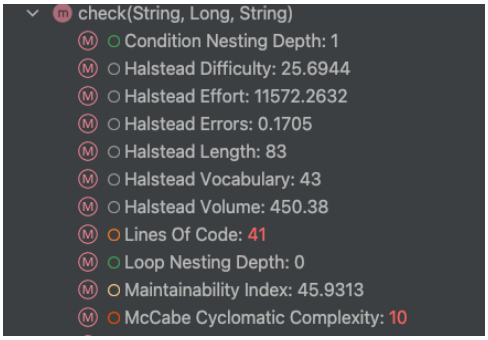


Figure 23: Before refactoring "check" method

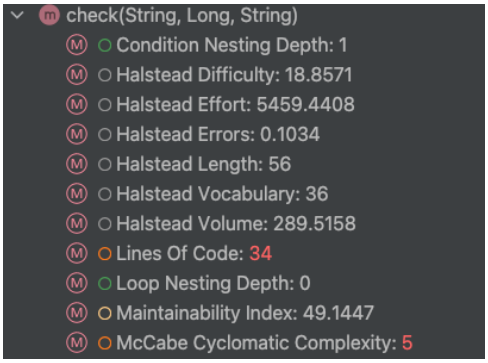


Figure 24: After refactoring "check" method

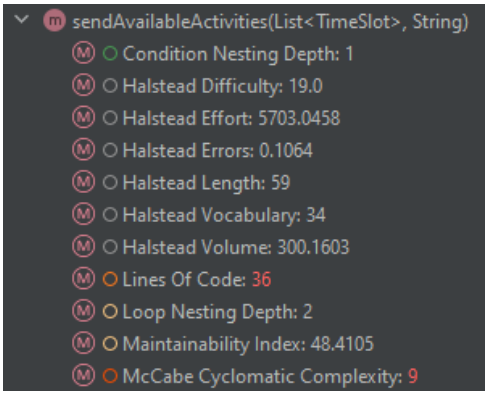


Figure 25: Before refactoring "sendAvailableActivities" method

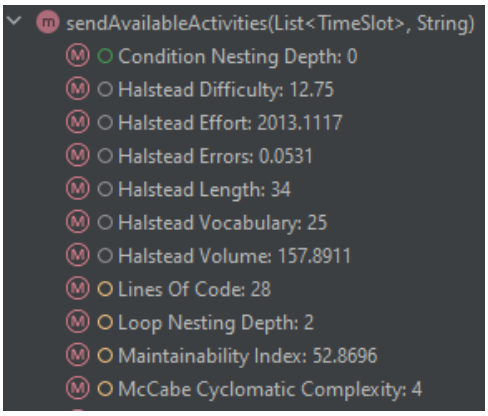


Figure 26: After refactoring "sendAvailableActivities" method

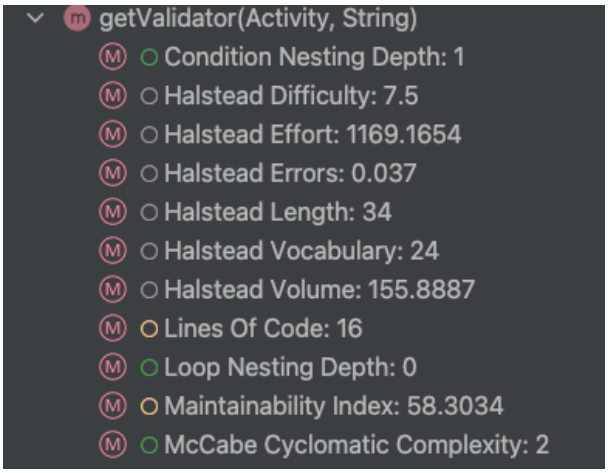


Figure 27: "getValidator" helper method which despite being responsible for lowering the CC of the "check" method has just a CC of 2

Another change was done in the *Validators* package, where the **checkNext** method from the *BaseValidator* class and the **handle** methods from all the Validators classes were having too many parameters (NOP was too high). The solution here was to encapsulate the parameters into an object. Instead of passing four separate parameters, we created a new class *ActivityContext* that holds all the necessary information and passes an instance of this class to all the methods

that use those parameters. This reduced the number of parameters in the method signature and made it easier to understand the purpose of each parameter.

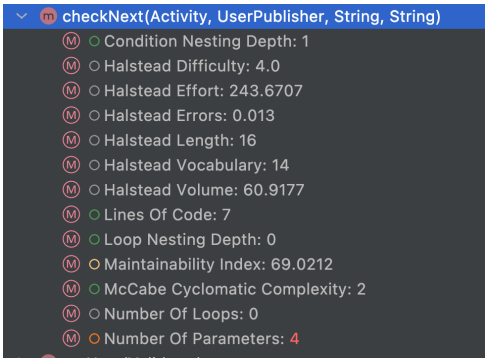


Figure 28: Before refactoring "checkNext" method

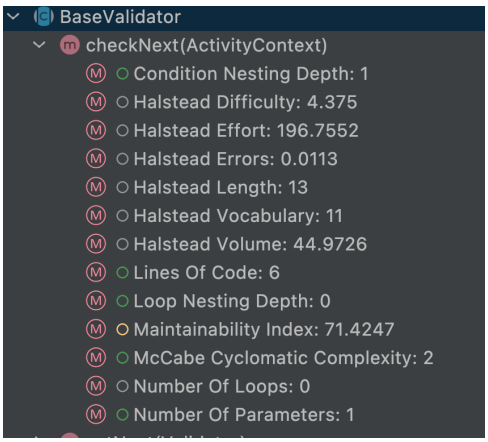


Figure 29: After refactoring "checkNext" method

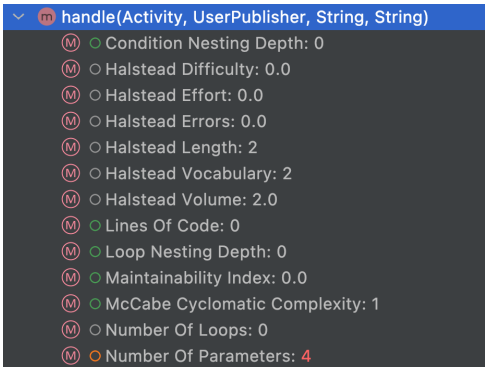


Figure 30: Before refactoring "handle" method

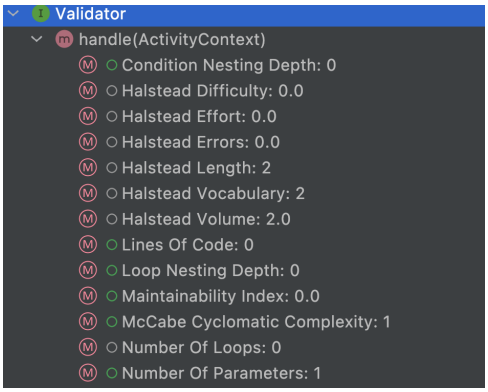


Figure 31: After refactoring "handle" method

Notification microservice

For the *Notification* class that can be found in the *Domain* package, we have improved *WMPC* (weighted Methods Per Class) by getting rid of redundant methods (such as unused getters, setters, equals and hashcode methods). Moreover we removed a variable that was not used anymore because the information it provided could be acquired by using the authorisation token. These changes resulted in reducing the *Number of Methods* and *Number of Attributes* metrics.

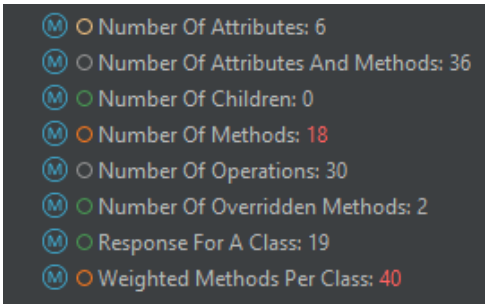


Figure 32: Before refactoring Notification class

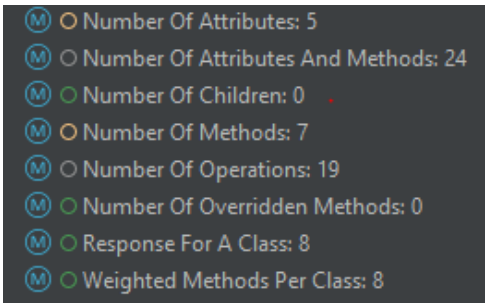


Figure 33: After refactoring Notification class