



CSE2115 Software Engineering Methods

---

## Assignment 2

---

### AUTHORS

Bogdan-Andrei Bancuta - 5507340  
Alexandru Gabriel Cojocaru - 5520967  
Khalit Gulamov - 5491541  
Alexandru-Nicolae Ojica - 5477484  
Wing-Yan Joyce Sung - 5011825  
Efe Unluyurt - 5452481

18 January 2023

---

## 1 Choice of parameters and thresholds

In order to refactor the code the metrics of the code, which are important for code quality and the values these metrics would be considered bad had to be determined. For ease of use we first looked at the *MetricsTree* plug-in, which can be used with *IntelliJ*. This plug-in provided us with code metrics to consider for our code. To reduce the number of metrics to look at, we take only the metrics considered in [1], which are also present in the metrics calculated by the *MetricsTree* plug-in. Then we filtered these metrics by looking at the metrics of our code. We decided to focus on the metrics marked as bad by the *MetricsTree* plug-in. This means some of the metrics mentioned by [1], which are also calculated by the *MetricsTree* plug-in are not included in the metrics we chose to focus on. In table 1 these metrics are listed with their threshold calculated by [1], whether these metrics are considered class or method level and their names as mentioned by [1] and the *MetricsTree* plug-in. Of this we will use the names and abbreviations of the latter, with an exception of methods lines of code (MLOC) for more clarity. The nested block depth threshold is set as a threshold for the maximum of the condition nesting depth and the loop nesting depth.

<i>MetricsTree</i>	Corresponding name from [1]	Class/ Method	Threshold Good	Threshold Common	Threshold Bad
number of methods (NOM)	number of methods	Class	$m \leq 6$	$6 < m \leq 14$	$m > 14$
number of attributes (NOA)	number of fields	Class	$m \leq 3$	$3 < m \leq 8$	$m > 8$
weighted methods per class (WMC)	weighted methods per class	Class	$m \leq 11$	$11 < m \leq 34$	$m > 34$
coupling between objects (CBJ)	afferent coupling	Class	$m \leq 7$	$7 < m \leq 39$	$m > 39$
lines of code (LOC)	method lines of code	Method	$m \leq 10$	$10 < m \leq 30$	$m > 30$
McCabe cyclo-matic complexity (CC)	McCabe cyclo-matic complexity	Method	$m \leq 2$	$2 < m \leq 4$	$m > 4$
condition/loop nesting depth (CND/LND)	nested block depth	Method	$m \leq 1$	$1 < m \leq 3$	$m > 3$

Table 1: Code metrics thresholds.

---

The thresholds sort code into three categories on each metric: Bad, common and good. These thresholds are derived from published open-source object-oriented software. The software showed a heavily tailed distribution on almost all metrics, which is why the thresholds categorize all classes and methods in one of three categories.

It was shown by [1] that bad software had relatively more methods which were categorized in the bad category and relatively fewer methods which were categorized in the good category. For classes, they showed that problematic classes will have multiple metrics outside of the good range. However, the quality of a class could not be determined using only one metric. Lastly they showed that scoring in the bad category for either classes or methods corresponded with code "bad smells".

Conclusively the metrics showed effectiveness in indicating whether a software might have problems and can be used to analyze whether software needs to be looked at for refactoring.

## 2 Refactoring methods

### 2.1 *ActivityService.updateActivity()*

The *updateActivity()* method was chosen to refactor. In the code metrics (fig 1a) a large number of lines of code and McCabe Cyclomatic Complexity can be seen, as well as a somewhat high number for Condition Nesting Depth. This method can be categorized in the bad category for MLOC and CC and the common category for CND (table 1).

The method was refactored by extracting two methods: *createUpdateNotificationRequestModel()* and *removeUnavailableUser()*. This split the lines of code, cyclomatic complexity and the conditions over three methods. This reduces aforementioned metrics for the *updateActivity()* method (fig 1b), while keeping the metrics low for the two extracted methods (fig 2). The *updateActivity()* method now scores good for CND, common for CC and bad for MLOC, with improvement in the MLOC. The two extracted methods score in the common or good category for all metrics. (table 1)

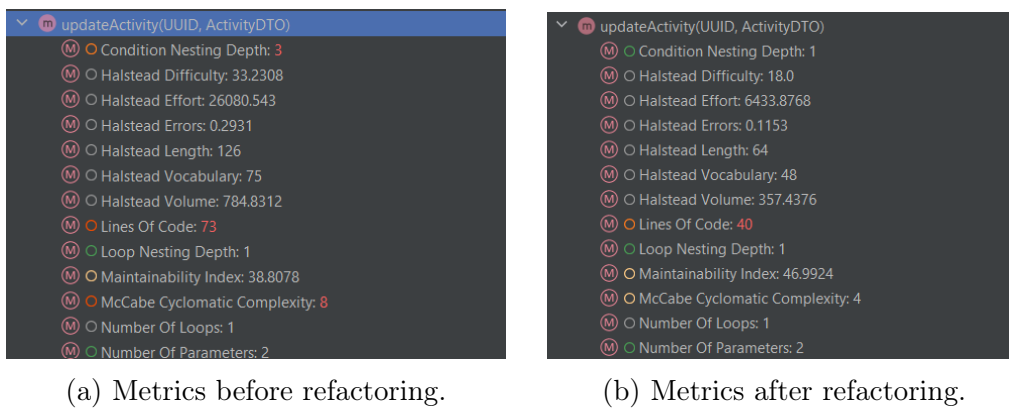


Figure 1: Code metrics of the *updateActivity()* method.

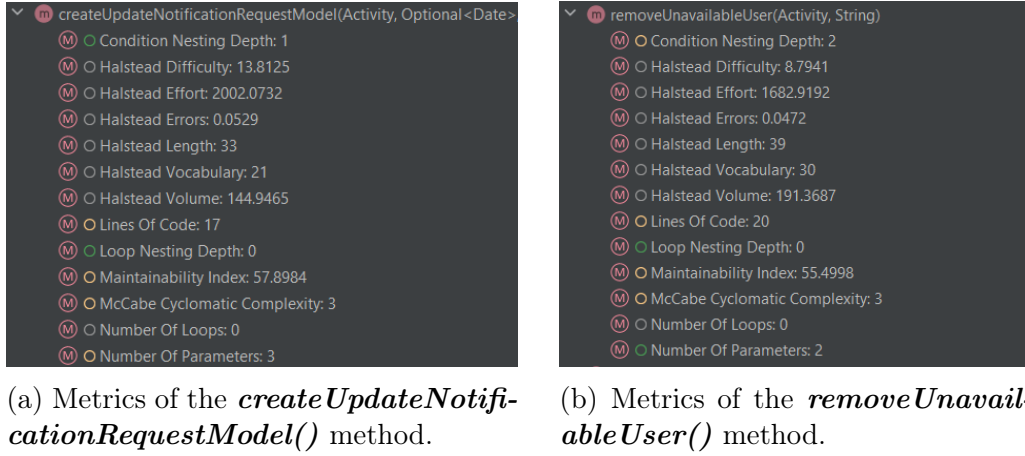


Figure 2: Code metrics of the extracted methods of the *updateActivity()* method.

## 2.2 *ActivityService.getParticipants()*

The *getParticipants()* method was chosen to refactor, due to a high McCabe Cyclomatic Complexity (fig 3a). The method was refactored by re-ordering the code as well as using a stream instead of a for-loop. This reduced the CC from 5 to 2 (fig 3), going from the bad to the good category. Additionally CND was reduced, being placed in the good instead of the common category (table 1).

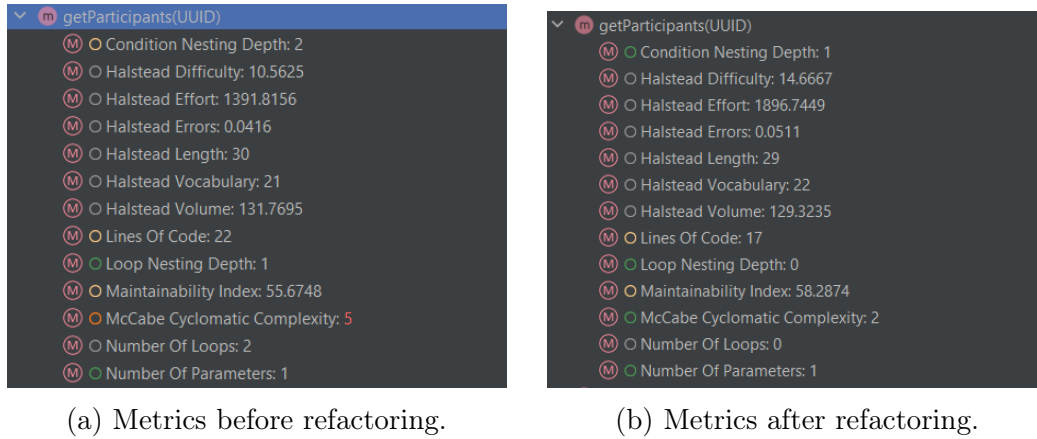


Figure 3: Code metrics of the *getParticipants()* method.

## 2.3 *ActivityService.acceptUser()*

The *acceptUser()* method was chosen to refactor, due to a high Condition Nesting Depth (fig 4a). The method was refactored by changing the code so that we change the structure of the conditions. This reduced the Condition Nesting Depth from 2 to 1, which is the desirable level in this method. (fig 4).

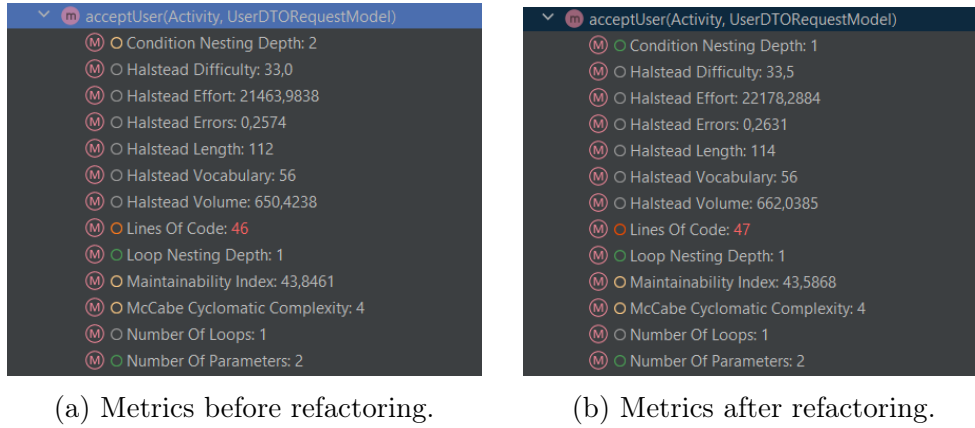


Figure 4: Code metrics of the `acceptUser()` method.

## 2.4 `UserService.updateUser()`

The `updateUser()` method was chosen to refactor since in the code metrics (fig 1a) a somewhat large number of lines of code and McCabe Cyclomatic Complexity can be seen.

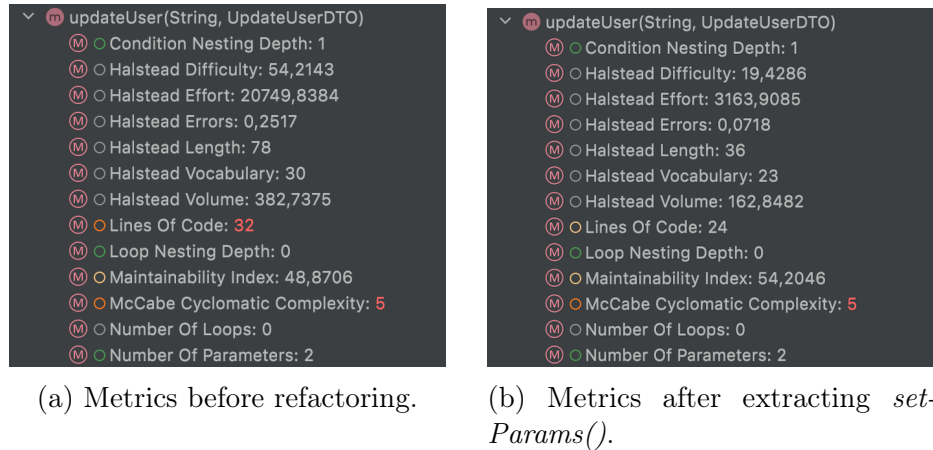
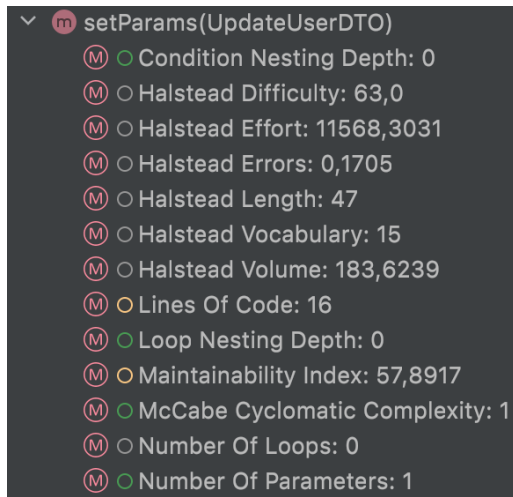
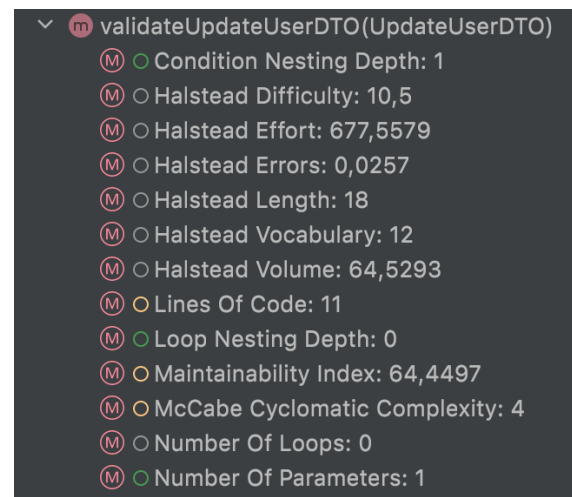


Figure 5: Code metrics of the `updateUser()` method.

The method was refactored by extracting two methods: *validateUpdateUserDTO()* and *setParams()*. The method *setParams()* was created inside of the User class since it logically belongs there. The *validateUpdateUserDTO()* method was created in the same class *UserService* and was made to be a private method only used by the methods already existing in that class. This *setParams()* method reduces the number of lines of code in the original method and combines all parameters' setter operations in one method so it could be scalable easier in the future. The method *validateUpdateUserDTO()* was created to reduce the cyclomatic complexity. This reduces the metrics for the *updateUser()* method (fig 5c) making the lines of code go down to the "common" range and cyclomatic complexity to the "good" range according to table 1. The metrics for newly created methods are low so both methods are maintainable (fig 6).



(a) Metrics of the *setParams()* method.



(b) Metrics of the *validateUpdateUserDTO()* method.

Figure 6: Code metrics of the extracted methods for the *updateUser()* method.

## 2.5 *NotifyUserService.retrieveSubject()*

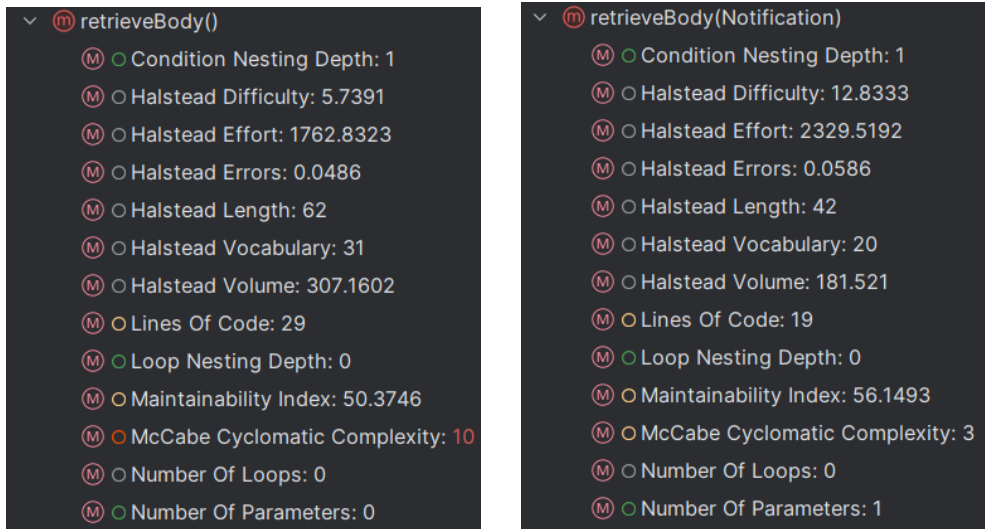
The *retrieveSubject()* method was chosen to refactor since we can see from the code metrics (fig 7a) that it has a somewhat large number of lines of code and a high CC.

The method was refactored by using a map to store the possible values for the subject of the notification instead of storing each value in a variable and using if/else statements to find the right value. As you can see in figure 7 this reduced the cyclomatic complexity of the *retrieveSubject()* method from 4 to 1 (common to good), and raised the maintainability index from 59.4 to 66.2.

## 2.6 *NotifyUserService.retrieveBody()*

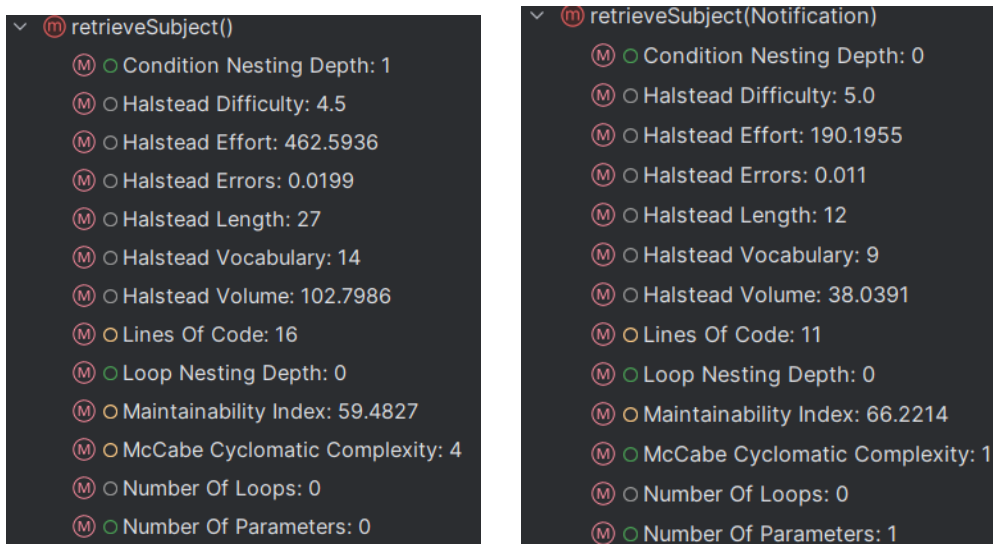
The *retrieveBody()* method was chosen to refactor since in the code metrics (fig 8a) a somewhat large number of lines of code and McCabe Cyclomatic Complexity can be seen.

The method was refactored by using a map to store the possible values for the body of the notification instead of storing each value in a variable and using if/else statements to find the right value. As you can see in the figures(fig 8) below this reduced the cyclomatic complexity of the `retrieveSubject()` method from 10 to 3 (bad to common), and raised the maintainability index from 50.3 to 56.14.



(a) Metrics of the `retrieveBody()` method before. (b) Metrics of the `retrieveBody(Notification)` method after.

Figure 7: Code metrics of the `retrieveBody()` method before and after refactoring.



(a) Metrics of the `retrieveSubject()` method before. (b) Metrics of the `retrieveSubject(Notification)` method after.

Figure 8: Code metrics of the `retrieveSubject()` method before and after refactoring.

---

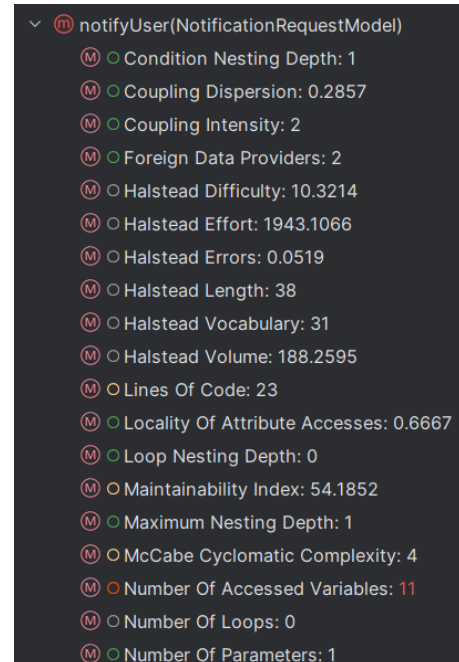
## 2.7 *NotifyUserService.notifyUser()*

The *notifyUser()* method was chosen to refactor since in the code metrics (fig 9a) a somewhat large number of lines of code, McCabe Cyclomatic Complexity, Coupling Intensity, Foreign data providers and Number of Accessed variables can be seen.

The method was refactored by abstracting the functionality of validating the *NotificationRequestModel* and validating the configuration into separate methods(*validateRequest()* and *validateConfiguration()*), alongside moving the functionality of building the request to retrieve the email address and executing the notification using the appropriate strategy to new methods(*buildRequest()* and *executeNotificaition()*). As you can see in the figures(fig 9) below this reduced the cyclomatic complexity of the *notifyUser()* method from 15 to 4 (bad to common), and raised the maintainability index from 42.39 to 54.18.



(a) Metrics of the *notifyUser()* method before.



(b) Metrics of the *notifyUser()* method after.

Figure 9: Code metrics of the *notifyUser()* method before and after refactoring.

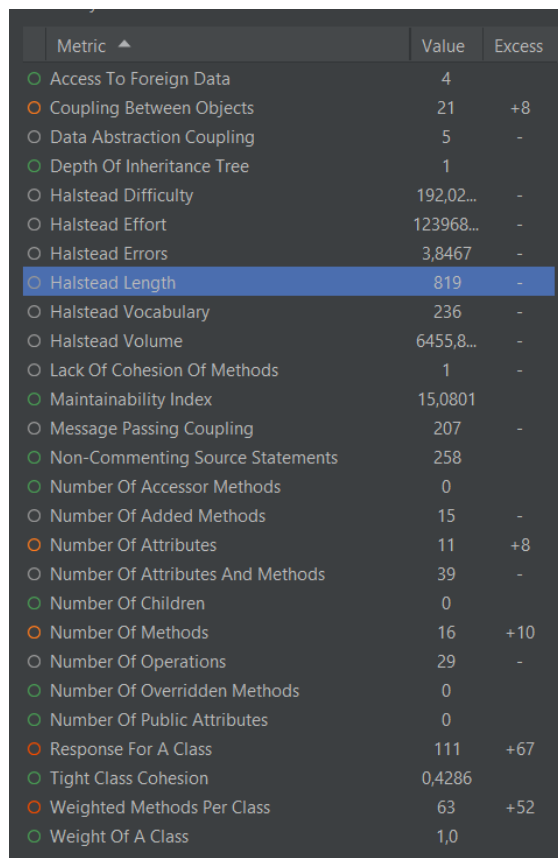


---

## 3 Refactoring classes

### 3.1 *ActivityService* class

The activity service is the class that defines all internal operations that handle the repositories when it comes to activities. This service was originally implemented as one class that handled all functions. We decided to refactor this service because having it split in two smaller services would not only decrease the number of attributes used by each, but it would also improve maintainability and coupling between objects. So in order to refactor the service we decided to split all functionality that handles only internal operations in one service and in the other service we placed any function that sends requests to other microservices. The two services will respectively be named *ActivityService* and *ActivityOutService*. This splitting will improve both the number of attributes, but also class coupling because *ActivityService* will no longer be using the *RestTemplate*, the same way *ActivityOutService* will not be using the *authmanager*. As you can be seen in figures 10 and 11 the coupling between objects goes from 21 to 16 in each, also number of attributes is now within thresholds.



Metric ▲	Value	Excess
Access To Foreign Data	4	
Coupling Between Objects	21	+8
Data Abstraction Coupling	5	-
Depth Of Inheritance Tree	1	
Halstead Difficulty	192,02...	-
Halstead Effort	123968...	-
Halstead Errors	3,8467	-
Halstead Length	819	-
Halstead Vocabulary	236	-
Halstead Volume	6455,8...	-
Lack Of Cohesion Of Methods	1	-
Maintainability Index	15,0801	
Message Passing Coupling	207	-
Non-Commenting Source Statements	258	
Number Of Accessor Methods	0	
Number Of Added Methods	15	-
Number Of Attributes	11	+8
Number Of Attributes And Methods	39	-
Number Of Children	0	
Number Of Methods	16	+10
Number Of Operations	29	-
Number Of Overridden Methods	0	
Number Of Public Attributes	0	
Response For A Class	111	+67
Tight Class Cohesion	0,4286	
Weighted Methods Per Class	63	+52
Weight Of A Class	1,0	

Figure 10: Code metrics of the *ActivityService* method before refactoring.

Metric	Value	Excess
Access To Foreign Data	5	
Coupling Between Objects	16	+3
Data Abstraction Coupling	3	-
Depth Of Inheritance Tree	1	
Halstead Difficulty	70,9024	-
Halstead Effort	109276...	-
Halstead Errors	0,7619	-
Halstead Length	233	-
Halstead Vocabulary	98	-
Halstead Volume	1541,2...	-
Lack Of Cohesion Of Methods	3	-
Maintainability Index	30,1215	+11,12...
Message Passing Coupling	62	-
Non-Commenting Source Statements	81	
Number Of Accessor Methods	0	
Number Of Added Methods	6	-
Number Of Attributes	3	
Number Of Attributes And Methods	22	-
Number Of Children	0	
Number Of Methods	7	+1
Number Of Operations	20	-
Number Of Overridden Methods	0	
Number Of Public Attributes	0	
Response For A Class	51	+7
Tight Class Cohesion	0,4	
Weight Of A Class	1,0	
Weighted Methods Per Class	21	+10

(a) *ActivityService* metrics after refactoring.

Metric	Value	Excess
Access To Foreign Data	1	
Coupling Between Objects	16	+3
Data Abstraction Coupling	3	-
Depth Of Inheritance Tree	1	
Halstead Difficulty	150,27...	-
Halstead Effort	656152...	-
Halstead Errors	2,517	-
Halstead Length	586	-
Halstead Vocabulary	175	-
Halstead Volume	4366,4...	-
Lack Of Cohesion Of Methods	1	-
Maintainability Index	19,9735	+0,9735
Message Passing Coupling	144	-
Non-Commenting Source Statements	178	
Number Of Accessor Methods	0	
Number Of Added Methods	8	-
Number Of Attributes	3	
Number Of Attributes And Methods	24	-
Number Of Children	0	
Number Of Methods	9	+3
Number Of Operations	22	-
Number Of Overridden Methods	0	
Number Of Public Attributes	0	
Response For A Class	80	+36
Tight Class Cohesion	0,6429	
Weight Of A Class	1,0	
Weighted Methods Per Class	42	+31

(b) *ActivityOutService* metrics after refactoring.

Figure 11: Code metrics of the *ActivityService* method after refactoring.

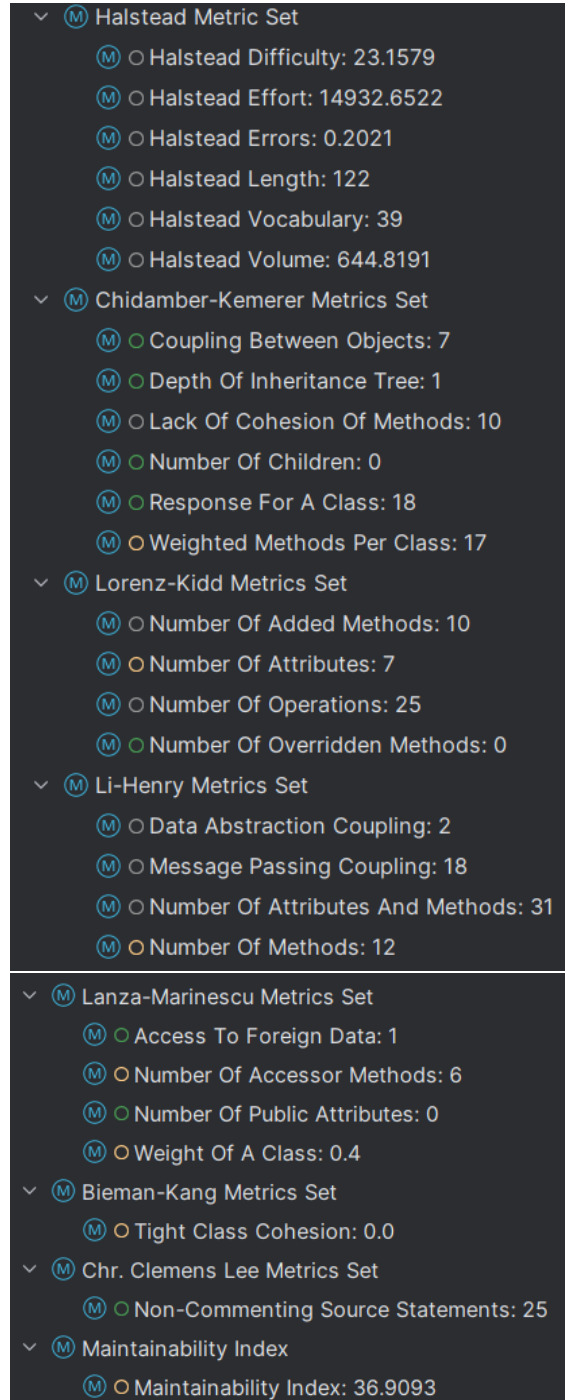
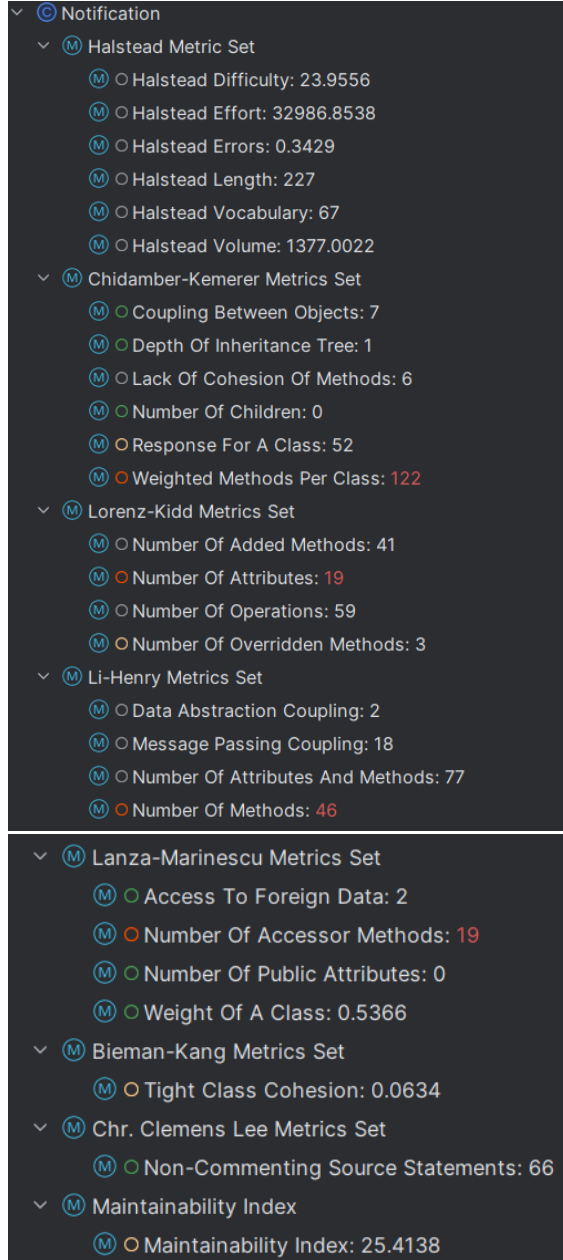
### 3.2 Notification class

The *Notification* class was chosen to refactor due to the high number of attributes, accessor methods, number of weighted method per class and number of methods it had. As you can see in figure 12, these numbers have been drastically improved, alongside the maintainability index of the class.

This was mainly done by removing the attributes used for composing the subject and body of the notification. They were loaded from *application.properties* using the *@Value* annotation in order to avoid hardcoding them. This came with the downside of increasing the number of attributes of the *Notification* class, alongside the number of accessor methods. The *Notification* class also contained the logic for putting together these attributes for creating the subject and body of a notification. Since this class purpose is mainly to pass data around, this logic was moved to the *NotifyUser* class.

The *@Data* annotation was removed from the class and replaced with only the necessary getters and setters.

All this refactoring improved the Maintainability Index from 25.41 to 36.9.



(a) Metrics of the *Notification* class before.

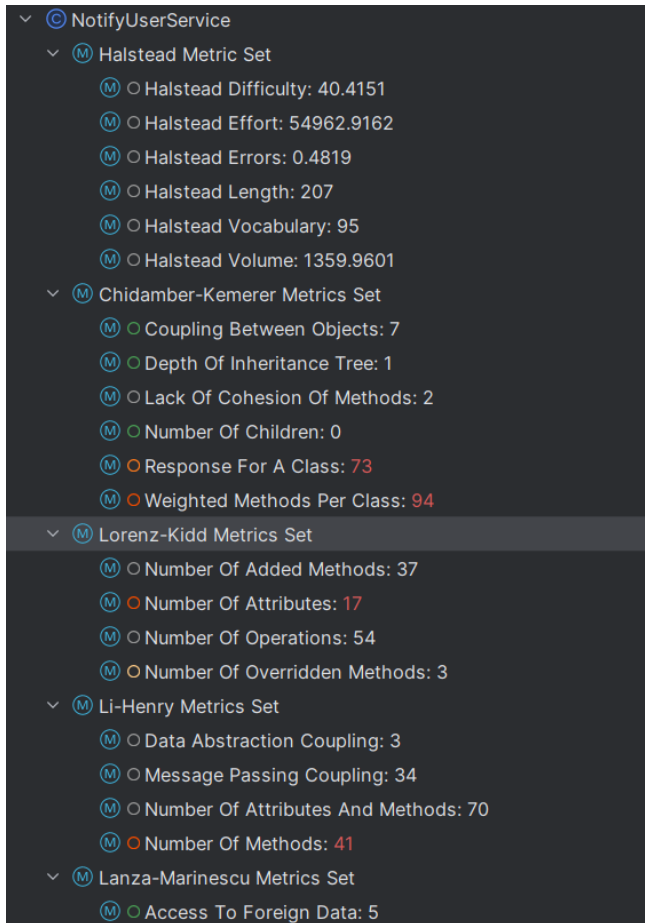
(b) Metrics of the *Notification* class after.

Figure 12: Code metrics of the *Notification* class before and after refactoring.

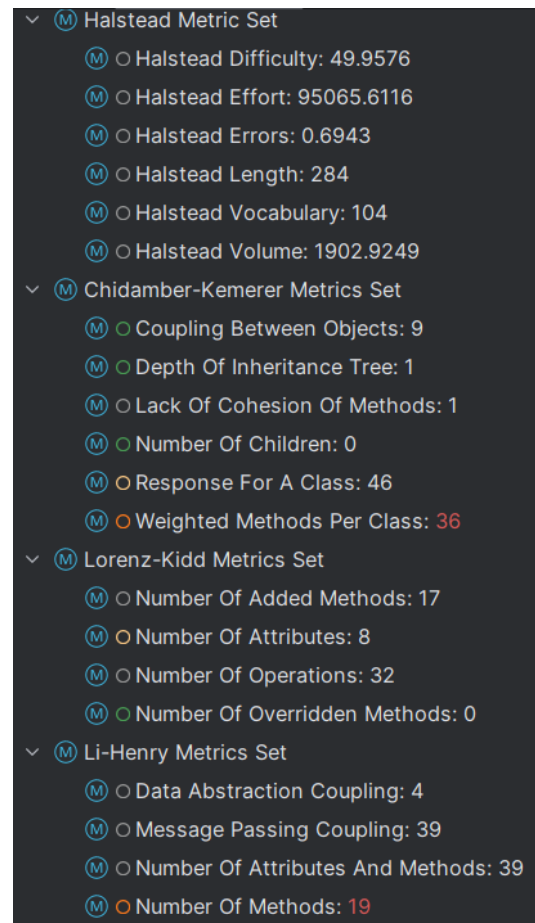
---

### 3.3 NotifyUserService class

The *NotifyUserService* class was chosen to refactor due to the high number of attributes, accessor methods, number of weighted method per class and number of methods it had. This class contained many attributes that were loaded from the `application.properties` file using the `@Value` annotation. While this removed hardcoded values, it lowered the maintainability index. In order to reduce the number of attributes we replaced them with 2 maps, one for the subject and one for the body of the notification being transmitted. These maps are now initialized with 2 `@Autowired` setters which is a better trade-off instead of having many attributes. This also improved the cyclomatic complexity of the `retrieveBody()` and `retrieveSubject()` methods. As you can see in the figures below, the number of attributes of the *NotifyUserService* class lowered alongside the number of accessor methods and the overall number of methods.



(a) Metrics of the *NotifyUserService* class before.



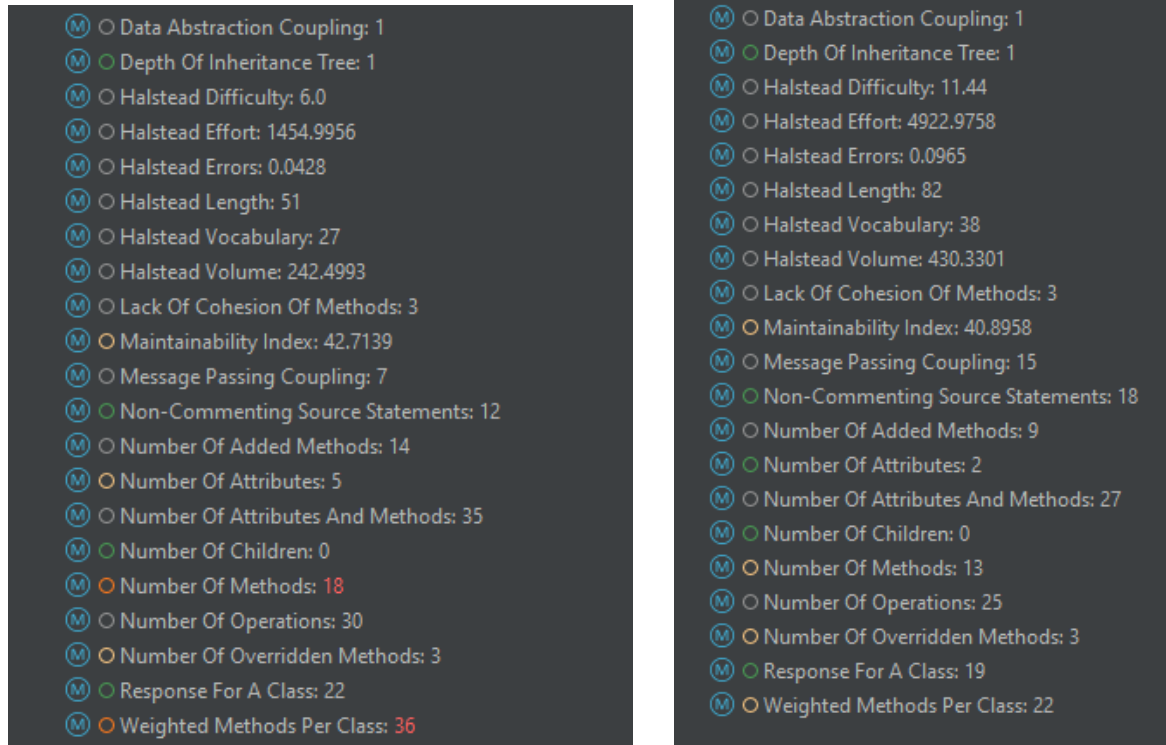
(b) Metrics of the *NotifyUserService* class after.

Figure 13: Code metrics of the *NotifyUserService* class before and after refactoring.

---

### 3.4 KafkaProducerConfig class

The *KafkaProducerConfig* class was chosen to refactor due to the high number of attributes, number of methods and weighted methods per class it had. This class contained many configuration variables for kafka that were annotated with *@Value* and loaded from *application.properties*. Since their value is constant throughout the program, it isn't maintainable and efficient to store them as class attributes and also have accessor methods for them, like setters since that can provide unauthorized opportunities to change the values and thus break the program. However, having them as hardcoded values throughout wasn't really an improvement, decreasing maintainability. Therefore, I decided to create a map that will store all of these values, providing easy and secure access to them while also improving the number of attributes metric. Moreover, due to the removal of the unnecessary getters and setters, the number of methods and weighted methods per class also improved, going from the thresholds of bad to casual.



(a) Metrics before refactoring.

(b) Metrics after refactoring.

Figure 14: Code metrics of the *KafkaProducerConfig* class.

---

## References

- [1] T. G. S. Filó and M. Bigonha, “A catalogue of thresholds for object-oriented software metrics,” 2015.