1. Code Metrics / Motivation refactorings

For identification of difficult to maintain code in our project, we used *CodeMR* (IntelliJ plugin) to compute our code metrics on both class-level and method-level.
Afterwards, we identified a list of classes and methods that performed the worst, and thus could be improved upon, leading to a top five for both classes and methods.
Below, we specify the thresholds that allow us to identify these classes or methods, and what appropriate refactoring operations we have in mind to improve on the respective computed metrics.

The 5 classes to be refactored

1. SportRoomController

Metric That Needs Improvement
This class is responsible for handling sports room related API requests from external microservices, and subsequently responds to these requests by calling service layer logic, depending on the type of HTTP request.
However, this controller contained many different functionalities for different HTTP requests. Hence, *CodeMR* showed a relatively high score for *Class Lines of Code (CLOC):* 119. This can make the class hard to maintain and read in the future.
Furthermore, due to this high number of methods in the class, the *WMC (Weighted Method Count)*[1] metric score was 25. High score on this metric signifies increased complexity of the class, and thus lower maintainability and testability.
We noticed that a majority (62%) of the REST API methods in this controller were responsible for GET and POST HTTP requests. All these GET and POST HTTP requests are then handled by calling getter and setter methods in the sports room service layer. Thus, we can split up these two main distinct functionalities provided to improve on both *CLOC* and *WMC*, since they are highly intertwined.

Results and Refactoring Techniques
We believed we could improve on the size (and complexity) of the class *SportRoomController* using *'Extract Class Refactoring*; the methods responsible for GET and POST requests are now divided into two new classes, namely *GetSportRoomController* and *SetSportRoomController*.
This helped reduce the *CLOC* metric from 119 to 44 and *WMC* from 25 to 9, meaning the class size, and thus class complexity, shrank substantially. The *CLOC* scores for the newly created classes are also low: these are 46 and 33 for *GetSportRoomController* and *SetSportRoomController* respectively.
All in all, this refactoring ensures this class is much more maintainable in the long run.

---

[1] The weighted sum of all class' methods represents the McCabe complexity of a class. It is equal to the number of methods, if the complexity is taken as 1 for each method. (CodeMR)

2. *ReservationController*

Metric That Needs Improvement

Just as with the previous class *SportRoomController*, this class *ReservationController* is supposed to handle reservation related REST API requests by making use of reservation service layer functionality. Its methods should allow other microservices to send requests and retrieve results to and from the *Reservation* microservice.

However, this class also contained methods which communicated with external microservices, namely the *SportsFacility* microservice and *User* microservice, and returned their results. These methods were not REST API methods that belong in this class.

Hence, this class had a high score on the metric *Class Lines of Code (CLOC)*[2] of 144, and 132 on *Class methods - lines of Code (CM-LOC)*[3]. A high score for both of these metrics define the size quality attribute of the class. A larger class can cause issues in maintaining the class in the future, since it becomes difficult to determine the optimum number of white box test cases needed to achieve high test coverage.

Results and Refactoring Techniques

To improve this class' size, '*Extract Class Refactoring'* was applied, since we could split up incoherent functionality. All the methods which communicated with the *SportsFacility* microservice and *User* microservice, as described above, were removed from the *ReservationController* and divided among the new *SportsFacilityCommunicator* class and *UserFacilityCommunicator* class appropriately.

This way, all the classes in the *Reservation* microservice can communicate with other microservices through the two newly created classes instead of communicating through the *ReservationController,* which now is only responsible for *Reservation* microservice related requests. This refactorization reduced the *CLOC* to 131 and *CM-LOC* to 121.

---

[2] The number of all nonempty, non-commented lines of the body of the class. CLOC is a measure of size and also indirectly related to the class complexity. (CodeMR)

[3] Total number of all nonempty, non-commented lines of methods inside a class. (CodeMR)

3. *ReservationService*

Metric That Needs Improvement

The *ReservationService* class has a low-medium score for *Lack of Cohesion (LOC)*. The cohesion score is determined via three metrics, namely the *Lack of Cohesion of Methods* (*LCOM), Lack of Cohesion Among Methods (LCAM)* and *Lack of Tight Class Cohesion* (*LTCC)* scores. For this class, the scores were 0.0, 0.662, and 0.491 respectively.
Since the *LCOM* score is already as low as it can possibly be, we will need to reduce the *LCAM* and *LTCC* score to turn the overall lack of cohesion score from low-medium to low.
The *LCAM* score is defined by the parameter types of methods. A class is cohesive if all methods use the same type of parameters, since it can then be assumed that these methods process related information.
The *LTCC* score is measured by the relative amount of directly connected public methods in the class. A method is connected if they share attributes or have a method call among them.
The cause of our lack of cohesion is due to the fact that *ReservationService* now fulfills two different responsibilities. The majority of methods communicate with the reservation repository, while two others check the validity of a reservation. To lower our lack of cohesion score, we could apply "*Extract Class Refactoring*".

Results and Refactoring Techniques

As suggested above, we applied "*Extract Class Refactoring"* by removing the methods that did not access the repository and put them in a separate class, called *ReservationChecker*. This improved cohesion, since the methods of the class now all relate to a single task, communicating with the repository. The improvement is reflected in the *LCAM* and *LTCC* scores, which turned into 0.58 and 0.378.
The *LCAM* score has been reduced, because the removed methods used a parameter that was not shared by any other method. After refactoring these methods, the parameter types of *ReservationService* have become more consistent. Furthermore, assigning these methods to a new class has also reduced the *LTCC* score. The disconnected methods did not use the main attribute of the class, reservation repository, and did not invoke any other methods. Therefore, the *LTCC* has been lowered, as two methods that did not have any connections have been taken out.
The individual reductions of the *LCAM*- and *LTCC* score have in turn managed to lower the overall lack of cohesion score from low-medium to low.
This refactorization has made our *ReservationService* much more understandable, and thus testable and maintainable in the long run, since different functionalities are now assigned to dedicated classes.

4. *EquipmentNameStrategy*

Metric That Needs Improvement

This class is used to sort reservations based on the names of pieces of equipment, as the name suggests. It is one of the “Concrete Strategies” part of our “Strategy Pattern”. Hence, it is supposed to mainly be concerned with sorting logic.

However, using the *CodeMR* metrics report we detected that this class had high *Lack of Tight Class Cohesion (LTCC)*, due to the fact that it makes a call to external microservices using the *restTemplate*. So our class does not have a single responsibility anymore; it is sorting the reservations based on the name of the equipment, but it also makes an API call to *SportFacilityService* to get the name of the equipment for that reservation.

Hence, this led to a score of 1.0 for the metric *LTCC.*

To improve on cohesion, we want to avoid this class having to make external calls via the *restTemplate* field.

Results and Refactoring Techniques

To solve this problem, we added a *String* attribute *bookedItemName* in the *Reservation* entity’s constructor, which stores the name of either the *SportRoom*, *Equipment* or *Lesson*. In that way, the name can be retrieved using a getter in *Reservation* class.

Therefore, we did not need the *restTemplate* field anymore to make the call to the other service. As a result, the *restTemplate* could be removed from the constructor of the *EquipmentNameStrategy* class. These operations made the lack of tight class cohesion *(LTCC)* go from 1.0 to 0.0.

This refactorization ensures that the respective class now contains coherent logic, related to sorting reservations, instead of being responsible for making external calls as well. All in all, the class will be easier to understand for us as developers, which leads to better testability and maintainability in the future.

5. BasicPremiumUserStrategy

Metric That Needs Improvement
Similar to the class above, this class is one of the “Concrete Strategies” part of our “Strategy Pattern”. This class sorts reservations on the type of subscription, instead of equipment name.
We have detected the same issue here as we had for *EquipmentNameStrategy* and the reason is the same as explained above. The class has *restTemplate* as parameter in the constructor, and this *restTemplate* is used to make API calls to other microservices.
This makes the *BasicPremiumUserStrategy* have to call another microservice for every reservation, since it needs to check if the user that made the reservation has a premium subscription or not. We thought this needed to be refactored since the *BasicPremiumUserStrategy* should only have a single responsibility, which is sorting the reservations and not the one of communication with other microservices.

Results and Refactoring Techniques
In order to achieve high cohesion, we applied the same methodology we used for refactoring *EquipmentNameStrategy*. So, we added another attribute in the *Reservation* entity which stores if the user who made the reservation has a premium subscription or not (*boolean isPremium*). This allowed us to remove the *restTemplate* from the constructor of the *BasicPremiumStrategy* class, which calls the user microservice. The way we get this information now, is through a call to the *Reservation* entity with the method *getMadeByPremiumUser()*. This operation made the cohesion go all the way from 1.0 to 0.0. Again, this refactorization ensures that the respective class now contains coherent logic, related to sorting reservations, instead of being responsible for making external calls as well. All in all, the class will be easier to understand for us as developers, which leads to better testability and maintainability in the future.

The 5 methods to be refactored

1. *checkReservation()* (class: *ReservationChecker*)

Metric That Needs Improvement

This method provides important logic when it comes to validating reservations that are created, for either a piece of equipment or a sports room (hall/field). In our system, each reservation object gets checked by passing it through a “Chain of Responsibility”, which consists of several validators, each one responsible for checking a certain condition. However, this single method was responsible for initializing this chain, passing the reservation object to be checked to the first validator in the chain, and also verifying whether or not the reservation is valid.

Hence, the metric *Lines of Code* in this method had a high value of 21, mainly due to the fact that no helper methods were being used. The result of this metric was a clear sign to us that we should improve on the maintainability of this method. Having long methods (that do not carry a single responsibility) leads to bad readability as well as testability, thus increasing the chance of bugs in the source code.

Results and Refactoring Techniques

We eventually reduced the metric *LOC* from 21 to 9 by applying “*Extract Method Refactoring*”. We added a new helper method *createChainOfResponsibility()*, which will now be responsible for creating the Chain of Responsibility, as the name suggests, by instantiating all required validators.

This means that the main *checkReservation()* method merely has to call this helper, so it can subsequently pass the reservation object to the first validator in the chain.

Thus, we have split up different responsibilities, leading to a reduced score for the metric LOC, which was the goal of this refactorization. Additionally, we have added a new method containing coherent logic for creating the “Chain of Responsibility”, that can be reused in the future. This was not feasible before, as this logic was embedded in the main method *checkReservation()*.

2. *makeEquipmentReservation()* (class: *ReservationController*)

Metric That Needs Improvement

The method *makeEquipmentReservation()* is used for checking for available equipment, and making a reservation for this piece of equipment.
This method had a quite high value of 25 for the *Lines of Code* metric, which we decided could be improved.
This score was due to two reasons, the first being that there were three try-catch clauses, while it could be done with two, and second that no helper methods were being used. While the code still worked, it was hard to understand because the method was so long. It also became harder to test, since it was difficult to determine what the cause of a failed test was.

Results and Refactoring Techniques

To improve the *LOC* of this method, we applied two refactorings.
Firstly, we edited the try-catch clauses so that there would only be two instead of three, which also made the code a lot more readable.
Secondly and most importantly, we applied *"Extract method refactoring"* by adding two helper classes, *getAvailableEquipmentId()* and *createAndCheckEquipmentReservation()*.
GetAvailableEquipmentId() is now responsible for creating an equipment id based on the equipment's name (and now this method can also be reused by other methods in the future for this purpose).
CreateAndCheckEquipmentReservation() is responsible for creating the reservation and passing it through the "Chain of Responsibility" to check whether the reservation can actually be made. If there is an exception thrown somewhere along the way, it will catch the exception and forward (throw) it so that the main method, *makeEquipmentReservation()*, can catch and handle it appropriately. After the refactoring, the *LOC* metric value went from 25 to 13, which is a good improvement.

3. *makeSportRoomReservation()* (class: *ReservationController*)

Metric That Needs Improvement

The method *makeSportRoomReservation()* is used for checking for available sports rooms (halls/fields), and making a reservation for the desired room if it has been validated successfully by our “Chain of Responsibility”.
This method had a quite high value for the *Lines of Code* (23) which we decided could be improved.
This score was mainly due to the fact that no helper methods were being used, similar to the method described above. While the code still worked, it was hard to understand because the method was so long, and it also became harder to test since it was difficult to determine what the cause of a failed test was. All in all, sufficient reasons for bugs to appear in our code.

Results and Refactoring Techniques

To improve the *LOC* of this method we applied *“Extract method refactoring”* by adding two helper classes, *getSportRoomName()* and *createAndCheckSportRoomReservation()*.
GetSportRoomName() is now responsible for retrieving the name of the sport room, and this method can also be used by other methods for this purpose.
CreateAndCheckSportRoomReservation() is responsible for creating the reservation and passing it through the “Chain of Responsibility” to check whether the reservation can actually be made. If there is an exception thrown somewhere along the way it will catch the exception and forward (throw) it so that the main method, *makeSportRoomReservation()*, can catch and handle it appropriately. After the refactoring, the LOC metric value was reduced from 23 to 16, which makes this method more maintainable in the long run.

4. ***handle()*** (class: *SportFacilityAvailabilityValidator*)

Metric That Needs Improvement

The *SportFacilityAvailabilityValidator* is part of our “Chain of Responsibility”. It checks availability of a piece of equipment or a sports room to be reserved, by checking the starting time (1) and checking whether the given sports room or equipment is available for the selected time (2).

For its main *handle()* method, the metric *McCabe Cyclomatic Complexity (MCC)* of 11 is noticeably higher than any other method in our code base.

It became clear that this is due to the many checks being done on this reservation object, leading to a long, complicated method. Hence, this leads to many decision points in the code, and thus leading to bad readability and testability. Also, it is common for bugs to occur more often when the design complexity exceeds what the developers themselves can handle.

Thus, we could definitely lower the number of sub-conditions being checked in this main method *handle()*. Focusing on *sports room* reservations, rather than *equipment* reservations, will have the biggest impact on the MCC to be reduced, since this certain reservation type contributes mostly to the number of conditions being checked.

Results and Refactoring Techniques

To improve on the complexity, and thus maintainability, of this main *handle()* method, we decided to apply “*Extract Method Refactoring”.* We split up different sub-conditions that are to be checked for any reservation object by introducing helper methods.

The two new helper methods we introduced, *checkTime()* and *checkExistingAvailableSportsRoom()*, are now responsible for checking the reservation starting time (1) and checking whether the given sports room or equipment is available for the selected time (2) respectively.

Whereas these checks were all being done in this single complex *handle()* method before, they are now embedded in separate cohesive methods instead. This also allows us to reuse this logic provided by these helper methods for other functionalities in the future.

This refactoring leads to a significantly lower number of possible flows in this main *handle()* method seeing that the *MCC* score went from 11 to 4.

5. ***handle()*** (class: *UserReservationBalanceValidator*)

Metric That Needs Improvement

Similar to the method above, this *handle()* method is part of one of our validators in the "Chain of Responsibility" for checking reservations. In this case the *UserReservationBalanceValidator()*, which is responsible for checking whether or not customers have exceeded their daily limit on sports room reservations.

In this main *handle()* method, the metric *Number of methods called (#MC)* of 17 is significantly high, and this suggests to us to improve on the maintainability of this method, since this implies that this method is highly coupled to those external classes it relies on. Hence, changes in production code in the one class requires us to make changes in this method as well, which violates the "*Open-Close Principle".*

We indeed noticed that this method is making many calls to external (library) classes to retrieve objects or attributes; it can be considered a "*Feature Envy"*.

Since we cannot change standard library code however, applying *"Move Method Refactoring"* on these library classes is no option; rather we will improve on this metric by refactoring our own *handle()* method.

Results and Refactoring Techniques

To reduce the *#MC* score, we applied "*Extract Method Refactoring"* by adding dedicated helper methods that return values we needed to perform certain checks in the main *handle()* method. As an example, to check the reservation count on a certain day for users, we required *LocalDateTime* objects denoting the start and end of a day and string manipulation by calling *String.substring()*, to be able to query the database for reservations between those times. Instead of creating these external library classes in the main *handle()* method, we now dedicate two helper methods which return these objects: *getStartOfDay()* and *getEndOfDay()*.

Furthermore, we reduced the number of method calls by instantiating one variable for the starting time of the reservation to be checked at the top of the method. This field can now be referred to, instead of calling the reservation method *getStartingTime()* several times to refer to this attribute.

In the end, the *#MC* score was reduced from 17 to 8.

The 5 refactored classes

Class	Class-level metric	Score Before	Score After
SportRoomController	Class Lines of Code (CLOC) Weighted Method Count (WMC)	119 25	44 9
ReservationController	Class Lines of Code (CLOC) Class-Method Lines of Code (CM-LOC)	144 132	131 121
ReservationService	Lack of Cohesion (LCohesion) • Lack of Cohesion Among Methods (LCAM) • Lack of Tight Class Cohesion (LTCC)	Low- Medium • 0.662 • 0.491	Low • 0.58 • 0.378
EquipmentNameStrategy	Lack of Tight Class Cohesion (LTCC)	1	0
BasicPremiumUserStrategy	Lack of Tight Class Cohesion (LTCC)	1	0

The 5 refactored methods

Method	Class	Method-level metric	Score Before	Score After
checkReservation()	*ReservationChecker*	Lines of Code (LOC)	21	9
makeEquipmentReservation()	*ReservationController*	Lines of Code (LOC)	25	13
makeSportRoomReservation()	*ReservationController*	Lines of Code (LOC)	23	16
handle()	*SportFacilityAvailabilityValidator*	McCabe's Cyclomatic Complexity (MCC)	11	4
handle()	*UserReservationBalanceValidator*	No. of Method Calls (#MC)	17	8