Lab Assignment 2: Code Refactoring

Software Engineering Methods

Group 30a

Markas Aišparas

Gayeon Jee

Maria Mihai

Jakub Patałuch

Zofia Rogacka-Trojak

Andrei Simionescu

Thresholds

A max value of 20 for the **WMC** and 45 for the **RFC** are considered good thresholds because they provide a balance between code complexity and maintainability.

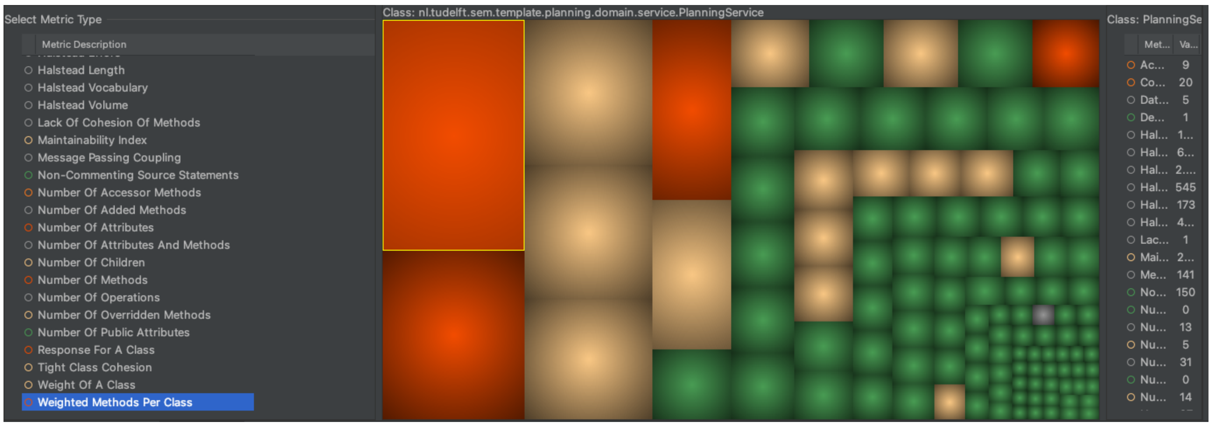
WMC value of 20 suggests that a class is not overly complex, with a moderate number of methods. This makes the code relatively easy to understand and follow, and it allows for a good level of detail and functionality to be included in a single class without becoming too unwieldy. However, for the **WMC** of controllers, we put it as an exception to disregard the **WMC** caused by the try-and-catch statements. We created several custom exceptions to accurately inform the users of the current status of the application. Thus, it is reasonable to keep them as they are.

Secondly, a **RFC** value of 45 suggests that a class is not overly dependent on other classes, with a moderate number of method calls. This makes the code relatively easy to understand and follow, and it reduces the risk of introducing bugs when making changes to the code.

A cyclomatic complexity **(CC)** of 10 suggests that the code is composed of a small number of relatively simple branches, loops or conditionals, making it more straightforward to follow the flow of execution. It allows for a good level of detail and functionality to be included in a single method, while preserving code readability.

The threshold chosen for the appropriate **number of parameters** per method was no more than 3. This ensures code readability, clarity, maintainability and performance as methods with a large number of parameters can negatively impact performance, especially when the method is called frequently. A smaller number of parameters may improve performance by reducing the amount of memory and processing required to execute the method.

.For refactoring the application, we decided to use the Metrics Tree plugin. Initially, we inspected the Project Metrics and the Class Metrics tabs to get an understanding of where the most code smells can be found.



Met...	Va...
Ac...	9
Co...	20
Dat...	5
De...	1
Hal...	1...
Hal...	6...
Hal...	2....
Hal...	545
Hal...	173
Hal...	4...
Lac...	1
Mai...	2...
Me...	141
No...	150
Nu...	0
Nu...	13
Nu...	5
Nu...	31
Nu...	0
Nu...	14

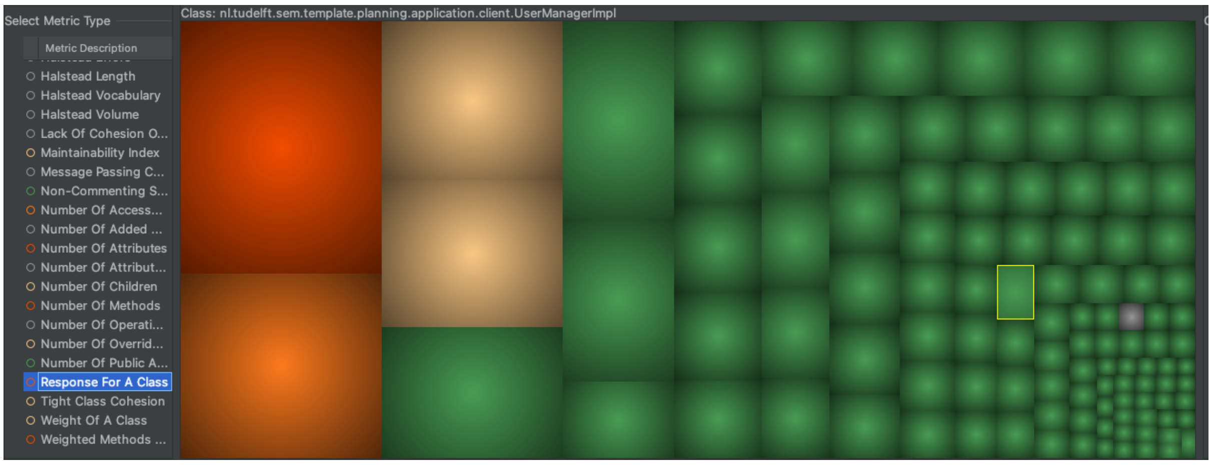


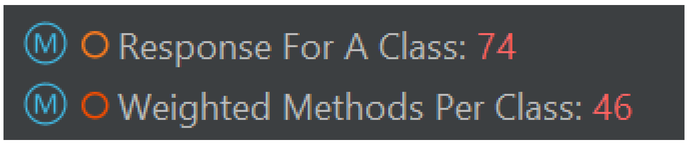
Figure 1 & 2. Metrics Trees before and after the refactoring

We paid special attention to the CK Metrics (WMC = Weighted Methods (Complexity) for Class, CBO = Coupling between objects, RFC = Response For Class, LCOM = Lack of Cohesion of Methods, DIT = Depth of Inheritance Tree, NOC = Number of Children).

In the following sections, we will present the refactorings we undertook for the project, explaining why they were chosen, what the refactoring process was like, as well as its result. In most cases, the values for the thresholds were chosen with reference to MetricsTree. When they differ, a reason is given.

Class Level Refactoring

I. ActivityService in Activity Microservice



Response For A Class: 74
Weighted Methods Per Class: 46

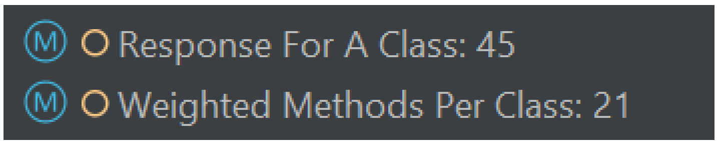
The MetricsTree plugin, a tool used to analyze the structural and design-related characteristics of software, identified several code smells within the class ActivityService.

Specifically, the metrics "Response For A Class" and "Weighted Methods Per Class" indicated that there were issues with the organization and functionality of the class. In our refactoring efforts, the desired threshold for the weighted method count (WMC) was within the standard range of 0-12, and the response for class (RFC) within the range of 0-45. However, due to the complexity of the class, it was not possible to attain these goals. As an alternative, we decided to lower our targets to a WMC of 25 and an RFC of 50, as this still resulted in a significant reduction in the values, and improved the readability and maintainability of the code. This modification to the class resulted in a visible improvement in its overall quality.

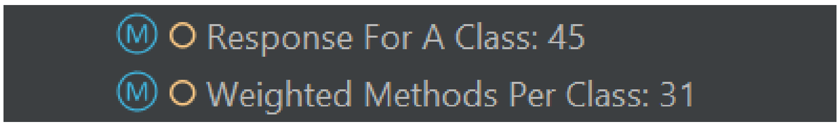
To address these issues, a refactoring process was undertaken. Two methods, filterByRoleLevel and filterByRole, were deemed unnecessary and were removed from the class. This helped to reduce the complexity and improve the cohesion of the class.

To further improve the organization and design of the class, two new classes were introduced: ActivityFilters and ActivityGetters. The method filterByAvailability was moved to the ActivityFilters class, and all of the getter methods were moved to the ActivityGetters class. This served to encapsulate the functionality of the class and make it more modular, improving the overall extensibility of the application.

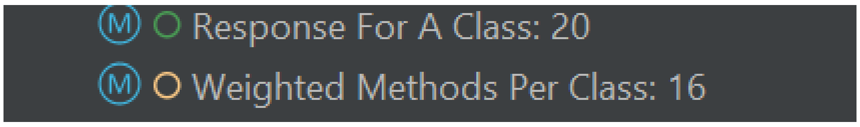
Overall, this refactoring process resulted in an improvement in the metrics scores and a more maintainable and extensible codebase.


Response For A Class: 45
Weighted Methods Per Class: 21

II. ActivityController in Activity Microservice


Response For A Class: 45
Weighted Methods Per Class: 31

The MetricsTree report shows that there are too many instances of response for a class and weighted methods per class. In the activityController class, there are two parser utility methods which use several conditional statements leading to higher cyclomatic complexity. Therefore, the two methods, *parseAvailabilities* and *parseRoles* are separated into a separate method. In addition, the *parseAvailabilities* is refactored in method level to reduce the complexity of the method itself. In addition, there is another helper method *validActivity* which is unused so it is removed.

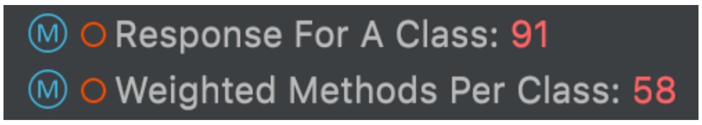

Response For A Class: 20
Weighted Methods Per Class: 16

The above image shows the improved statistics after creating a separate parser utility class. Although there is a remaining problem for the weighted methods per class, this is caused by the try and catch statements. There are different exceptions thrown for various errors. However, these catch statements are kept to accurately inform the current status of the application. Thus, we decided to disregard the try-and-catch statements when resolving this code smell. In the activityController class, there are no conditional statements used that increase the cyclomatic complexity other than the try-and-catch statements. Thus, it is the best range that we can achieve in this class. Although it is still in the yellow range, the number of the code smell decreased by one half so there is improvement coming from the refactoring.

III. PlanningService in Planner Microservice

Metrics before refactoring:

PlanningService.java



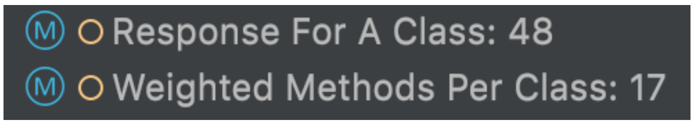
Response For A Class: 91
Weighted Methods Per Class: 58

The MetricsTree report shows that there are too many instances of response for a class and weighted methods per class. In the PlanningService class, there are 4 utility methods related to activities, 3 related to users and 2 related to appointments.

To address these issues, the refactoring process involved the creation of 3 utility classes: ActivityUtils, UserUtils and AppointmentUtils. All 9 utility methods were moved to their corresponding utility classes. Separating these responsibilities into separate classes would encapsulate the PlanningService and benefit the maintainability of the project.

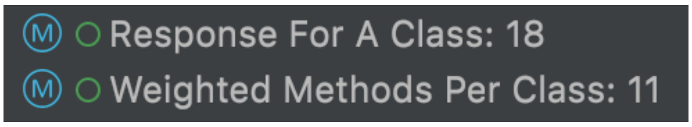
Metrics after refactoring:

PlanningService.java



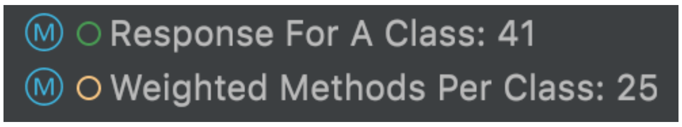
Response For A Class: 48
Weighted Methods Per Class: 17

ActivityUtils.java



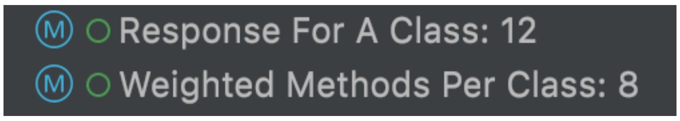
Response For A Class: 18
Weighted Methods Per Class: 11

UserUtils.java



Response For A Class: 41
Weighted Methods Per Class: 25

AppointmentUtils.java

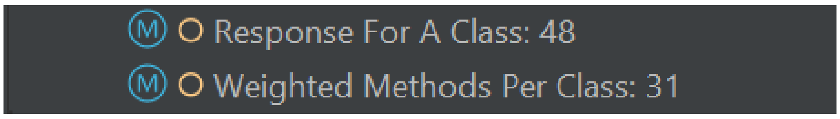


Response For A Class: 12
Weighted Methods Per Class: 8

Although the remaining “response for a class” metric of 48 and “weighted methods per class” of 17 for the PlanningService are still in the yellow range, they were deemed satisfactory as the only methods that remain in the PlanningService are the responsible methods for each route in the controller (with exception of one). Additionally it can also be seen that the metrics “response for a class” and “weighted methods per class” are also satisfactory for the newly introduced classes meaning that the refactoring step did not simply displace the code smell to another class, but removed it entirely from the project.

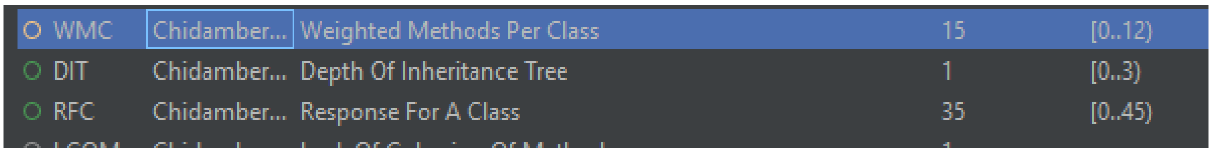
IV. UserService in User Microservice

Metrics before refactoring:



Response For A Class: 48
Weighted Methods Per Class: 31

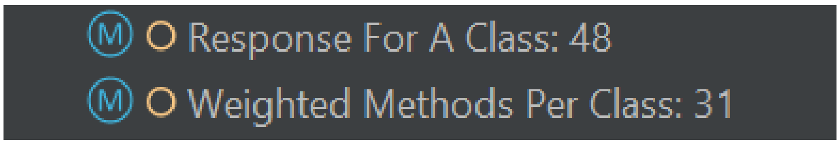
Metrics after refactoring:



WMC	Chidamber...	Weighted Methods Per Class	15	[0..12)
DIT	Chidamber...	Depth Of Inheritance Tree	1	[0..3)
RFC	Chidamber...	Response For A Class	35	[0..45)

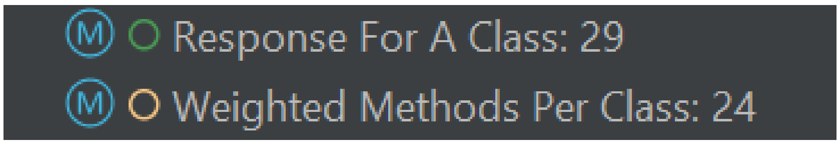
According to the Metrics Tree class metrics, the userService class has code smells due to too many responses for a class and weighted method. The reason for that was separate methods for retrieving each of the attributes of User class separately instead of having one simple function for retrieving User class and then using in-built class getter to receive expected attributes. getUserData function was also redundant as it was returning User class transformed into DTO, so that was changed to *fromUser* static method within DTO. Besides refactoring the issues related to WMC and RFC, we also reduced the Cyclomatic Complexity by removing unnecessary checks for some values, also by moving null checks into lombok @NonNull annotations in the respective entities, what more we made use of Optional elseThrow statement to get rid of unnecessary checks for record existence in the DB. WMC cannot be reduced more.

V. UserController in User Microservice



Response For A Class: 48
Weighted Methods Per Class: 31

According to the Metrics Tree class metrics, the userController class has code smells due to too many responses for a class and weighted method. One of the causes is the usage of checkUserAuthenticated in the AuthManager class which checks if the user has been authenticated and returns a boolean. userController calls this function at the beginning of all the methods and returns an unauthorised response if it returns false. Nevertheless, it is redundant because jwt filter configuration is set to return all unauthorised users for all requests arriving at the microservice. Therefore, there is no need for an additional check on authorization in the controller methods. By deleting this method, there will be less calls to different methods and the cyclomatic complexity of all methods will decrease. Hence, both types of code smells are relieved. Moreover, the checkUserAuthenticated method is removed from the authManager class and related test classes since it is no longer used. The following is the resulting report after refactoring.



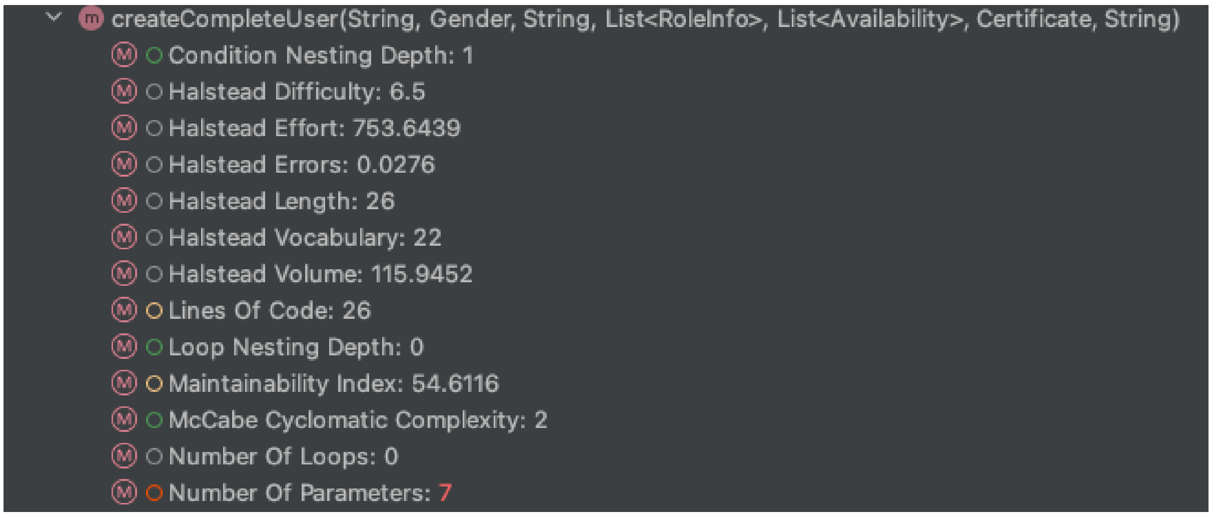
```
Response For A Class: 29
Weighted Methods Per Class: 24
```

Nevertheless, there is still a problem with the weighted methods per class because of the try and catch blocks. There are custom exceptions thrown in the service classes which this controller is encapsulating. As aforementioned in the activity controller section, this is essential to accurately delivering the current status of the application to the users. The userController class uses only two conditional statements other than the try-and-catch statements. Therefore, it is not possible to go down below 24 in this case. There is still an improvement in the code smell since the occurrence of the weighted methods per class decreased from 31 to 24.

Method Level Refactoring

I. UserService in User Microservice

A. createCompleteUser

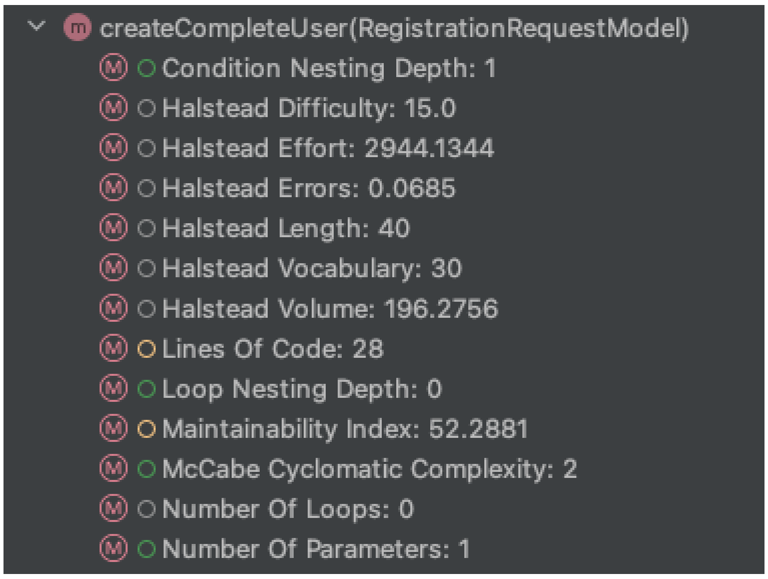


```
createCompleteUser(String, Gender, String, List<RoleInfo>, List<Availability>, Certificate, String)
    Condition Nesting Depth: 1
    Halstead Difficulty: 6.5
    Halstead Effort: 753.6439
    Halstead Errors: 0.0276
    Halstead Length: 26
    Halstead Vocabulary: 22
    Halstead Volume: 115.9452
    Lines Of Code: 26
    Loop Nesting Depth: 0
    Maintainability Index: 54.6116
    McCabe Cyclomatic Complexity: 2
    Number Of Loops: 0
    Number Of Parameters: 7
```

The createCompleteUser method accepts fields referring to user information (such as their name, gender, email) and creates a new user entity, which is saved in the User Database. After successfully creating the user, its ID is returned. After using MetricsTree to compute the metrics for this method, we immediately noticed that the number of parameters metric is outside the range recommended by the plugin: between 0 and 3.

It became obvious after a short analysis that the parameter list could easily be replaced by an already existent object (RegistrationRequestModel). This class already has the exact same fields and is used by the UserService when accepting user input. Furthermore, the createCompleteUser method is only called by the createUser, which also has as parameter a RegistrationRequestModel object.

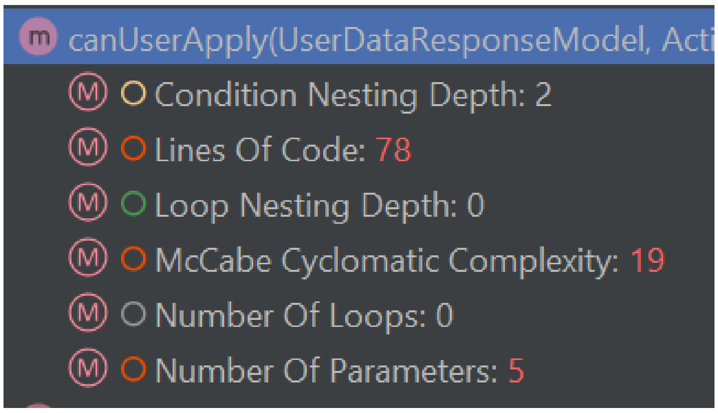
After replacing the parameter list for this method, we ran again the tests and recomputed the metrics. After all of the tests passed and the Number of Parameters metric had a value in the accepted range again, we considered the refactoring process for this method to be finished.



```
createCompleteUser(RegistrationRequestModel)
    Condition Nesting Depth: 1
    Halstead Difficulty: 15.0
    Halstead Effort: 2944.1344
    Halstead Errors: 0.0685
    Halstead Length: 40
    Halstead Vocabulary: 30
    Halstead Volume: 196.2756
    Lines Of Code: 28
    Loop Nesting Depth: 0
    Maintainability Index: 52.2881
    McCabe Cyclomatic Complexity: 2
    Number Of Loops: 0
    Number Of Parameters: 1
```

II. PlanningService in Planner Microservice

A. canUserApply

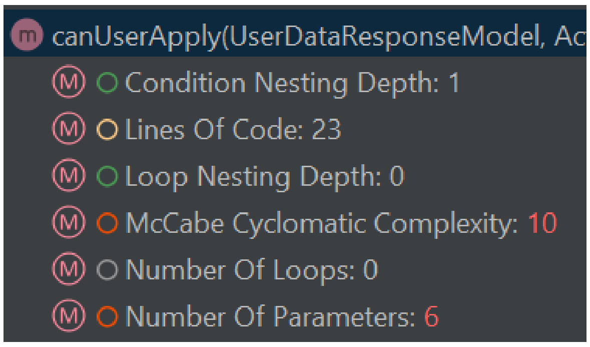


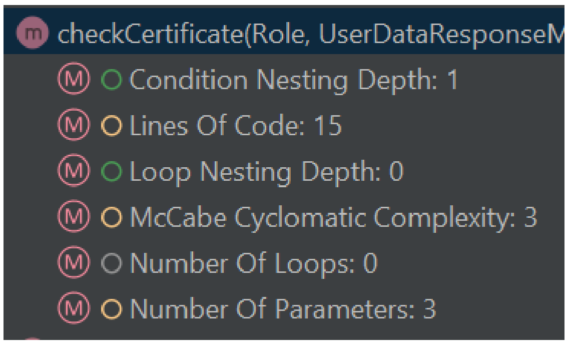
```
canUserApply(UserDataResponseModel, Acti
    Condition Nesting Depth: 2
    Lines Of Code: 78
    Loop Nesting Depth: 0
    McCabe Cyclomatic Complexity: 19
    Number Of Loops: 0
    Number Of Parameters: 5
```

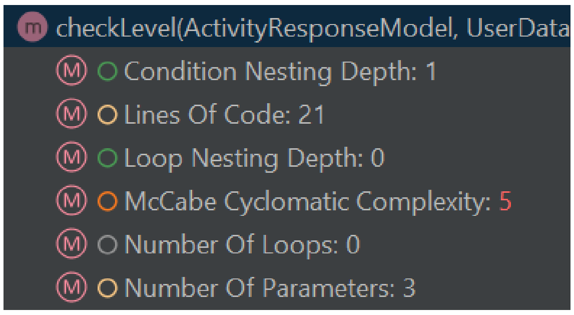
The "canUserApply" method within the PlannerService class was found to have significant issues, particularly with regard to the high values of the "Lines of Code" and "McCabe Cyclomatic Complexity" metrics. Additionally, the method was determined to be difficult to understand and interpret. In our efforts to optimise the codebase, we aimed to achieve a cyclomatic complexity (CC) within the standard range of 0-3 and a lines of code (LOC) within the range of 0-11. However, due to the complexity of the method employed, it was not possible to attain these goals. As an alternative, we decided to lower our targets to a CC of 10 and a LOC of 30, as this resulted in a significant reduction in the values. This modification to the codebase resulted in a visible improvement in its overall quality.

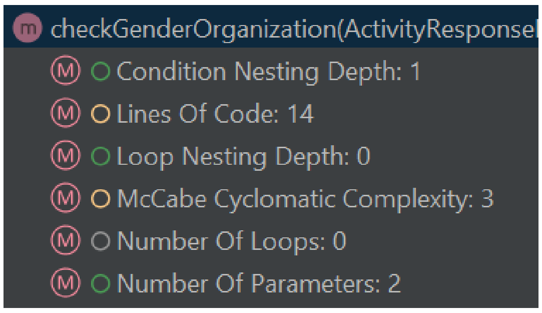
To address these issues, a refactoring of the method was undertaken. The primary objective of the refactoring was to break down the method into smaller, more cohesive and readable segments. To achieve this, a new class, "UserEligibilityForActivity," was introduced. The introduction of this class helped to improve the cohesion and readability of the code, while also reducing the value of the "Weighted Methods per Class" (WMC) metric. The functionalities within the original method were then extracted and converted into separate methods, such as "checkTime," "checkLevel," "checkRole," "checkCertificate," "checkGenderOrganization," and "checkSpotAvailabilities." Furthermore, a significant number of nested conditional statements were merged using logical operators, which improved the "Condition Nesting Depth" metric.

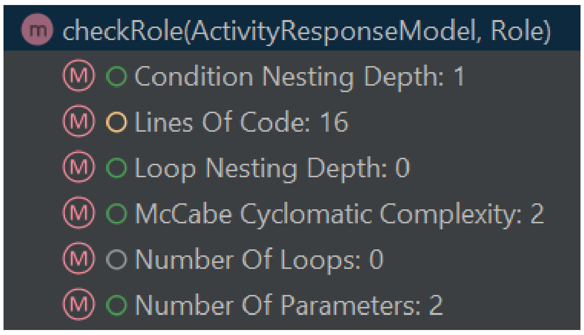
As a result of the refactoring, the "McCabe Cyclomatic Complexity" metric was almost halved. However, it still remained in the red range. Nonetheless, the "Lines of Code" metric was visibly improved.

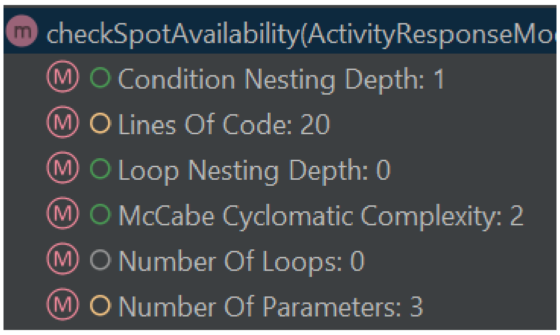

```
canUserApply(UserDataResponseModel, Ac
    Condition Nesting Depth: 1
    Lines Of Code: 23
    Loop Nesting Depth: 0
    McCabe Cyclomatic Complexity: 10
    Number Of Loops: 0
    Number Of Parameters: 6
```

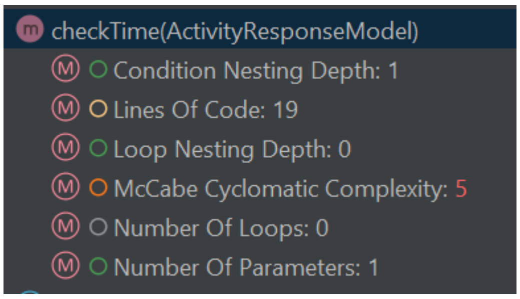

```
checkCertificate(Role, UserDataResponseM
    Condition Nesting Depth: 1
    Lines Of Code: 15
    Loop Nesting Depth: 0
    McCabe Cyclomatic Complexity: 3
    Number Of Loops: 0
    Number Of Parameters: 3
```


```
checkLevel(ActivityResponseModel, UserData
    Condition Nesting Depth: 1
    Lines Of Code: 21
    Loop Nesting Depth: 0
    McCabe Cyclomatic Complexity: 5
    Number Of Loops: 0
    Number Of Parameters: 3
```


```
checkGenderOrganization(ActivityResponse
    Condition Nesting Depth: 1
    Lines Of Code: 14
    Loop Nesting Depth: 0
    McCabe Cyclomatic Complexity: 3
    Number Of Loops: 0
    Number Of Parameters: 2
```

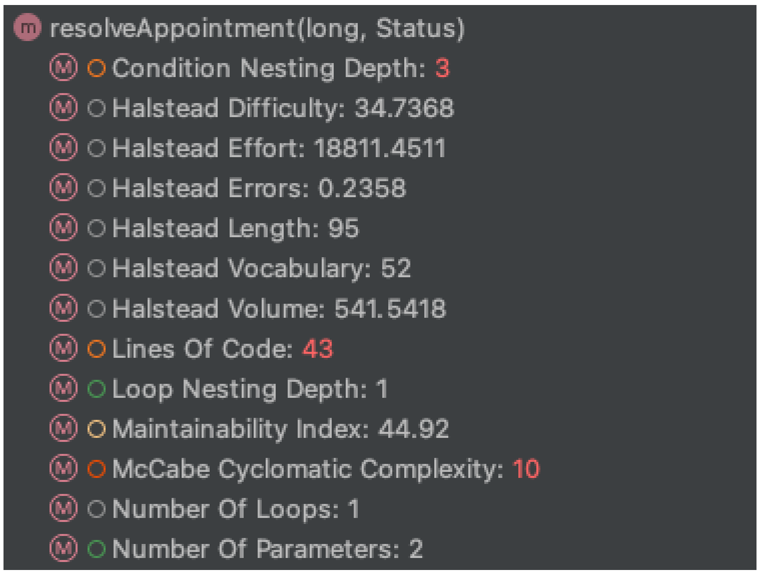

```
checkRole(ActivityResponseModel, Role)
    Condition Nesting Depth: 1
    Lines Of Code: 16
    Loop Nesting Depth: 0
    McCabe Cyclomatic Complexity: 2
    Number Of Loops: 0
    Number Of Parameters: 2
```


```
checkSpotAvailability(ActivityResponseMo
    Condition Nesting Depth: 1
    Lines Of Code: 20
    Loop Nesting Depth: 0
    McCabe Cyclomatic Complexity: 2
    Number Of Loops: 0
    Number Of Parameters: 3
```



```
checkTime(ActivityResponseModel)
    Condition Nesting Depth: 1
    Lines Of Code: 19
    Loop Nesting Depth: 0
    McCabe Cyclomatic Complexity: 5
    Number Of Loops: 0
    Number Of Parameters: 1
```

B. resolveAppointment



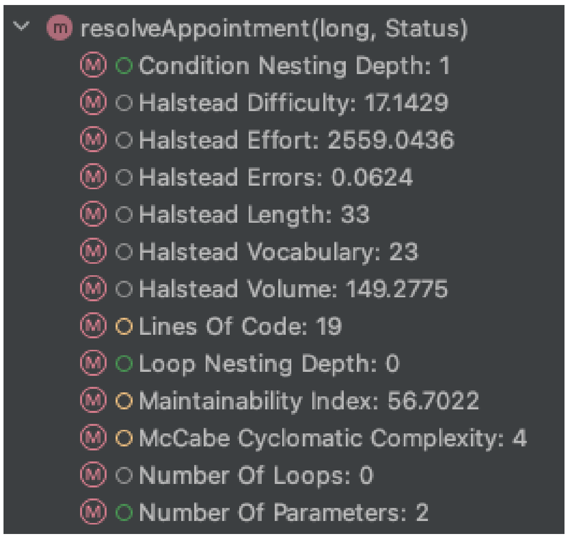
```
resolveAppointment(long, Status)
    Condition Nesting Depth: 3
    Halstead Difficulty: 34.7368
    Halstead Effort: 18811.4511
    Halstead Errors: 0.2358
    Halstead Length: 95
    Halstead Vocabulary: 52
    Halstead Volume: 541.5418
    Lines Of Code: 43
    Loop Nesting Depth: 1
    Maintainability Index: 44.92
    McCabe Cyclomatic Complexity: 10
    Number Of Loops: 1
    Number Of Parameters: 2
```

The resolveAppointment method, belonging to the PlanningService class in the Planner microservice, changes the status of an appointment. It takes as parameters the ID of the applicant and a status (which can be either accepted, denied or pending). Initially, the resolveAppointment method had the following steps:

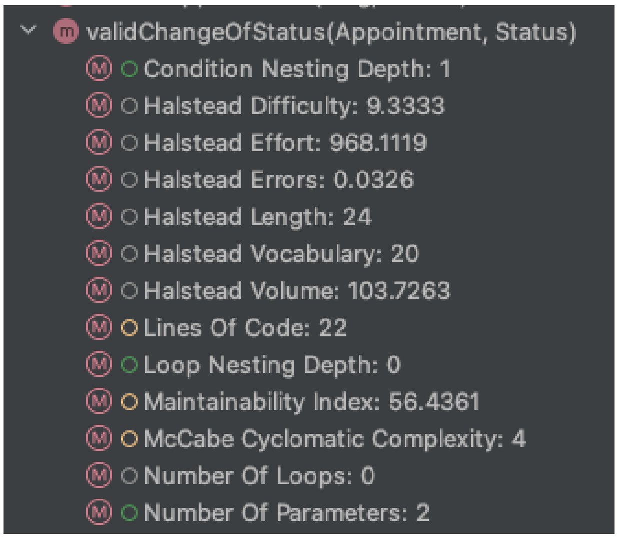
1. Checking whether the status change is valid (appointments which have already been accepted or denied cannot be changed anymore, and changing the status of an appointment to "PENDING" is also invalid);
2. Checking whether the requesting user is allowed to change the status of the appointment;
3. For accepting appointments, checking whether the applicant is still qualified and available (for the cases in which they changed their roles / availabilities after applying);
4. Changing the status of the appointment and saving the change in the database;
5. If the appointment was accepted, preventing the user from applying to any overlapping activities.

By reviewing the responsibilities of this method and the Cyclomatic Complexity and Lines of Code metrics, it was decided that Extract Method refactoring was necessary. We decided that our goal was to bring the value of the Cyclomatic Complexity strictly below 5.
We began this process by analysing the method and deciding which responsibilities could be isolated. Step 5) was identified because it is quite complex and independent from the rest of

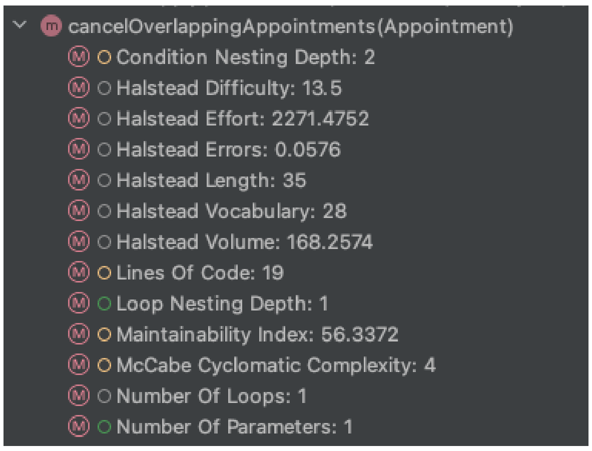
the method’s logic. Steps 1) and 2) both contain validity checks on the status and appointment, and can both be extracted into a method. Additionally, step 3) also contains a validity check which can be done outside of the method itself.
After deciding on the fragments to extract, we created three methods: validChangeOfStatus (for steps 1 and 2), appointment can be accepted (for step 3), and cancelOverlappingAppointments (for step 5). We then moved the code corresponding to each responsibility in the respective method, passed the needed variables as parameters and included method calls in resolveAppointment. Finally, we ran the tests for PlanningService to make sure everything works again. When all the tests passed, and the CC metric was under the chosen metric for all the involved methods, we concluded that the refactoring process for resolveAppointment is finished - for now.



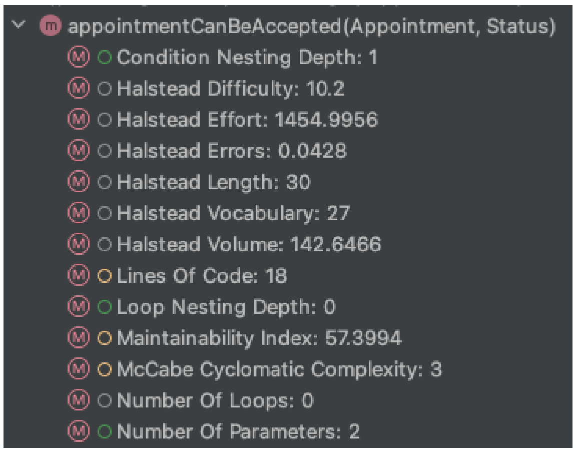
```
resolveAppointment(long, Status)
    Condition Nesting Depth: 1
    Halstead Difficulty: 17.1429
    Halstead Effort: 2559.0436
    Halstead Errors: 0.0624
    Halstead Length: 33
    Halstead Vocabulary: 23
    Halstead Volume: 149.2775
    Lines Of Code: 19
    Loop Nesting Depth: 0
    Maintainability Index: 56.7022
    McCabe Cyclomatic Complexity: 4
    Number Of Loops: 0
    Number Of Parameters: 2
```



```
validChangeOfStatus(Appointment, Status)
    Condition Nesting Depth: 1
    Halstead Difficulty: 9.3333
    Halstead Effort: 968.1119
    Halstead Errors: 0.0326
    Halstead Length: 24
    Halstead Vocabulary: 20
    Halstead Volume: 103.7263
    Lines Of Code: 22
    Loop Nesting Depth: 0
    Maintainability Index: 56.4361
    McCabe Cyclomatic Complexity: 4
    Number Of Loops: 0
    Number Of Parameters: 2
```



```
cancelOverlappingAppointments(Appointment)
    Condition Nesting Depth: 2
    Halstead Difficulty: 13.5
    Halstead Effort: 2271.4752
    Halstead Errors: 0.0576
    Halstead Length: 35
    Halstead Vocabulary: 28
    Halstead Volume: 168.2574
    Lines Of Code: 19
    Loop Nesting Depth: 1
    Maintainability Index: 56.3372
    McCabe Cyclomatic Complexity: 4
    Number Of Loops: 1
    Number Of Parameters: 1
```



```
appointmentCanBeAccepted(Appointment, Status)
    Condition Nesting Depth: 1
    Halstead Difficulty: 10.2
    Halstead Effort: 1454.9956
    Halstead Errors: 0.0428
    Halstead Length: 30
    Halstead Vocabulary: 27
    Halstead Volume: 142.6466
    Lines Of Code: 18
    Loop Nesting Depth: 0
    Maintainability Index: 57.3994
    McCabe Cyclomatic Complexity: 3
    Number Of Loops: 0
    Number Of Parameters: 2
```

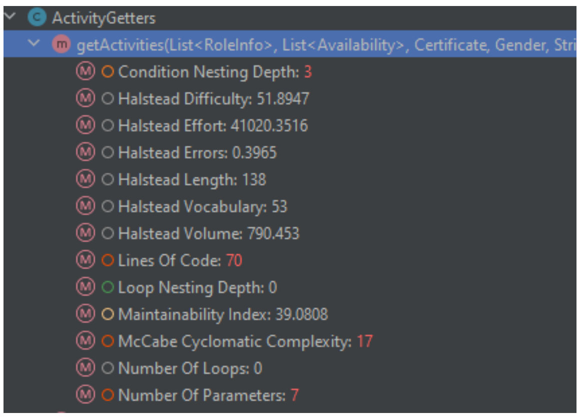
III. ActivityService in Activity Microservice

A. getActivities

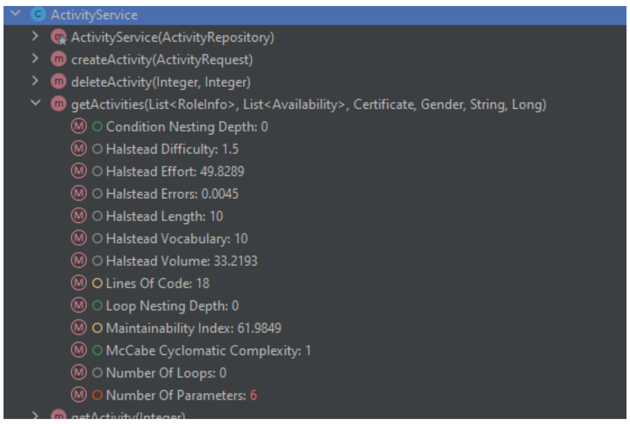
The getActivities method inside of the ActivityService class is the method that handles all the necessary filtering to retrieve the eligible activities for the user. As it was previously implemented it raised 4 major red flags: it had a large number of parameters, it was heavily convoluted and difficult to follow (long method), it had a very high cyclomatic complexity, and all the operations were entirely performed in memory, using Java’s streams API. This was more than enough to consider this method for refactoring. In the refactoring process performed on its class, we also decided to extract this method and move it to a separate class: ActivityGetters (when the pictures were taken a refactoring already took place, moving the getActivities method into the ActivityGetters class, although the included pictures showcase the same problematic metrics, but now in a different class)

We began by fixing the “Long Parameter List” metric, and to do so we reused one of our already existing DTOs which perfectly mapped all of our data to it. After that, we extracted most of the filtering functionality and implemented it separately in the activityRepository class, using Spring’s built-in JPQL Query annotations. We translated all the filtering that was happening on the streams in native SQL, and thus reduced the cyclomatic complexity and the length of the method itself significantly.

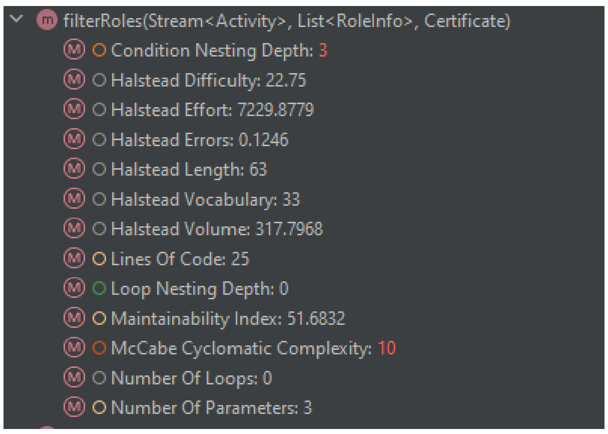
Moreover, for the role filtering, which would have been a bit too convoluted to do in SQL, we extracted that functionality in a separate method and improved the cyclomatic complexity there as well. By doing this refactoring, we reduced the lines of codes per method, improved on the cyclomatic complexity, and separated the logic of the system even better. As a side effect, this refactoring also led to fixing potential issues in the future regarding memory consumption and efficiency.



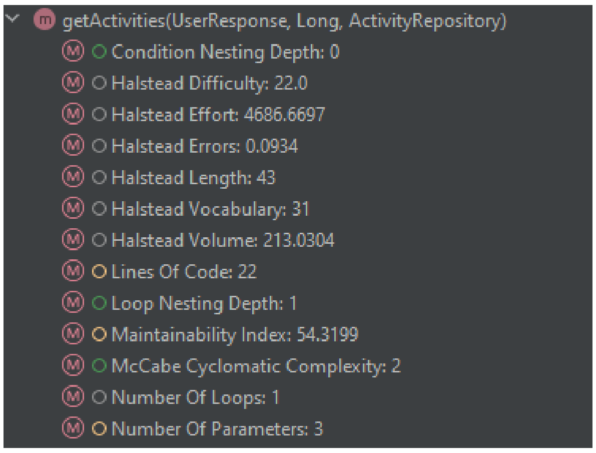
```
ActivityGetters
  getActivities(List<RoleInfo>, List<Availability>, Certificate, Gender, Stri
    Condition Nesting Depth: 3
    Halstead Difficulty: 51.8947
    Halstead Effort: 41020.3516
    Halstead Errors: 0.3965
    Halstead Length: 138
    Halstead Vocabulary: 53
    Halstead Volume: 790.453
    Lines Of Code: 70
    Loop Nesting Depth: 0
    Maintainability Index: 39.0808
    McCabe Cyclomatic Complexity: 17
    Number Of Loops: 0
    Number Of Parameters: 7
```



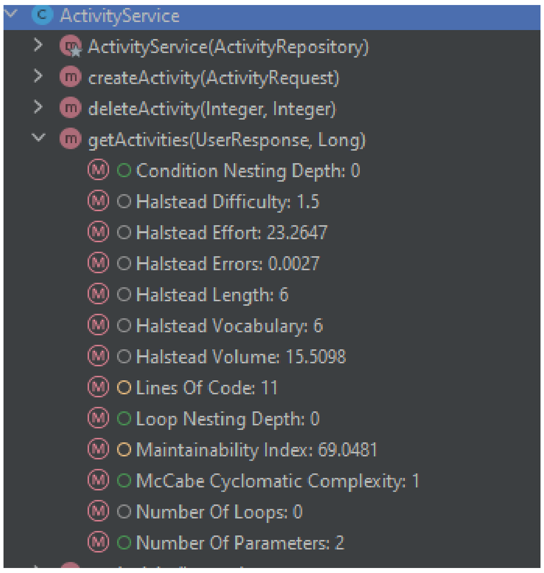
```
ActivityService
  ActivityService(ActivityRepository)
  createActivity(ActivityRequest)
  deleteActivity(Integer, Integer)
  getActivities(List<RoleInfo>, List<Availability>, Certificate, Gender, String, Long)
    Condition Nesting Depth: 0
    Halstead Difficulty: 1.5
    Halstead Effort: 49.8289
    Halstead Errors: 0.0045
    Halstead Length: 10
    Halstead Vocabulary: 10
    Halstead Volume: 33.2193
    Lines Of Code: 18
    Loop Nesting Depth: 0
    Maintainability Index: 61.9849
    McCabe Cyclomatic Complexity: 1
    Number Of Loops: 0
    Number Of Parameters: 6
  getActivity(Integer)
```



```
filterRoles(Stream<Activity>, List<RoleInfo>, Certificate)
  Condition Nesting Depth: 3
  Halstead Difficulty: 22.75
  Halstead Effort: 7229.8779
  Halstead Errors: 0.1246
  Halstead Length: 63
  Halstead Vocabulary: 33
  Halstead Volume: 317.7968
  Lines Of Code: 25
  Loop Nesting Depth: 0
  Maintainability Index: 51.6832
  McCabe Cyclomatic Complexity: 10
  Number Of Loops: 0
  Number Of Parameters: 3
```



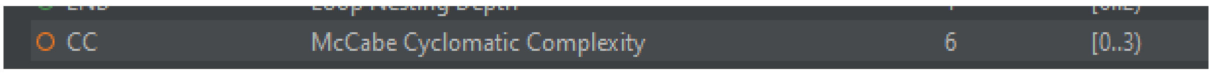
```
getActivities(UserResponse, Long, ActivityRepository)
  Condition Nesting Depth: 0
  Halstead Difficulty: 22.0
  Halstead Effort: 4686.6697
  Halstead Errors: 0.0934
  Halstead Length: 43
  Halstead Vocabulary: 31
  Halstead Volume: 213.0304
  Lines Of Code: 22
  Loop Nesting Depth: 1
  Maintainability Index: 54.3199
  McCabe Cyclomatic Complexity: 2
  Number Of Loops: 1
  Number Of Parameters: 3
```



```
ActivityService
  > ActivityService(ActivityRepository)
  > createActivity(ActivityRequest)
  > deleteActivity(Integer, Integer)
  v getActivities(UserResponse, Long)
      Condition Nesting Depth: 0
      Halstead Difficulty: 1.5
      Halstead Effort: 23.2647
      Halstead Errors: 0.0027
      Halstead Length: 6
      Halstead Vocabulary: 6
      Halstead Volume: 15.5098
      Lines Of Code: 11
      Loop Nesting Depth: 0
      Maintainability Index: 69.0481
      McCabe Cyclomatic Complexity: 1
      Number Of Loops: 0
      Number Of Parameters: 2
```

IV. ActivityController in Activity Microservice

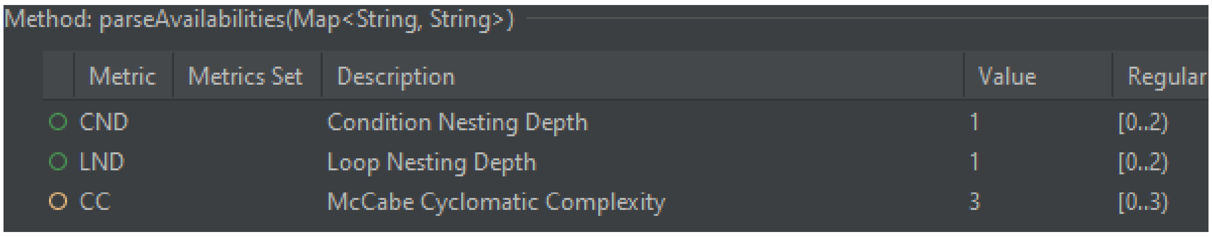
A. parseAvailabilities



CC	McCabe Cyclomatic Complexity	6	[0..3)

The parseAvailabilities method inside of the ActivityController class is the method that as the name suggests parses all the availabilities of given users and provides them to the calling method. We were tasked with reducing the CC of this method. In order to achieve this we used functional programming to do pre-filtering outside the main loop and moved logic of assigning the attribute to the newly created *timeSetter* method within the Availability class. All of these changes are attributed to reducing CC from 6 to 3, which is satisfactory and in-fact we cannot reduce it even more.

Thresholds are depicted below in the *regular* column



Method: parseAvailabilities(Map<String, String>)

Metric	Metrics Set	Description	Value	Regular
CND		Condition Nesting Depth	1	[0..2)
LND		Loop Nesting Depth	1	[0..2)
CC		McCabe Cyclomatic Complexity	3	[0..3)