Software Engineering Methods

CSE2115

Assignment 2

Source: https://www.pexels.com/nl-nl/foto/man-mensen-vrouw-bureau-7605981/

Group 29b

Group members:
Natanael Djajadi
Elena Mihalache
Jordan Sassoon
Paul Tervoort
Yunhan Wang
Jocelyn Woods

Task 1: Identify & motivate for thresholds

1.1 Tool used

Our team chose to use CodeMR to find possible improvements throughout the refactoring process. We found that it has a reputation for helping companies develop better code and better quality products through the use of visuals. CodeMR offers many different visual and non-visual options including pie charts to see the distribution of errors, code statistics to better understand the extent of issues, and CodeMR Metrics to picture the metric value of each type of issue on a scale from 0 to 1. Additionally, this tool is easily integrated into our chosen IDE (IntelliJ) used to generate reports.

1.2 Code metrics

1.2.1 ***Number of Methods Called*** *(NMC, #MC)*

Related Quality Attributes: Size

NMC has to do with exactly what the name states: The number of methods called in a single class. If a method is too complex it becomes hard to understand what it is doing and how to debug it if problems arise. Therefore, it is often better to break larger methods into smaller less complex methods that can be read easier and fixed/implemented in a more direct way. Fixing this will overall improve the project's code quality.

1.2.2 ***Lack of Cohesion of Methods*** *(LCOM)*

Related Quality Attributes: Cohesion

LCOM is a measure of to what extent methods in a class are related to each other. Low Cohesion in a class generally means that said class has been allocated multiple responsibilities. This is a problem because a change in any one of the methods involved in each responsibility will result in a change to the entire class. *LCOM* also can also have an impact on the clarity of the code, but this can often be resolved by dividing one class into multiple subclasses.

1.2.3 ***Weighted Method Count*** *(WMC)*

Related Quality Attributes: Complexity, Size

WMC is related to the number of methods where each method is equal to 1 point. A high count likely means that the class may be domain-specific and therefore less reusable. This count can be used to predict development and the amount of effort needed to maintain and test the code. Even in regards to inheritance if a parent class has a high amount of points then this can have an effect on the classes and methods in the child classes. Having high counts can lead to problems because these classes often are more prone to changes and defects.

1.2.4 *Lack of Cohesion Among Methods* *(LCAM)*

Related Quality Attributes: Cohesion

LCAM measures cohesion based on the parameter types of methods. It is calculated using the equation *LCAM* = 1 - *LCOM*. *LCOM* is explained in section 1.2.2 and is a measure of the extent to which methods are related to one another.

1.2.5 *Lack of Tight Class Cohesion* *(LTCC)*

Related Quality Attributes: Cohesion

LTCC metric is a measurement of the lacking cohesion between only the public methods of a class. This is in reference to the relative number of public methods directly connected in the class. If a class were to have a high lack of cohesion it would likely indicate errors in the design.

1.2.6 *McCabe's Cyclomatic Complexity* *(MCC)*

Related Quality Attributes: Complexity

MCC or *McCabe's metric* represents the number of independent linear paths in the code. It determines the stability and level of confidence in a program. A lower *MCC* indicates easier to understand and less risky to modify code. In his presentation 'Software Quality Metrics to Identify Risk' for the Department of Homeland Security, Tom McCabe introduced the following categorisation to interpret cyclomatic complexity:

- 1 - 10 Simple procedure, little risk
- 11 - 20 More complex, moderate risk
- 21 - 50 Complex, high risk
- > 50 Untestable code, very high risk

1.3 Thresholds Chosen

We chose to use the default settings given by CodeMR as our dividers. The system conveniently divides the levels of problems into five categories according to severity, and colour codes them from green (low) to red (high). Looking over the results from CodeMR we found that all of our classes were at worst coded yellow, so we decided to set our threshold between dark green (low) and light green (medium-low). This means that we are going to try to get all methods and classes we chose to refactor to either light or dark green through the metrics. However, for *Number of Methods Called* and *McCabe Cyclomatic Complexity*, it was decided to set our own defined thresholds that make more sense to our group. The threshold for *NMC* is less than or equal to 8 since we found that if the number of methods called in a method is above 8, then this method can be hard to understand, as we have to look up these methods called in our system and understand what they do. But if it is less than or equal to 8, then the method can still be considered easily understandable. The threshold for *MCC* is also less than or equal to 8. We found if the *MCC* of a method is above 8, then this method can become difficult to understand and debug since it contains too many independent linear paths. Thus, we set the threshold for *MCC* to become <= 8.

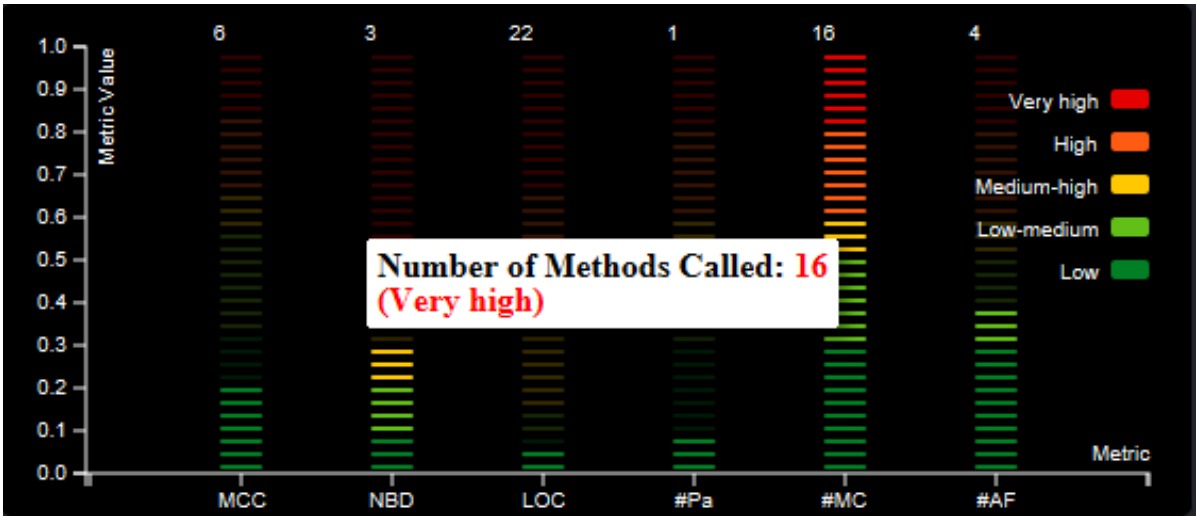
Task 2: Improve code quality

2.1 Method-level code refactoring

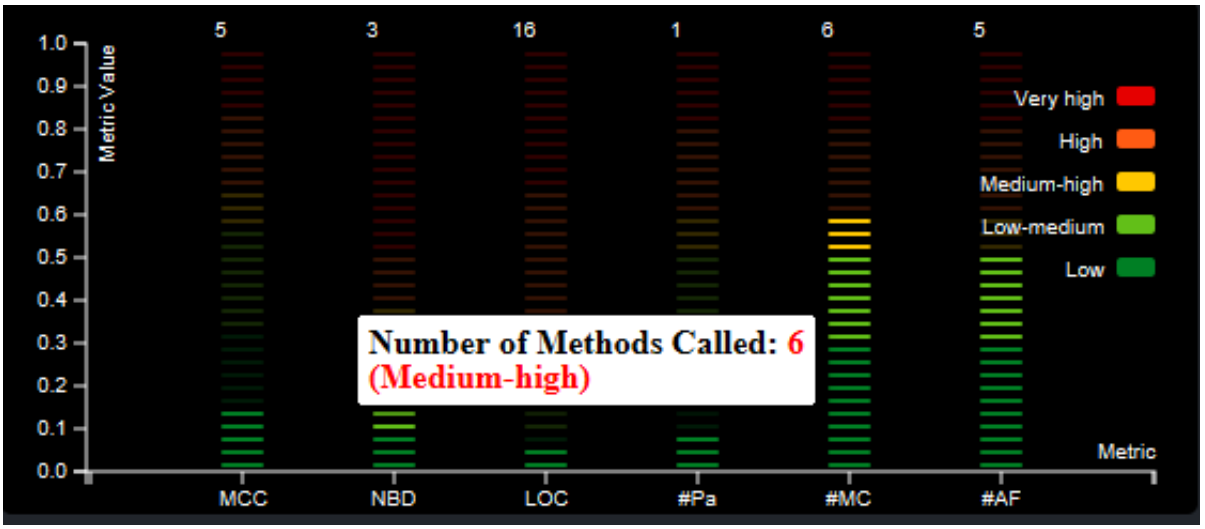
2.1.1) *login() in AuthController class*

File path:
authenticationservice/src/main/java/nl/tudelft/sem/framework/AuthController.java
Commit:
ce5ffe54

The metrics before refactoring:



The metrics after refactoring:



Problem: The *login()* method in the authentication service is responsible for much of the credential verification. For this reason, it is easy to add many data (username and password) checks inside the method's logic without realising the overall complexity goes up and

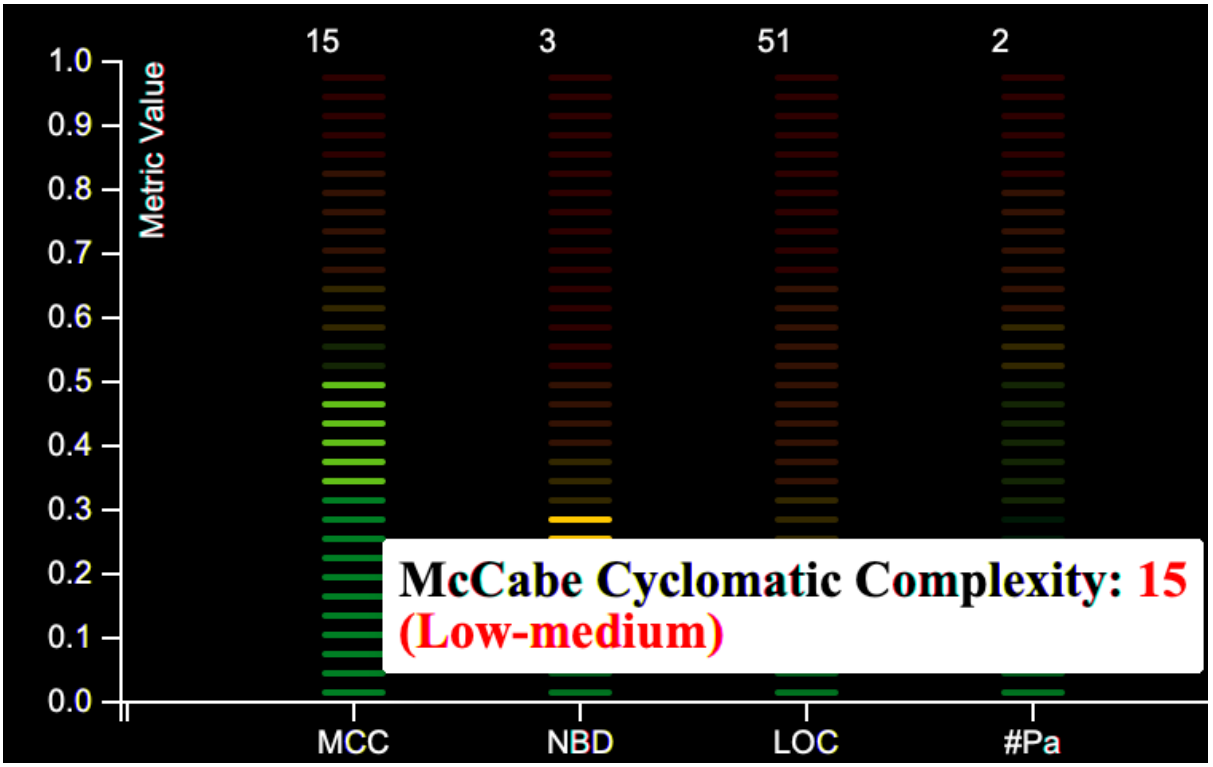
cleanliness goes down. The *Number of Methods Called (NMC)* reached a soaring 16, way above the threshold of 8. This was also due to code duplication in the responses: many failed responses were similar, only their body was different.

Operations applied: We used *Extract Method Refactoring* to split the credential checks into smaller, private methods. Methods *checkCredentials()* and *checkToken()* were introduced to delegate the overkill logic of the *login()* method. These two methods check the equality of usernames and passwords and the correct creation of session tokens. This extraction initially reduced the method's *NMC* metric to 8. We decided to further refactor the method and the class by introducing class level *ResponseEntity.BodyBuilders*, which are used to substitute all of the failed and successful request responses. In this way the response entity is the same, though the body changes if needed. The full *ResponseEntity* called *incorrectUsernameOrPassword* remained as a class-level variable since it was used very often. With all of this refactoring the *NMC* metric was reduced to 6, and the surrounding methods also were improved: *createAccount()* went from 11 to 7 (Very High to High).

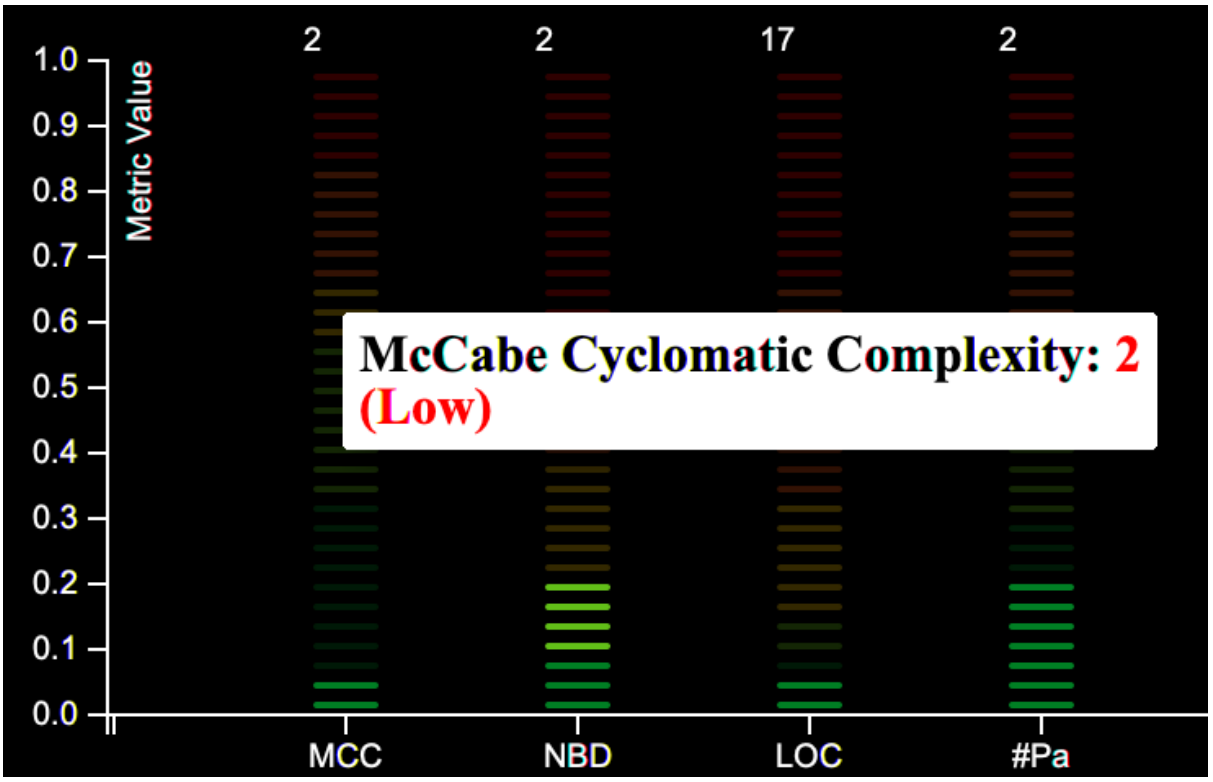
2.1.2) *reserveLocation() in ReservationService class*

File path:
reservationservice/src/main/java/nl/tudelft/sem/reservationservice/domain/ReservationService.java
Commit (this method was refactored in the past):
fd7b206b0d787ae2512346968316037799 12dffc [fd7b206]

The metrics before refactoring:



The metrics after refactoring:



Problem: *reserveLocation()* is the most complicated method in our project since reserving a location requires to

- Check if there is already some reservation in the time period that conflicts with the new reservation.
- Check if the sport wanting to play matches the location to be reserved.
- Check if the team size suits the location and the sport.
- Check if each team member is allowed to make another reservation on the day with his or her subscription type.
- Check if the start time of the reservation is from 16:00 to 23:00 (when there are no lessons).

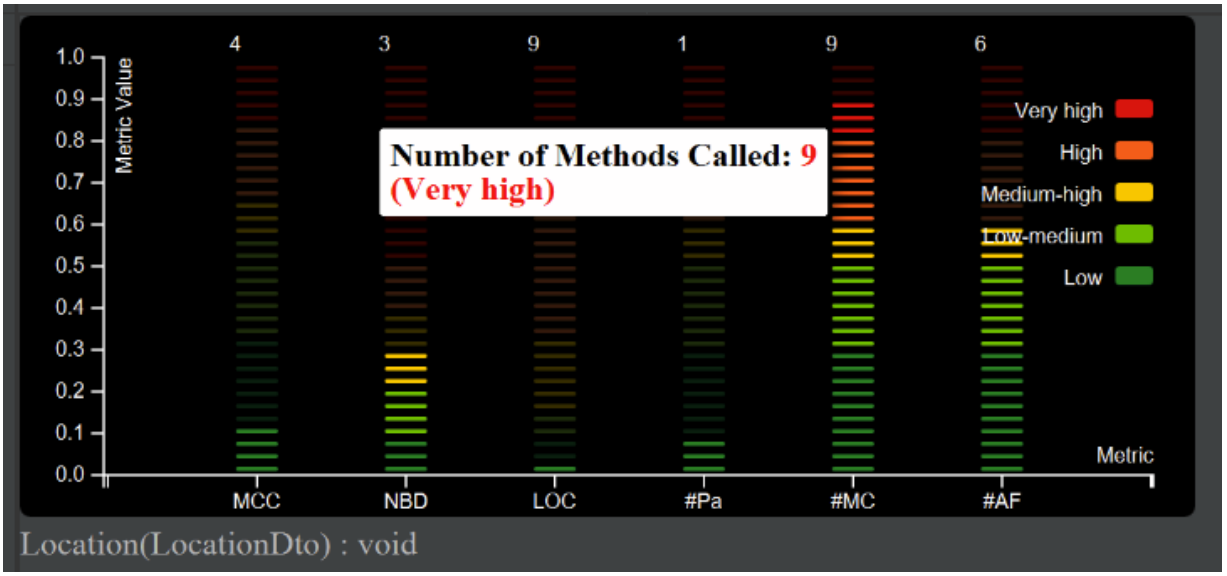
Thus, the McCabe Cyclomatic Complexity of this method is high according to the threshold and can be hard to debug and follow.

Operations applied: *Extract Method Refactoring* is used here to split the business logic for the checks in this single method into multiple private methods such that each private method handles one or two checks mentioned above. This is done by using Intellij's method extraction such that parameters and variables are properly handled in the new methods. Furthermore, the created private methods also do not have their cyclomatic complexities exceeding the threshold.

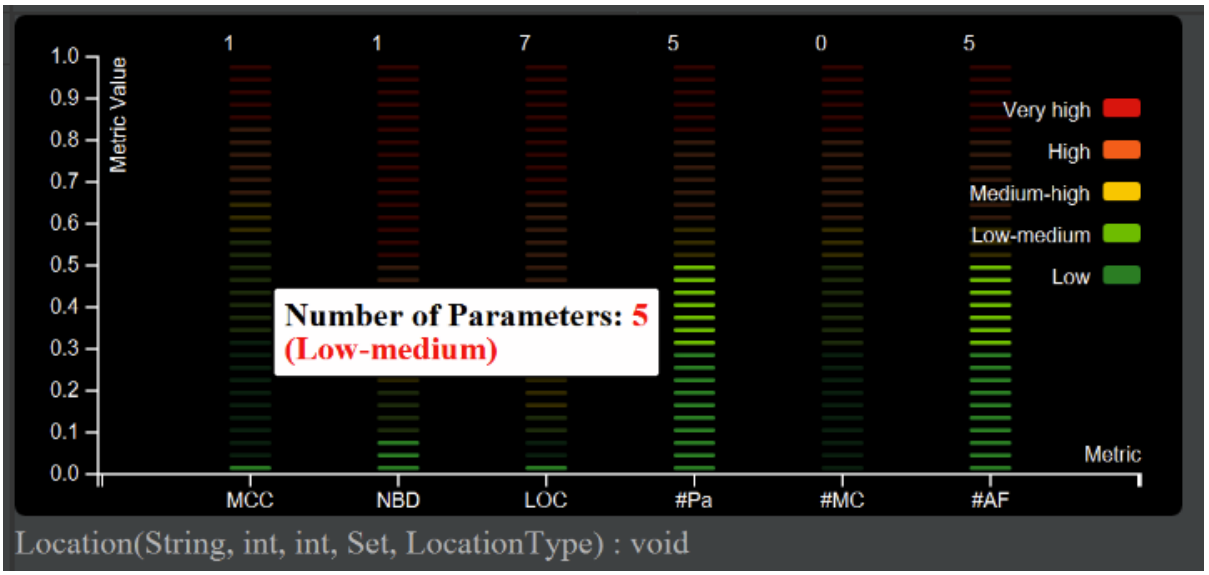
2.1.3) *Location()* in *Location* class

File path:
facilityservice/src/main/java/nl/tudelft/sem/facilityservice/domain/Location.java

The metrics before refactoring:



The metrics after refactoring:



Problem: This method took an object of the class *LocationDto* and had to call a lot of functions on that object in order to get values for all fields. The *Number of Methods Called* is higher than 8, so we decided to refactor this method to make the metric level below the threshold.

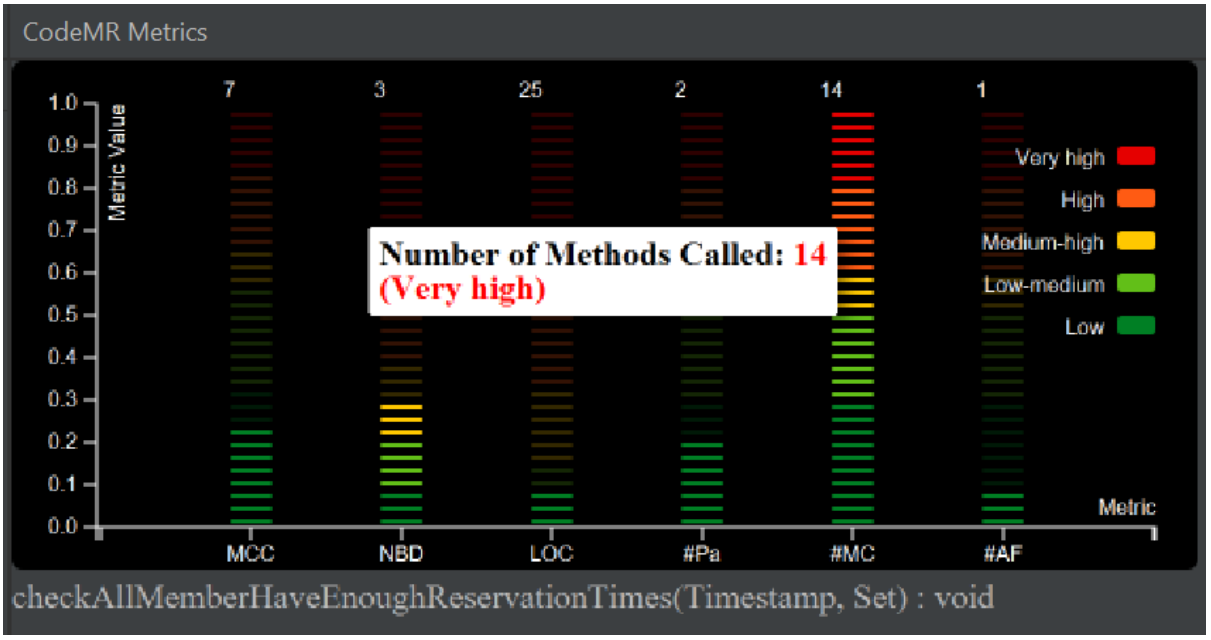
Operations applied: Using a technique similar to *Move Method Refactoring*, the responsibility of getting all the values is moved from this method to the caller of the method. This method now takes all the individual fields as arguments. The functionality of creating a

Location from a *LocationDto* is moved to the *LocationDto* class, where it fits better. Now the Location() method's *NMC* is 0, obviously below the threshold.

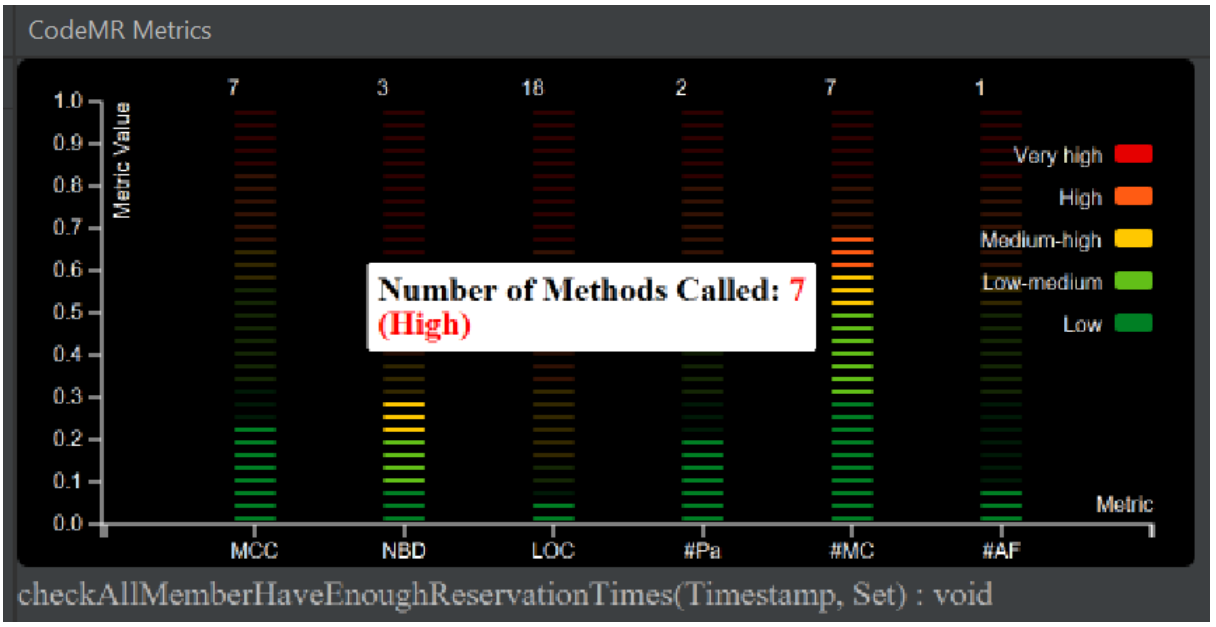
2.1.4) *checkAllMemberHaveEnoughReservationTimes()* in *ReservationService* class

File path:
reservationservice/src/main/java/nl/tudelft/sem/reservationservice/domain/ReservationService.java

The metrics before refactoring:



The metrics after refactoring:



Problem: This method's *Number of Methods Called* metric is very high (14), above the established threshold, making the method a lot bigger and unclear since it's doing a lot of things. Usually, with getters and setters in a method it makes sense, but there are in this specific method a lot of calendar setters for setting the date with the reason to look if a member has enough reservations for that day.

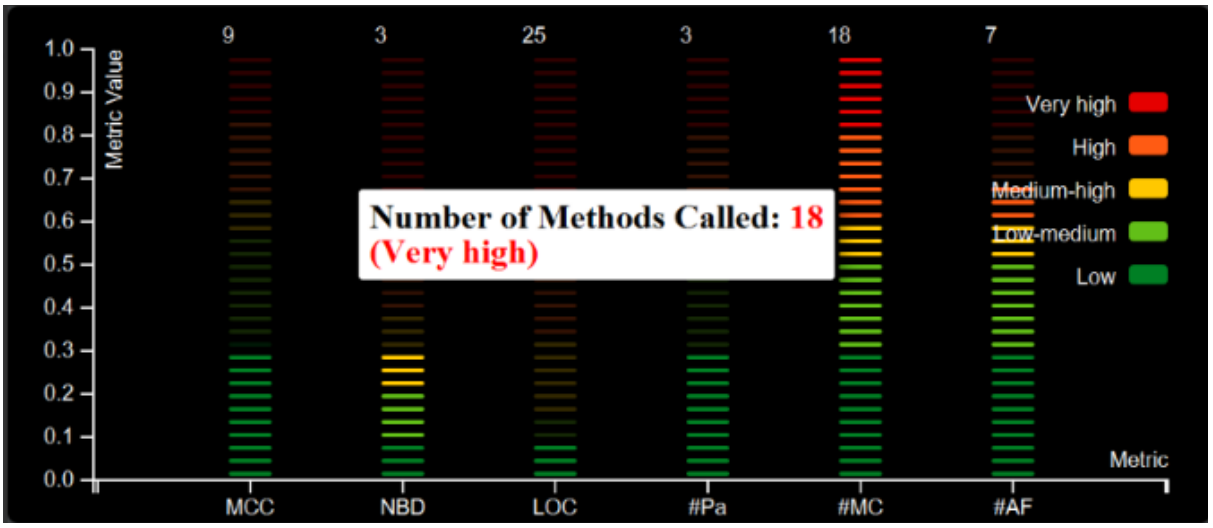
Operations applied: *Extract Method Refactoring* can be used here to split the logic of setting the calendar and the logic of checking if all group members have enough reservations left for that day. This is done by using Intellij's method extraction such that parameters and variables are properly handled in the new methods. Furthermore, the created private method has in total also fewer calls to other methods than by combining the newly created method and the old method. The *NMC* of this method is higher than the threshold so we chose to refactor it.

As a side note: The best way was by refactoring out the logic of getting the time of the beginning of the day and stopping refactoring after that. That method had 8 calls to other methods, mostly calendar setters. So by refactoring out that method, it has a *#MC* of 8. Then in the method *checkAllMemberHaveEnoughReservationTimes* it has 8 calls less, but a call to the new refactored private method, so 14-8+1 = 7, as you can see in the new metric diagram after refactoring. This is quite an improvement already, by 50%. But by looking and analysing the method *checkAllMemberHaveEnoughReservationTimes,* the remaining calls are needed to serve the main purpose of the method, checking if all members have enough reservations. After getting the timestamp of when the day begins and ends, there is a loop through all the members, calling for example the role of a member and how large the amount is of reservations of a member, retrieved from the repository. By splitting the method it will not improve the code quality but may even make the code more confusing.

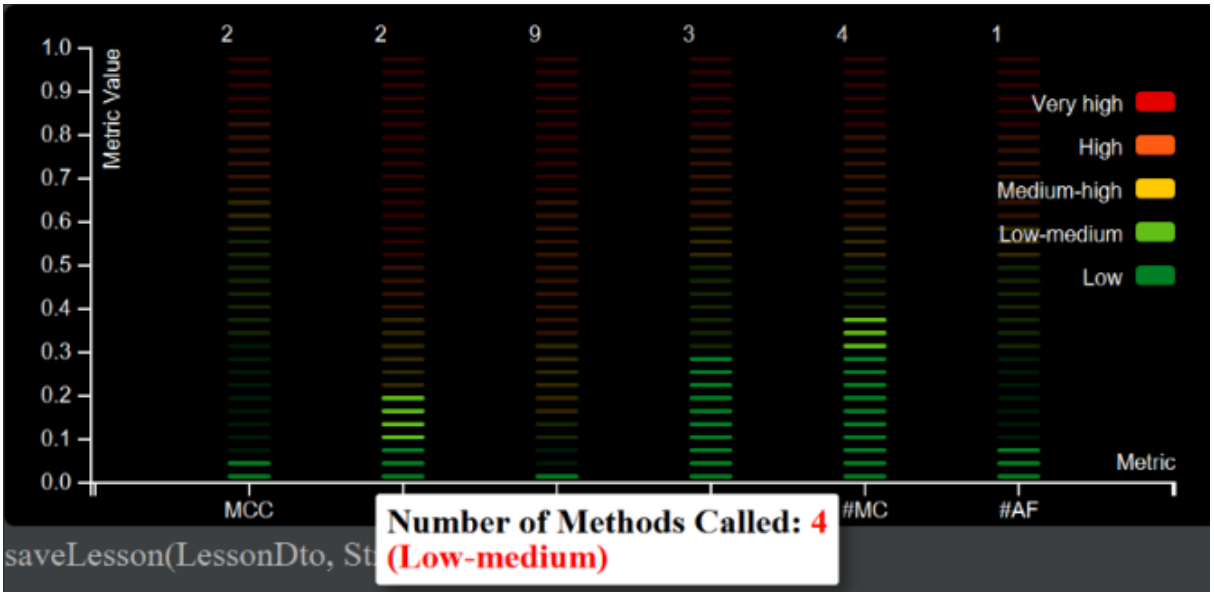
2.1.5) *saveLesson()* in *Lesson Controller*

File path:
\lessonservice\src\main\java\nl\tudelft\sem\lessonservice\controller\LessonController.java
Commit:
7a6b71df13c0f87684266e7a87e618c15457285c

The metrics before refactoring:



The metrics after refactoring:



Problem: The Number of Methods Called (#MC) for saving a lesson used to be 18 (Very high) and it is higher than the threshold: 1 *log.info()* call, 5 *log.error()* calls, 1 *isAuthenticated()* call, 1 *isAdmin()* call, 4 *getTimeStart()* calls, 3 *getHour()* calls, 1 *getMinutes()* call, 1 *save()* call and 1 *getLessonDto()* call. This method should check the authentication of the user wanting to save the lesson and the time of the lesson to be saved and, if both pass the conditions required, the respective lesson should be saved in the repository. The high number of method calls comes from the fact that the checks are done inside the method and that some methods get called multiple times. So we decided to refactor this method to make its *NMC* below the threshold.

Operations applied: Several operations were applied both before and during refactoring. Before starting the refactoring process, a number of *Extract Method Refactoring* operations were already used. For checking the time, methods *getHour()* and *getMinutes()* were added to the class such that code duplication and number of methods called were reduced, since

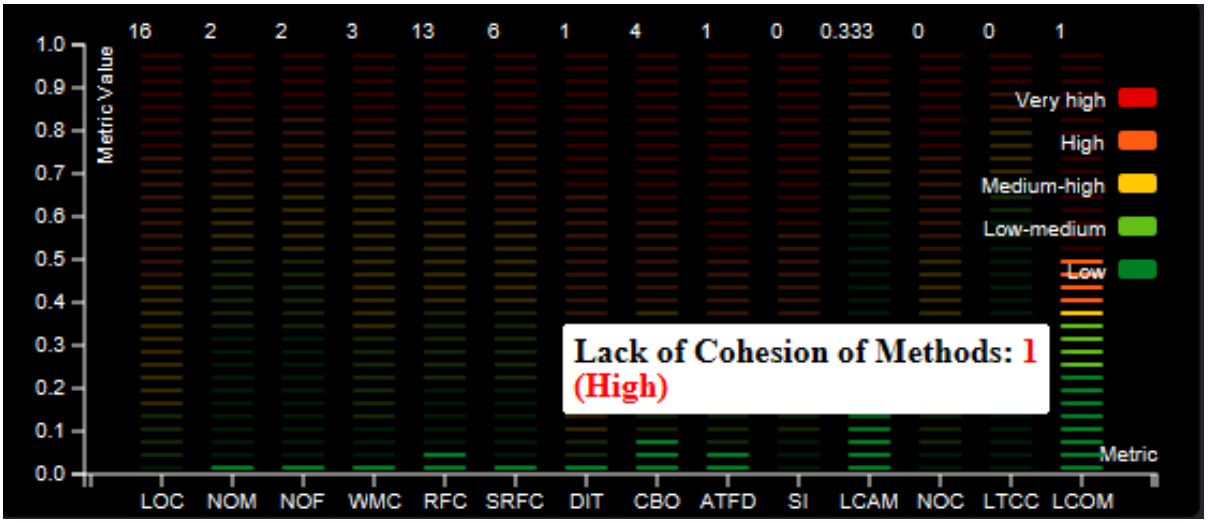
other methods in the *LessonController* class also needed the functionality provided for retrieving the hour and minutes. Code also becomes more readable and errors are less likely to appear as a result of extracting these two methods. During the refactoring process, *Extract Variable* operations were added to assign the results of *getHour()* and *getMinutes()* to variables hour and minutes in order to reduce the number of calls to those methods. Moreover, *Replace Error Code with Exception* operations were applied such that *log.info()* and *log.error()* were replaced with *NotAcceptable()* exceptions. Then, with some other *Extract Method* operations, method *checkTime()* grouped together the code fragment that checked the time requirements of a lesson, and method *checkAuthentication()* grouped together the *isAuthenticated()* and *isAdmin()* methods so that all the logic for checking the authentication of a user resides in a new separate method (which can also be used by other methods in *LessonController* to reduce code duplication and improve readability). As a result of these operations, the Number of Methods Called (#MC) for saving a lesson now is 4 (Low-medium): *checkAuthentication()*, *checkTime(), save()* and *getLessonDto().*

2.2 Class-level code refactoring

2.2.1) *UserDtoDetailsService Class*

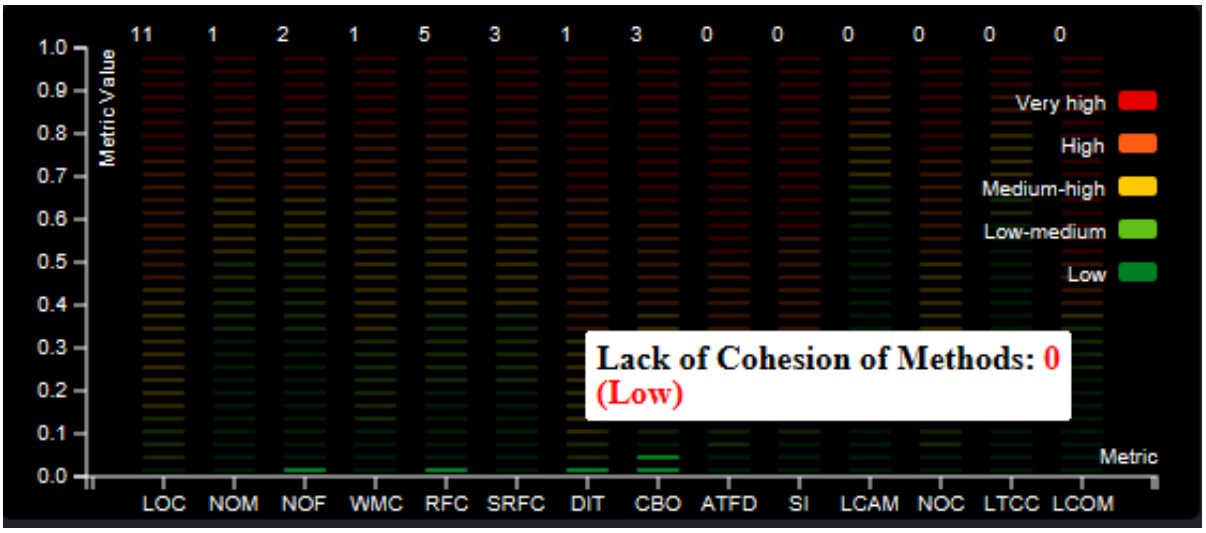
File path:
authenticationservice/src/main/java/nl/tudelft/sem/application/UserDtoDetailsService.java
Commit:
19d9df37

The metrics before refactoring:

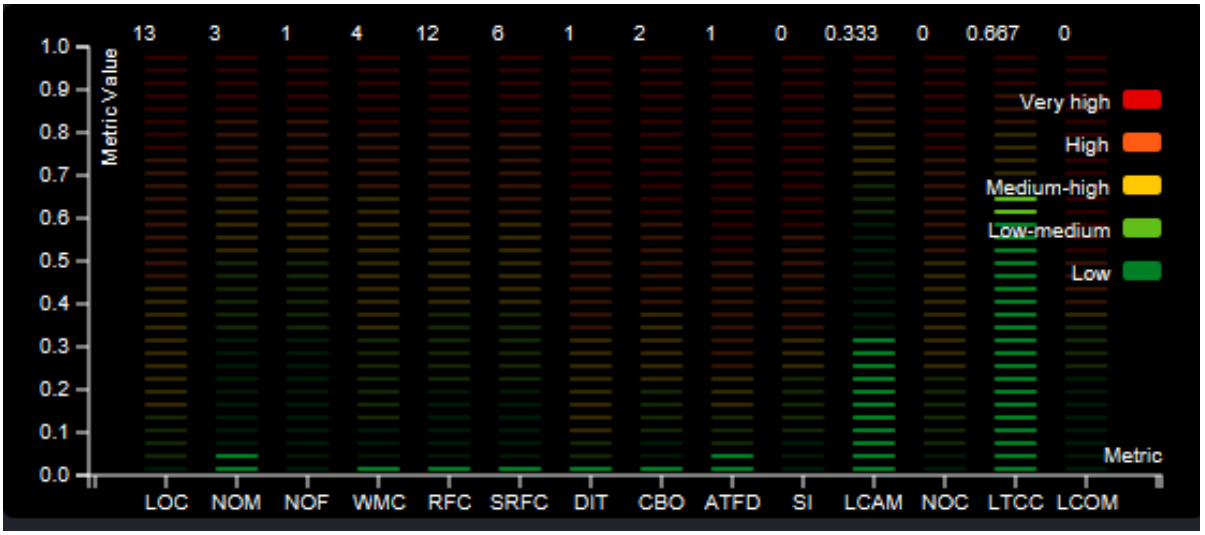


The metrics after refactoring:

UserDtoDetailsService Class:



UserRepositoryAccessor Class:



Problem: The *LCOM* metric indicated a divide within the internal class logic. This was due to the method implementing a Spring Boot security class, *UserDetailsService*, and therefore overriding the super method *loadByUsername()*, while also having a separate method (*createUser()*) which modified the *UserRepository*. While the class methods acted on similar ideas (user retrieval and user storing) they were not similar in input nor logic and were therefore not cohesive. In short, the class implemented more than one responsibility and it exceeds the threshold for *LCOM*. Therefore, we decided to refactor it.

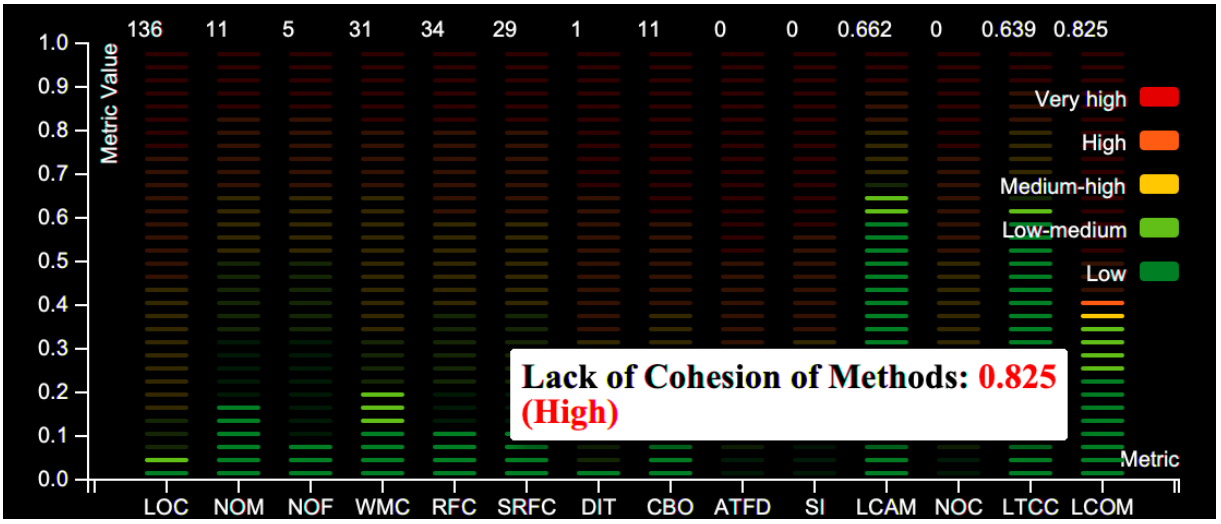
Operations applied: *Extra Class Refactoring* was applied to split the responsibility into two separate classes. This allowed the initial class *UserDtoDetailsService* to delegate *UserRepository* interactions, and therefore user storing, to another class, named *UserRepositoryAccessor*. The shift of the *createUser()* method location can be considered as Move Method refactoring. The new class is a Spring Service and can be therefore autowired both in *UserDtoDetailsService* and anywhere else that requires interaction with the repository (for example in the *AuthController* class, when checking credentials on login

or verifying successful storing on user creation). As *Move Method* refactoring requires quite a big change in code structure, tests and impacted classes had to be changed. This included writing a new test suite for the *UserRepositoryAccessor* class. As a result, the *LCOM* metric in the initial class was reduced to 0, and the new class reflected the original lack of cohesion in the *LTCC (Lack of Tight Class Cohesion)* and *LCAM (Lack of Cohesion Among Methods)* metrics, though on a more than acceptable level of 0.667 and 0.333 respectively.

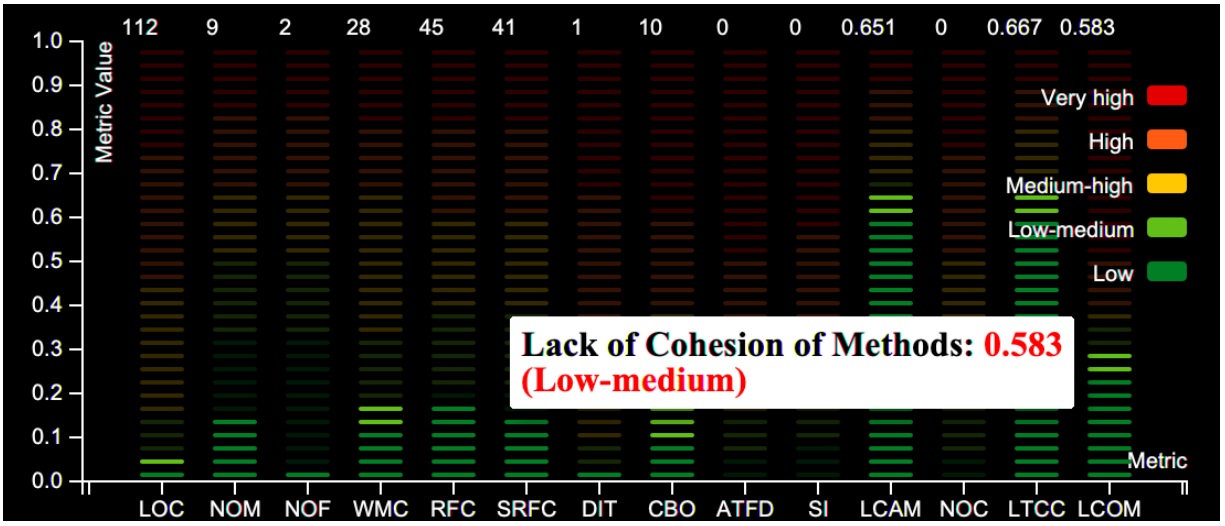
2.2.2) *ReservationService Class*

File path:
reservationservice/src/main/java/nl/tudelft/sem/reservationservice/domain/ReservationService.java

The metrics before refactoring:



The metrics after refactoring:



Problem: This class has a reservation repository injected into it and can query data from the repository and use a rest template to request data from other microservices, then combine their data to form new, more complex business logic for reservation. It contains all

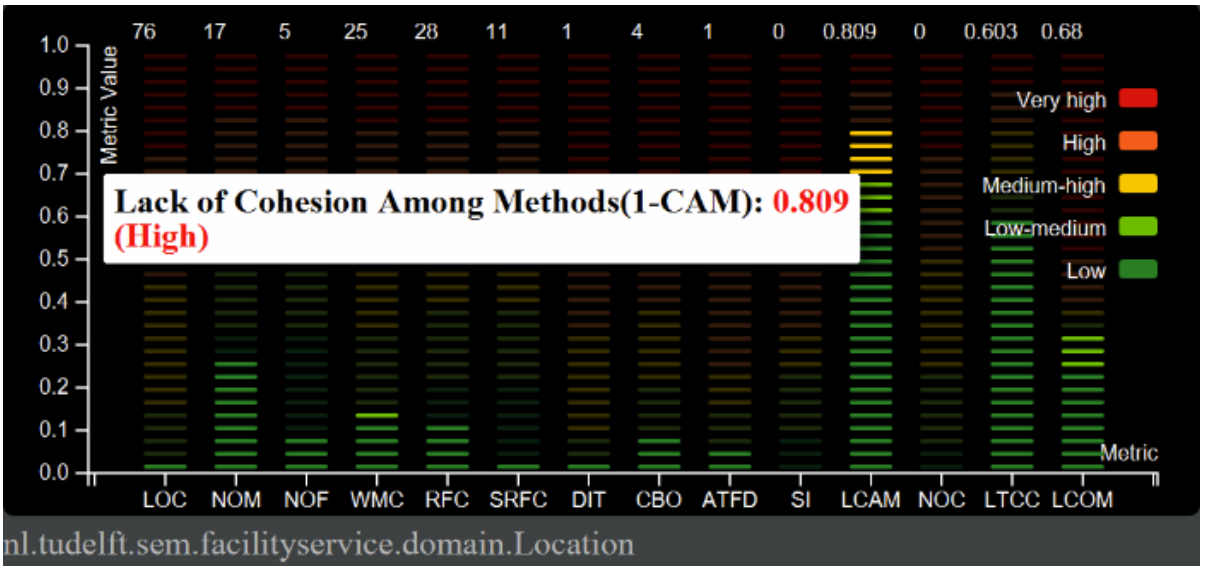
functionalities for equipment reservation, location reservation, and lesson reservation. Thus, this class is applied with too many responsibilities that cause the lack of cohesion of methods to be high (Some URL attributes are not shared, eg. reserving a lesson requires a request from lesson service while reserving a location requires a request from). Thus, we decided to refactor this class to make its *LCOM* below the threshold.

Operations applied: It was decided to apply *Extract Class Refactoring* to split the responsibilities of this class such that all business logic about lesson reservation is put to a new class called *LessonReservationService*. The reason why only split out lesson reservation is that it is considered that there would be more requests to reserve lessons on a day than the other two. Hence *LessonReservationService* will be more frequently used. Furthermore, all the methods from the main and tests folder which used to call *ReservationService* for lesson reservation are now called *LessonReservationService*.

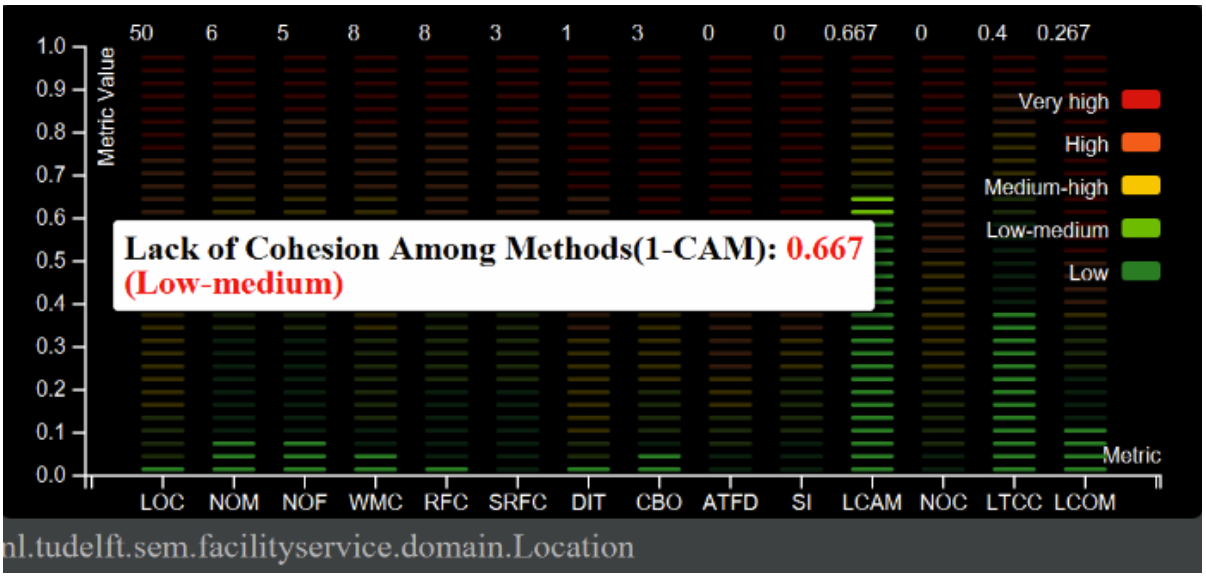
2.2.3) *Location Class*

File path:
facilityservice/src/main/java/nl/tudelft/sem/facilityservice/domain/Location.java

The metrics before refactoring:



The metrics after refactoring:



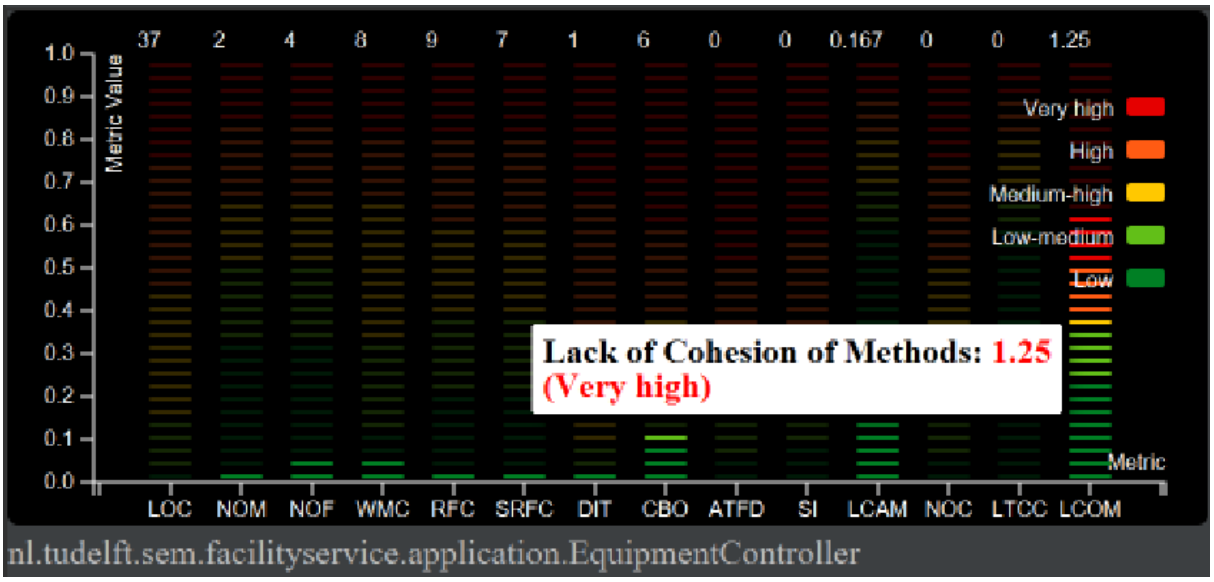
Problem: This method's *LCAM* metric was above the threshold, scoring high at 0.8. This was due to some methods in this class having tasks that were concentrated on a specific field, and other methods were doing too many operations in a single method.

Operations applied: To tackle this issue we removed some methods and refactored others, also splitting them up using the *Extract Method Refactoring* technique. The unused getters and setters are removed, which makes sure that private fields can only be accessed if necessary. Also, the method *getLocationDto()* had to convert a set of the class *Sport* to a set of the class *SportDto*, which is now done in a separate method. The *LCAM* metric level now has gone below the threshold, scoring low-medium at 0.667.

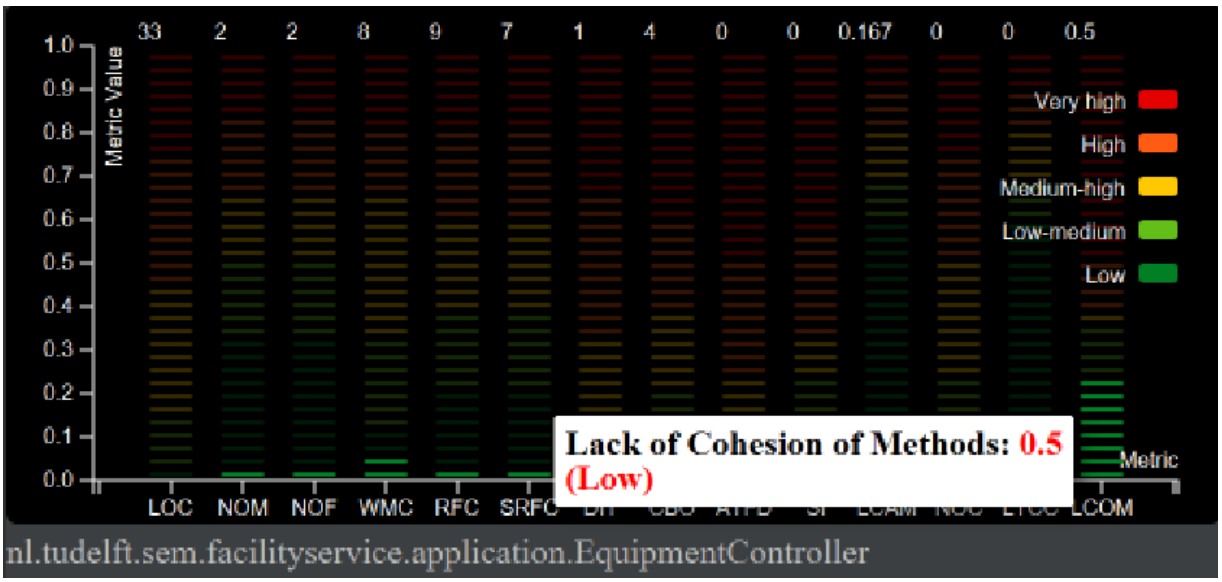
2.2.4) *EquipmentController Class*

File path:
facilityservice/src/main/java/nl/tudelft/sem/facilityservice/application/EquipmentController.java

The metrics before refactoring:



The metrics after refactoring:



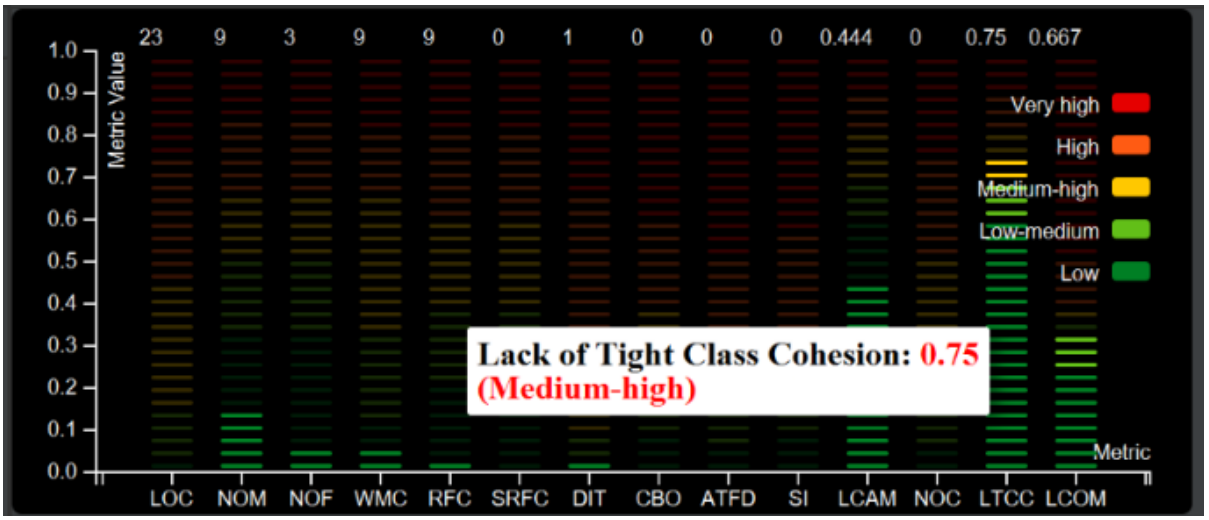
Problem: This class has a location, sport, equipment repository injected into it and can query data from the repository. Also it has a service communicator injected in it, where it can check if a user is authenticated or is an admin by communicating with other microservices. It contains all the endpoints needed for equipment. But there was still some outdated code, namely the autowired attributes at the top so the location and sport repository can be injected in the class. During our process in the project, we had first the controllers combined, but then also refactored it out. But the location and sport repository didn't get deleted in this class, so that's why the metric says there is a lack of cohesion of method, because the methods in this class don't share common attributes, namely with location and sport repository. The *LCOM* of this method exceeds the threshold so we choose to refactor this class.

Operations applied: By *Deleting Redundant Code*, namely removing the autowired location and sport repository attributes, the *LCOM* went from 1.25 to 0.5 (very high to low). It was possible since location and sport repository are used in their own controllers, location and sport controller respectively.

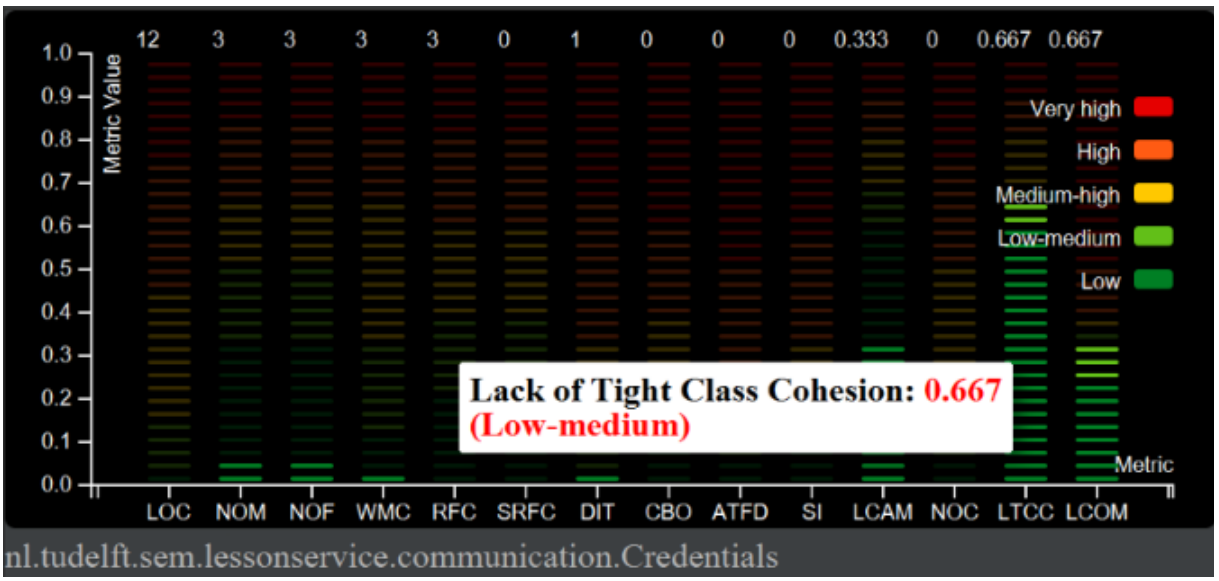
2.2.5) *Credentials* class

File path:
\lessonservice\src\main\java\nl\tudelft\sem\lessonservice\communication
Commit:
bc00b0046a1403913ba036c306f38322c6f49a1f

The metrics before refactoring:



The metrics after refactoring:



Problem: The *Lack of Tight Class Cohesion (LTCC)* metric for the *Credentials* class inside the *Lesson* microservice used to be 0.75 (Medium-high). This indicated that the directly connected public methods in this class lacked cohesion. Only the *getRole()* method of this

class was used somewhere in the application (in ServiceCommunication in the *isAdmin()* method). Thus, we decided to refactor this class to reduce the *LTCC*.

Operations applied: One type of operation used was the *Remove Setting Method*, because the values of an instance of *Credentials* should only be set at the time of creation and not get changed after. The reason for this is that the *User* microservice takes care of creating combinations of usernames and passwords and other services, out of security concerns, should not be able to change those fields. As a result of this operation, *setUsername()*, *setRole()* and *setPassword()* were removed. The *getUsername()*, *getPassword()* and *Credentials()* methods were also taken out, as they were not adding any functionality to the application. As a result of these operations, the *Lack of Tight Class Cohesion (LTCC)* metric for the *Credentials* class lowered to 0.667 (Low-medium).

Conclusion

At the beginning of the code refactoring process we analysed the results and realised that the majority of the metrics were low or low medium and at worst medium-high to high. We found issues throughout each service (except user-service where the issues are the lowest), so we decided to refactor one class and method from each. From authentication we refactored the *authController login()* method (originally very high severity on *#MC*) and the *UserDtoDetailsService* class (originally high severity in *LCOM*). Now these methods are moderate to low severity problems. From the reservation microservice the *reserveLocation()* method (originally high *MCC*), the *checkAllMemberHaveEnoughReservationTimes()* (originally very high *MC)* and *ReservationService* class (originally high severity *LCOM*) were refactored. Now the reservation service has been improved greatly and only has a low-medium *LCOM,* the method *reserveLocation()* has an *MCC* below the threshold and the method *checkAllMemberHaveEnoughReservationTimes()* also has an MC below the threshold. Next, for facility service, the *Location* class and the *equipmentController* class were refactored, as well as the *Location()* constructor method to reduce the *#MC*. Originally the *LCAM* of the *Location* class was medium-high but after refactoring it ranked only low-medium severity and the *EquipmentController* class was very high and after refactoring low. The constructor went from very high to low-medium as well. Looking at all of this it is clear that CodeMR has helped us greatly improve the quality of our methods and classes.

Note

Parameters of methods were not included since they are not relevant to the scope of the refactoring process, as to increase the readability of the document and shorten its length.

References:

1. https://refactoring.guru/
2. https://www.ibm.com/docs/en/raa/6.1?topic=metrics-cyclomatic-complexity
3. https://en.wikipedia.org/wiki/Cyclomatic_complexity
4. CodeMR