ASSIGNMENT 2

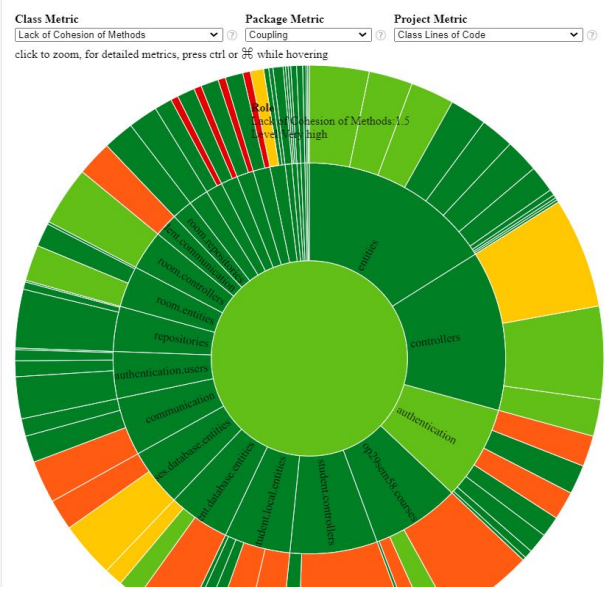
Software Engineering Methods 2020

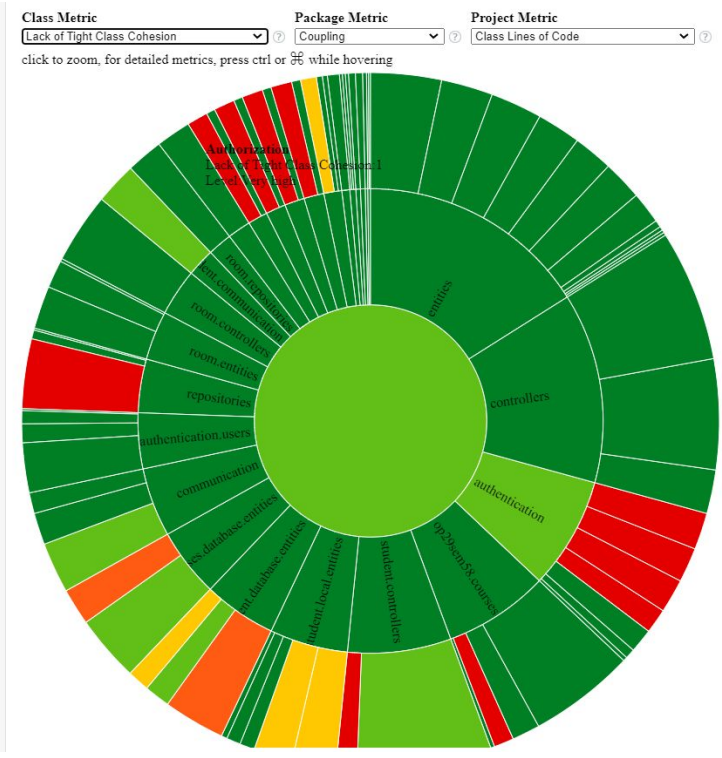
TEAM MEMBERS:
MICHEL BULTEN;NIKOLAOS EFTHYMIOU;LUCA HAGEMANS;ALEXANDRU LUNGU;ALEX MAAT;EE XUAN TAN

Refactoring

To get an overview of the current code metrics for our project, we used the CodeMR tool. This tool provides information of several metrics on class level, package level and project level. We used the class level information to decide what classes we wanted to refactor. The tool also calculates some method-level metrics which we used to decide the methods that needed a refactor.

We cycled through the different metrics that were provided in the overview and most of the graphs consisted of green, there were however two that stood out. We noticed that there were a few poorly scoring classes when it came to the Lack of Cohesion of Methods metric (see the red streaks in the outer ring of the image to the right). These were the Role enum classes that were used by all controllers to check whether the logged in user has the permission to access a certain endpoint. Since the goal of this enum class is to be used by all the controllers it is logical that this enum class scores badly. We decided to let these classes as they are because they are a very minor part of the system.





The other metric that stood out was lack of cohesion of methods. Again, the worst performing classes had to do with the authorization system on the different services (see the image on the left). The Authorization class has a method to check whether the logged in user has a certain role, this method is used by all endpoints to validate the user. We used similar reasoning as above (they are a small part of the system) and decided to leave these classes as they were.

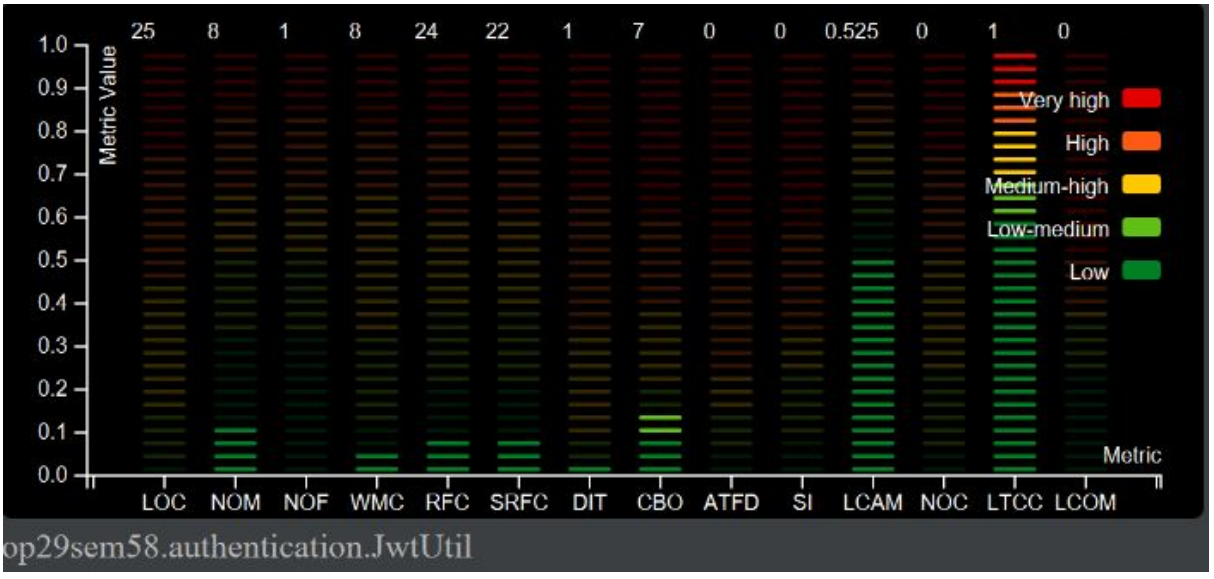
Since we noticed that the badly scoring classes and methods were pretty much evenly divided among all the services (except the Rooms service), we decided to choose one

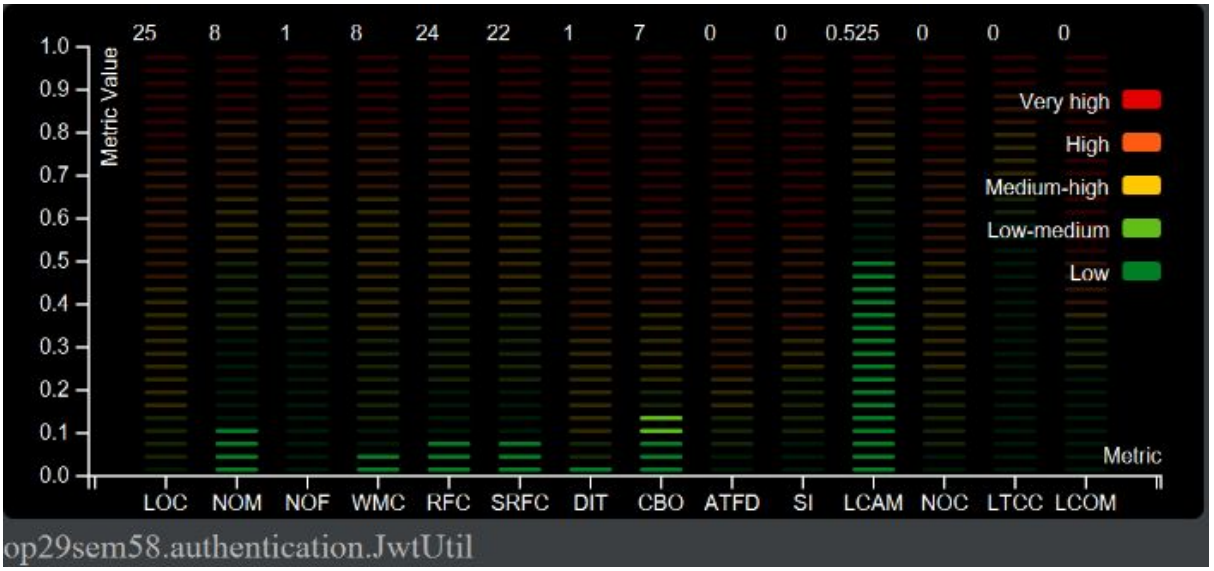
method and one class for each of the services, so that everyone can refactor something on the service that they worked on during the entirety of the project. We will now go over each of the microservices and motivate the decision of the class and method that were chosen to be refactored and explain what we did to improve them.

Authentication

Class: *JwtUtil*
The class that we are going to refactor for the authentication service is *JwtUtil*. We chose this particular class over the others in this service that had more issues because changing those was not a possibility, as most errors came from overriding methods and there is no feasible alternative to overcome that problem. As can be seen in the first picture, the only problematic metric is the Lack of Tight Class Cohesion(LTCC) which is the maximum possible value, 1. We will try to lower this as much as possible.

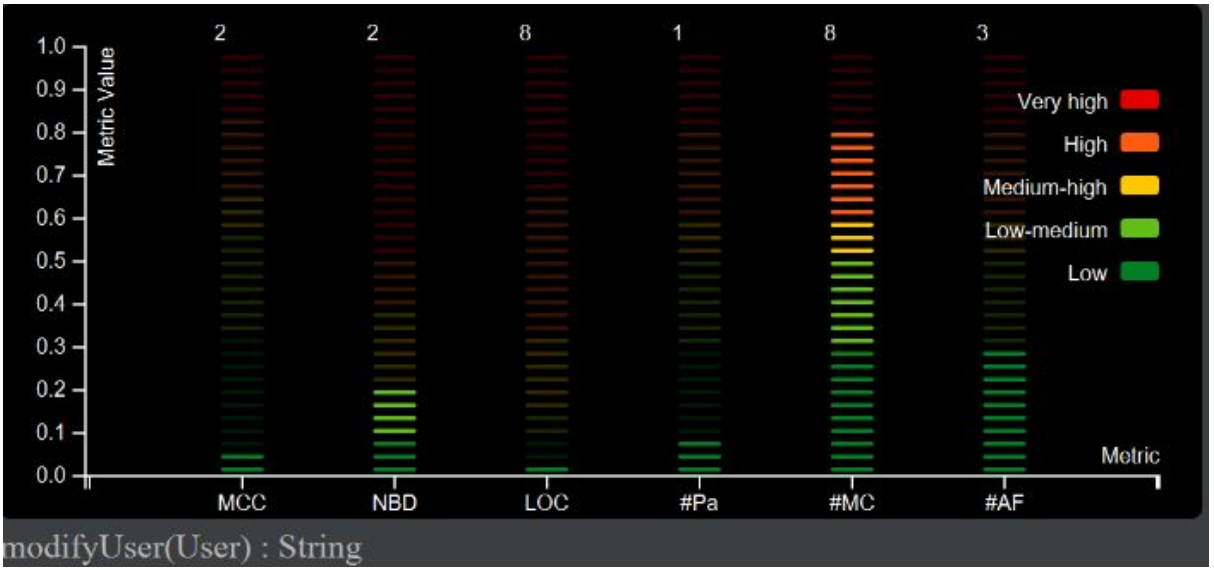
After we saw that most of the methods in this class did not need to be public at all, we simply changed them to package access and this change alone managed to bring this metric down to 0, effectively improving it by 100%.

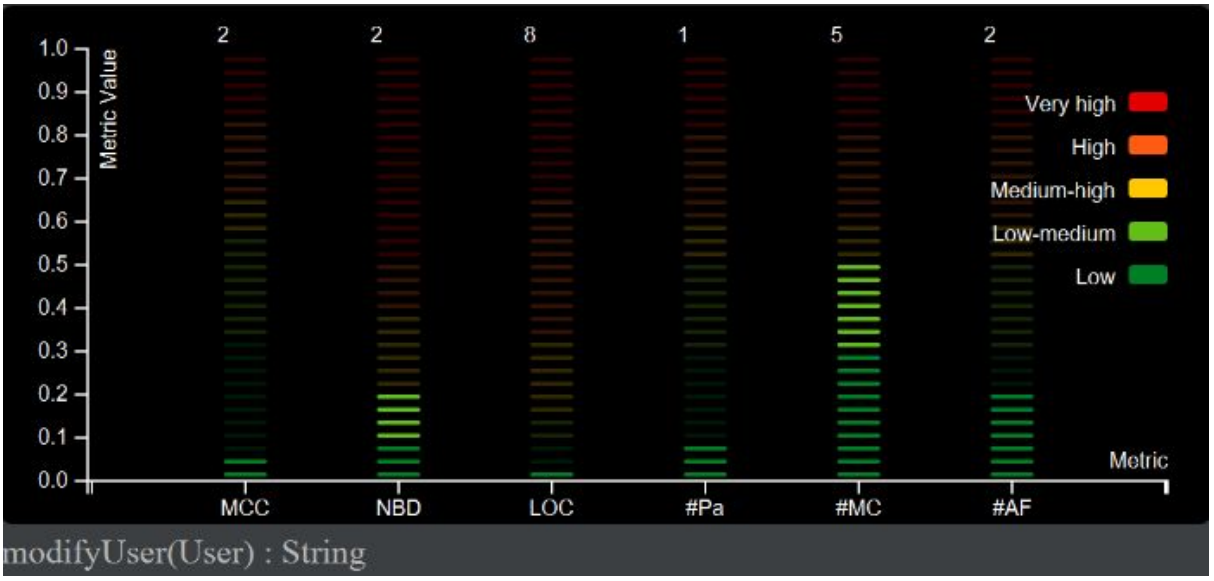




Method: *modifyUser*
The method we chose to refactor is the *modifyUser* method in the *UserService* class. The reason for this is that this method has a somewhat large number of methods called, which is 8. Ideally we want this to be at most 5 to be considered a good value.

After a bit of refactoring by splitting the method, the value is now down to 5. This is sufficient to now be considered as an appropriate amount of calls.





Courses

Class: *ServerCommunication*
The class that will be refactored in the *Courses* microservice is *ServerCommunication*. This class has a Lack of Cohesion of Methods of 1, according to CodeMR, which is high. The Lack of Tight Class cohesion is also 1, which is very high. Therefore, we will try to change this class. This class consists of only one ‘real’ method, some object variables and some getters. Luckily, we also have the Authorization class, which has a similar method implemented as the ServerCommunication class. This means that with some refactoring, this class with bad scores for the different software metrics can be removed. This will improve the overall rating of the project, as a class with a bad score in some categories is removed.

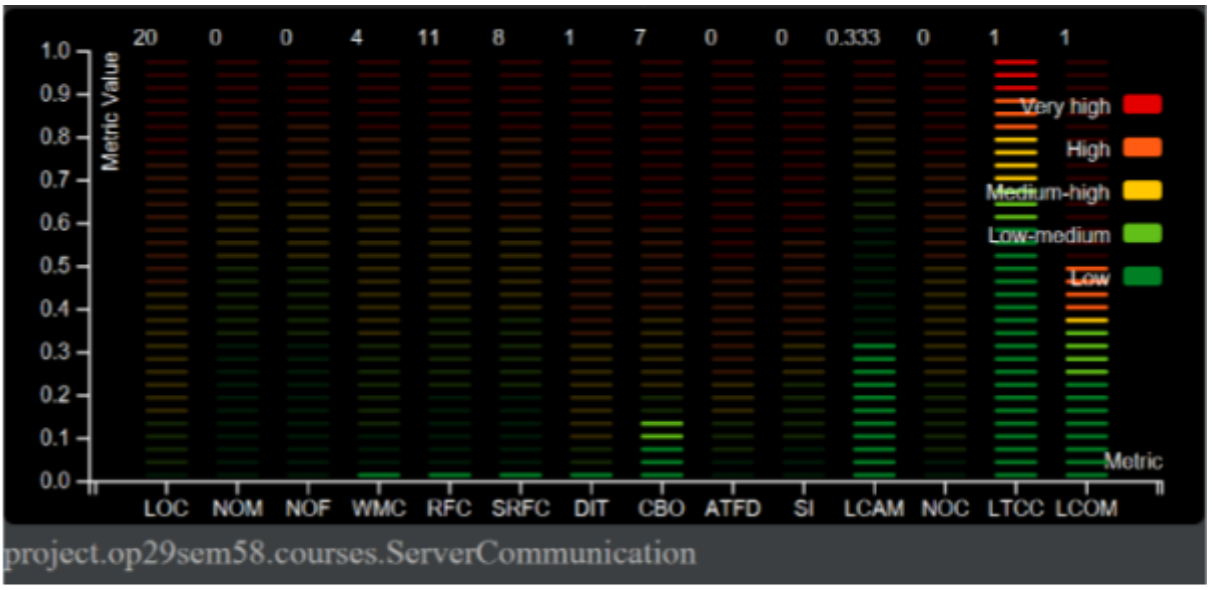
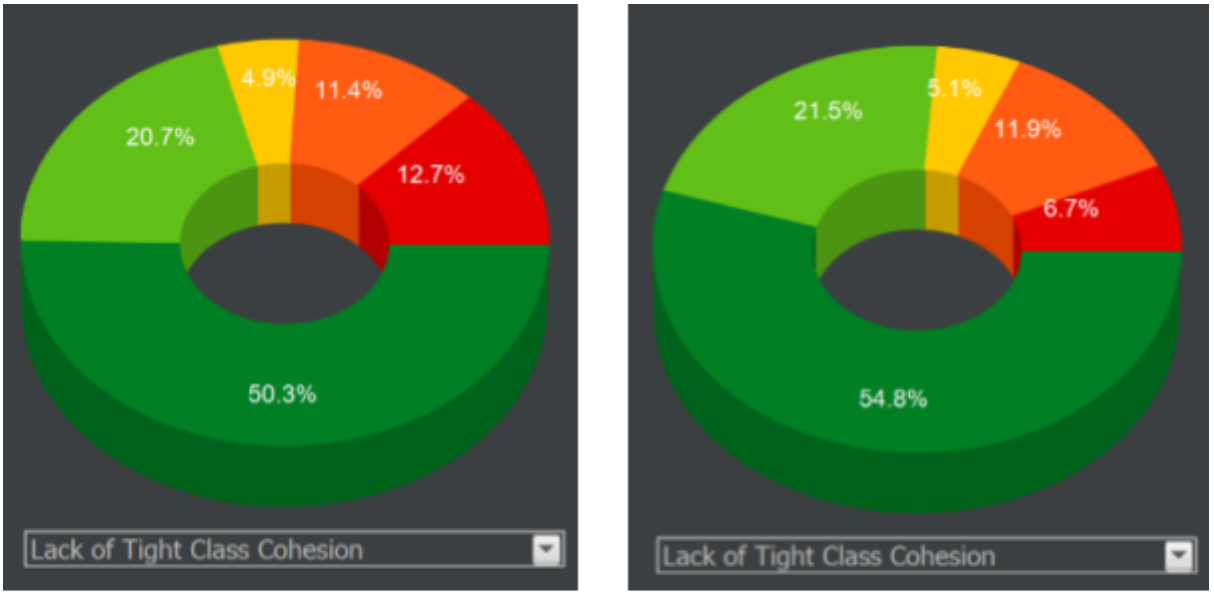


Figure 1: software metrics of the ServerCommunication class



Left: before refactoring, Right: after refactoring
Figure 2: Lack of Tight Class Cohesion for the courses microservice

As is visible in figure 1, the metrics for LTCC (Lack of Tight Class Cohesion) and LCOM (Lack of Cohesion Of Methods) are in the red and orange respectively, which means these values are (very) high. In figure 2 it is indicated that the percentage of classes with a very high LTCC, depicted in red in the pie chart, has gone down, while the other values have increased slightly.

Method: *createlecture,*
The method *createLecture* has a MCC of 5, but most importantly, a very-high coupling according to our CodeMR reports. This is not good, as we want the coupling to be as low as possible. Very high coupling means in this case that the CBO (Coupling Between Objects) is over 30. To try and get this metric down, we need to adapt the method to be less dependent on other classes. The way we managed to do this is by extracting different methods to divide up the logic into smaller methods that have less coupling. For example, instead of letting the createLecture method apply the builder pattern differently when a lecture is for a first or second year class, we extracted a method to take care of this, and we call this method from createLecture. After refactoring, the coupling went from very high (>30 CBO) to low-medium (CBO > 10 & CBO <= 20). Also, the McCabe's Cyclomatic Complexity went down from 5 to 3, and the number of methods called went from 20 to 9. These improvements are also visible in figure 3 and 4.

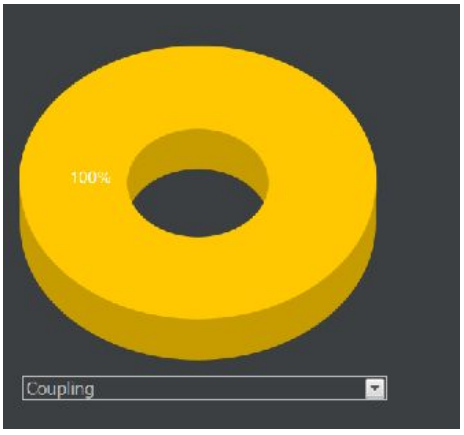
	Complexity	Coupling	Size	Lack of Cohesion	MCC	#MC
sem58						
oject.op29sem58.courses	low	low	low	low		
CoursesController	low-medium	medium-high	low-medium	medium-high		
createLecture(String, int, LectureInfo): ResponseEnt	low	very-high	low	low	5	20

	Complexity	Coupling	Size	Lack of Cohesion	MCC	#MC
sem58						
oject.op29sem58.courses	low	low	low	low		
CoursesController	low-medium	medium-high	low-medium	medium-high		
createLecture(String, int, LectureInfo): ResponseEnt	low	medium-high	low	low	3	9

Figure 3 and 4: the improved software metrics (MCC: McCabe's Cyclomatic Complexity, #MC: number of methods called)
The upper image shows the metrics before refactoring and the lower image shows the metrics after the refactoring.

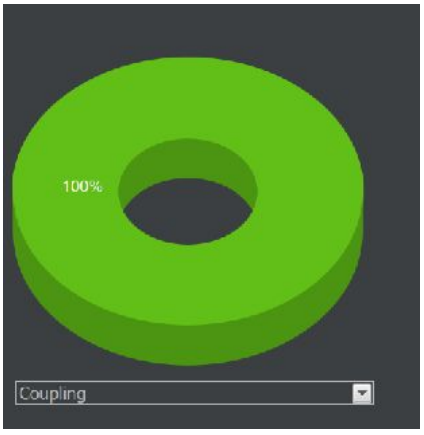
Room Scheduler

Class 1: *ServerCommunication*
This class has medium-high coupling according to CodeMR, what this means is that the *RoomSlotCommunication* and *RoomScheduleCommunication* that extend this class are affected heavily by changes made in the *ServerCommunication*. We want to aim for a low level of coupling because this gives more flexibility when we want to make changes to one of the two classes. Check the figures below for the result generated by CodeMR.



Name	Complexity	Coupling	Size	Lack of Cohesion
op29-sem58				
roomscheduler.communication	low	low	low	low
ServerCommunication	low	medium-high	low	low-medium

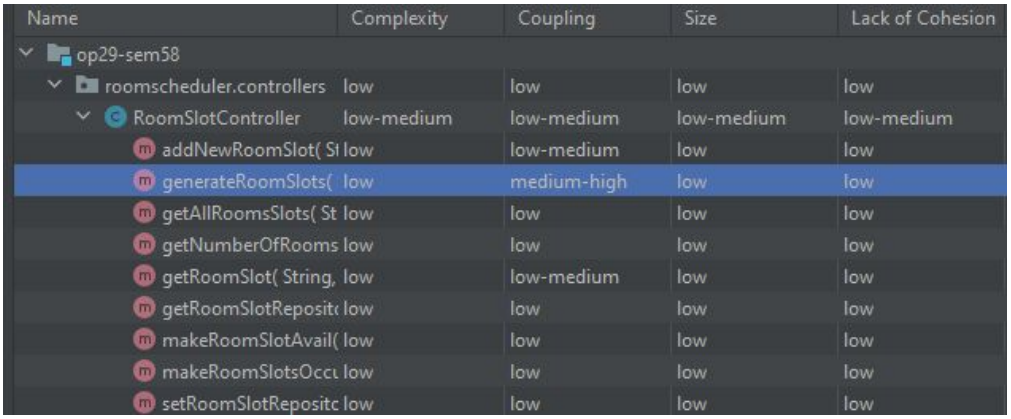
In our particular case, this was caused because the *ServerCommunication* had new objects created in almost every method, how we solved this was to refactor it so that the methods that were used from this extended class took those objects as parameters instead. This gave the classes *RoomSlotCommunication* and *RoomScheduleCommunication* more control when using methods in the *ServerCommunication* class. After doing so the class *ServerCommunication* went from medium-high coupling to low-medium coupling. Check the figures below for the result generated by CodeMR.



Name	Complexity	Coupling	Size	Lack of Cohesion
op29-sem58				
roomscheduler.communication	low	low	low	low
ServerCommunication	low	low-medium	low	low-medium

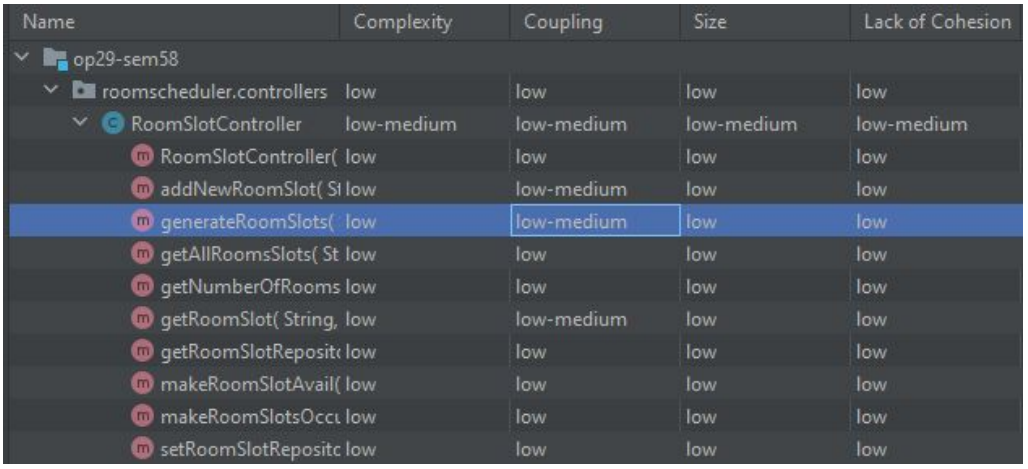
Method 1: *generateRoomSlots*

As for the method that we looked to refactor, in the Room Scheduler microservice we had a method to generate room slots that also had a medium-high level of coupling. For methods we wanted to aim for as loose coupling as possible. The reason for this is because methods that are loosely coupled have more flexibility to be used by itself. Check the images below for the results generated by CodeMR before refactoring.



Name	Complexity	Coupling	Size	Lack of Cohesion
op29-sem58				
roomscheduler.controllers	low	low	low	low
RoomSlotController	low-medium	low-medium	low-medium	low-medium
addNewRoomSlot(St	low	low-medium	low	low
generateRoomSlots(	low	medium-high	low	low
getAllRoomsSlots(St	low	low	low	low
getNumberOfRooms	low	low	low	low
getRoomSlot(String,	low	low-medium	low	low
getRoomSlotReposit	low	low	low	low
makeRoomSlotAvail(	low	low	low	low
makeRoomSlotsOcc	low	low	low	low
setRoomSlotReposit	low	low	low	low

In this case, it was caused because the method was calling methods from another class directly via the method. This meant that given that method was changed, it would affect the *generateRoomSlots* method. We solved this by creating a new void method called *makeRules* that would initialize these attributes and then calling this method inside the *generateRoomSlots* method. As you can see in the image below, this reduces the coupling of the method from medium-high to low-medium.

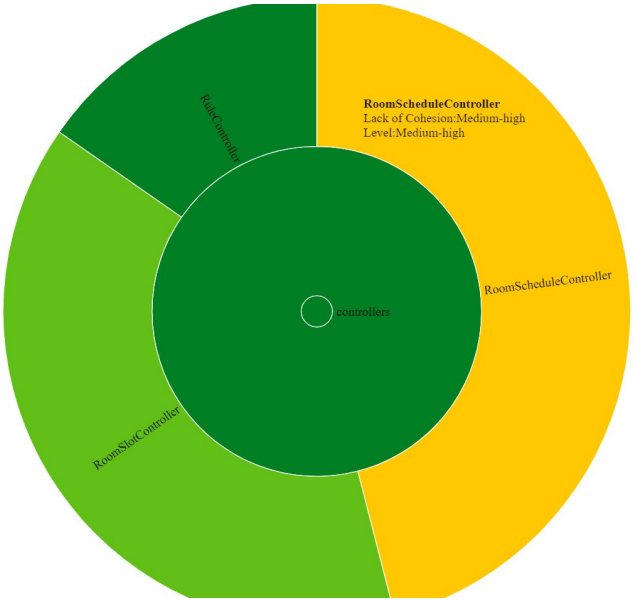


Name	Complexity	Coupling	Size	Lack of Cohesion
op29-sem58				
roomscheduler.controllers	low	low	low	low
RoomSlotController	low-medium	low-medium	low-medium	low-medium
RoomSlotController(	low	low	low	low
addNewRoomSlot(St	low	low-medium	low	low
generateRoomSlots(	low	low-medium	low	low
getAllRoomsSlots(St	low	low	low	low
getNumberOfRooms	low	low	low	low
getRoomSlot(String,	low	low-medium	low	low
getRoomSlotReposit	low	low	low	low
makeRoomSlotAvail(	low	low	low	low
makeRoomSlotsOcc	low	low	low	low
setRoomSlotReposit	low	low	low	low

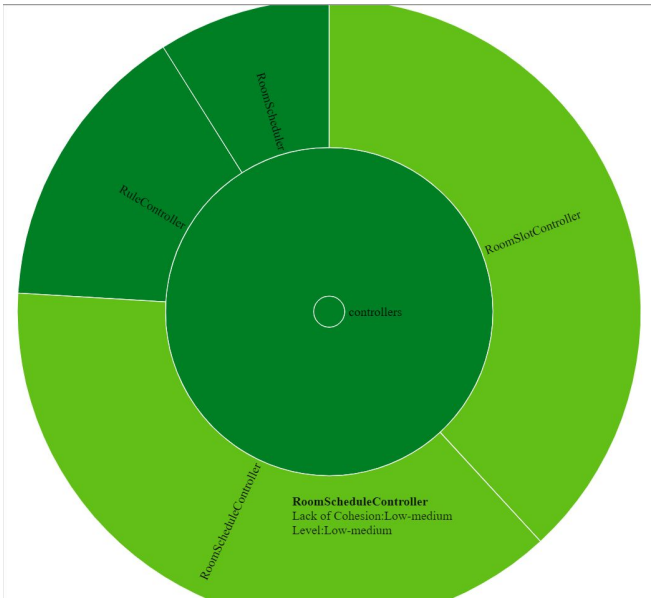
Class 2: *RoomScheduleController*

This class had a high lack of cohesion (LCohesion), as shown in the CodeMR report below (2.1). This measure indicates how well the methods of a class are related to each other. What is desired to be achieved is high cohesion, since this would entail numerous software quality aspects, such as reliability, reusability, robustness and understandability. On the other hand, low cohesion brings negative aspects, including difficulty in maintaining and testing the source code.

Lack of cohesion usually implies that the class has more responsibilities than it should have and a way to resolve this problem is to split the class into two or more classes. This is what we did in this particular situation for the *RoomScheduleController* class. We moved the *findSuitable* method in a new class called *RoomScheduler*. The effect of this operation can be seen in the figure below(2.2).



(2.1)Before refactoring



(2.2)After refactoring

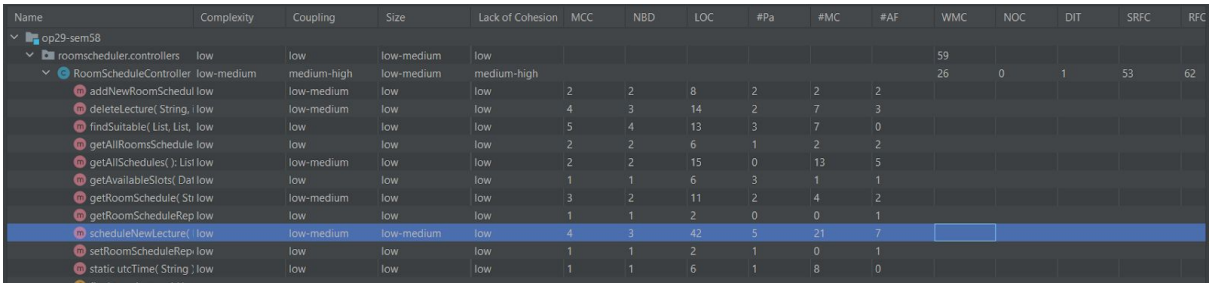
Method 2: *scheduleNewLecture*

The problem we identified with this method was mainly the number of method calls it makes (#MC). The measures of this attribute can be seen in the figure below (3.1). This method had many calls for methods belonging to other classes. This caused 2 problems:

a) The method was not sufficiently readable
b) It caused a high coupling (see 3.1) for the method, which also affected the coupling of the entire class. This is something we wanted to avoid, since high coupling means that a change in one place can have unknown effects in other places. It also adds complexity to the code, due to the intertwined relationships among classes.

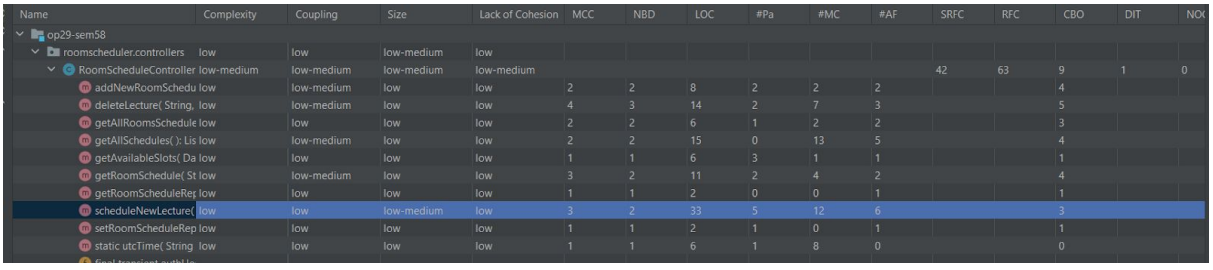
We solved this problem by performing the 'extract method' refactoring operation. In particular, the lines that had a common responsibility in the method (namely saving the entries for the scheduled lecture in the database and marking the corresponding room slots as occupied) were extracted and formed a new method (*scheduleLecture*). This method was also moved to the newly created class *RoomScheduler* to lower the coupling of the *RoomScheduleController* class. The metrics after performing the above-mentioned refactoring operations can be seen in (3.2)

(3.1)Before refactoring



Name	Complexity	Coupling	Size	Lack of Cohesion	MCC	NBD	LOC	#Pa	#MC	#AF	WMC	NOC	DIT	SRFC	RFC
op29-sem58															
roomscheduler.controllers	low	low	low-medium	low							59				
RoomScheduleController	low-medium	medium-high	low-medium	medium-high							26	0	1	53	62
addNewRoomSchedul	low	low-medium	low	low	2	2	8	2	2	2					
deleteLecture(String, i	low	low-medium	low	low	4	3	14	2	7	3					
findSuitable(List, List,	low	low	low	low	5	4	13	3	7	0					
getAllRoomsSchedule	low	low	low	low	2	2	6	1	2	2					
getAllSchedules( ): List	low	low-medium	low	low	2	2	15	0	13	5					
getAvailableSlots(Dat	low	low	low	low	1	1	6	3	1	1					
getRoomSchedule(Str	low	low-medium	low	low	3	2	11	2	4	2					
getRoomScheduleRep	low	low	low	low	1	1	2	0	0	1					
scheduleNewLecture(	low	low-medium	low-medium	low	4	3	42	5	21	7					
setRoomScheduleRep	low	low	low	low	1	1	2	1	0	1					
static utcTime(String)	low	low	low	low	1	1	6	1	8	0					

(3.2)After refactoring

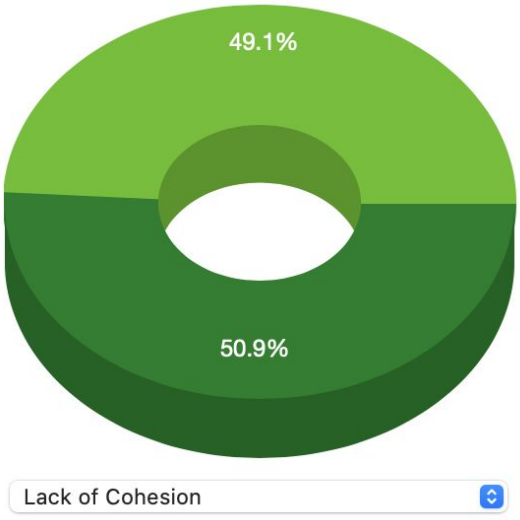


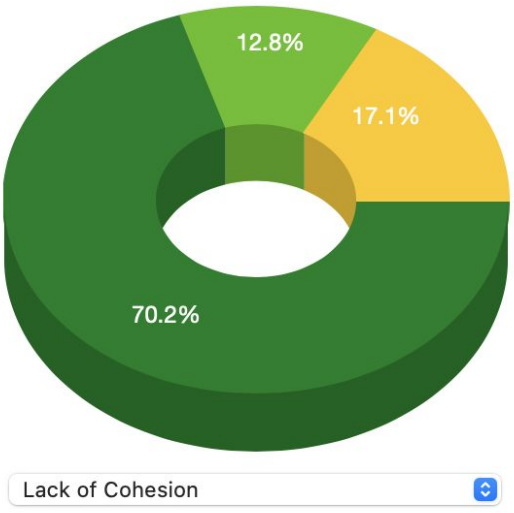
Name	Complexity	Coupling	Size	Lack of Cohesion	MCC	NBD	LOC	#Pa	#MC	#AF	SRFC	RFC	CBO	DIT	NOC
op29-sem58															
roomscheduler.controllers	low	low	low-medium	low											
RoomScheduleController	low-medium	low-medium	low-medium	low-medium							42	63	9	1	0
addNewRoomSchedu	low	low-medium	low	low	2	2	8	2	2	2			4		
deleteLecture(String,	low	low-medium	low	low	4	3	14	2	7	3			5		
getAllRoomsSchedule	low	low	low	low	2	2	6	1	2	2			3		
getAllSchedules( ): Lis	low	low-medium	low	low	2	2	15	0	13	5			4		
getAvailableSlots(Da	low	low	low	low	1	1	6	3	1	1			1		
getRoomSchedule(St	low	low-medium	low	low	3	2	11	2	4	2			4		
getRoomScheduleRep	low	low	low	low	1	1	2	0	0	1			1		
scheduleNewLecture(	low	low	low-medium	low	3	2	33	5	12	6			3		
setRoomScheduleRep	low	low	low	low	1	1	2	1	0	1			1		
static utcTime(String	low	low	low	low	1	1	6	1	8	0			0		

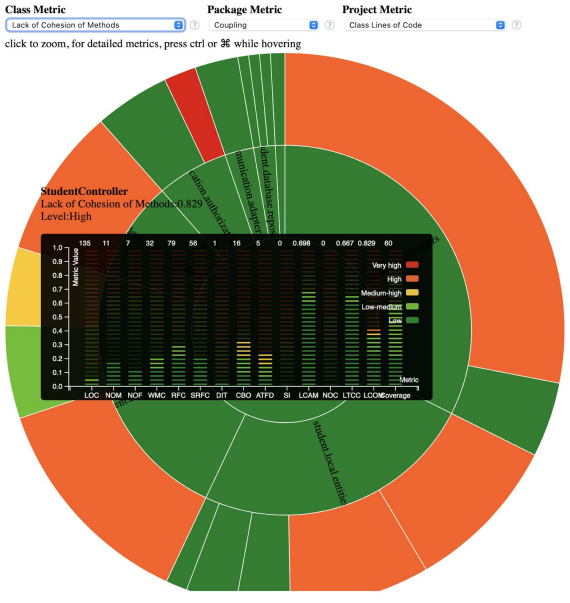
Student

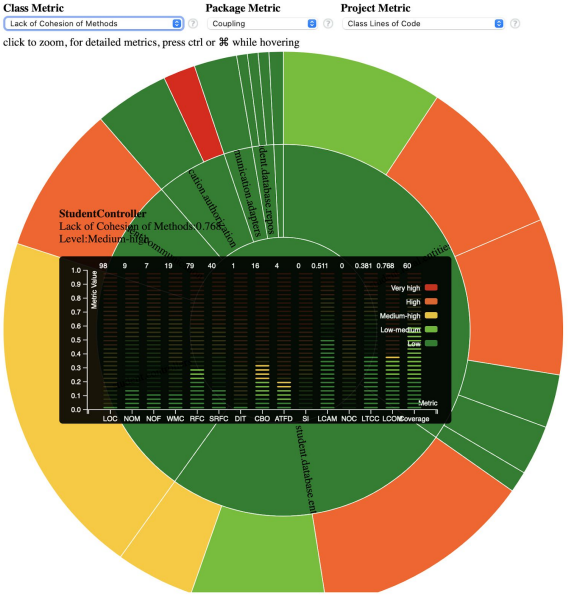
Class: *StudentController*
We decided upon refactoring this class, because we saw that there was a low lack of cohesion. The class also was pretty big and we really wanted to improve these two metrics. We saw a couple of methods which did not belong in the student controller and moved them to other classes. We decided to move, the method related to retrieving student enrollments to the student class. As this to us sounded more logical as it belonged to a student. We also moved a merge method to *LectureDetails* as it is for merging two lists of *LectureDetails* and we also moved both *getMyLectures* and *getCourseName* to a new class we made named *LectureManager*. This class is related to getting lectures for a student and includes the refactored *getMyLectures* method splitted up into *getMyLectures*, *getOnlineLectures* and *getCampusLectures*. This helped us in decreasing the size of the student controller, making it more organized and also helps with the cohesion.

Cohesion before: after:



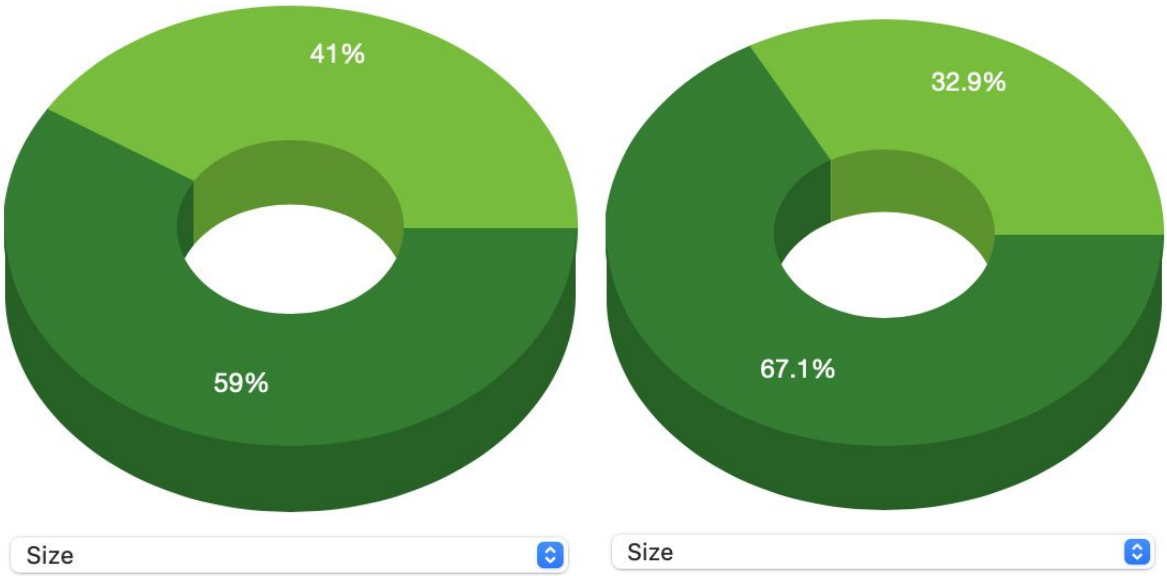






As you can see the cohesion improved, however because we moved two methods to two different classes which both only included getters/setters. The Cohesion of these two classes decreased, we still believe that this is an overall improvement from the start.

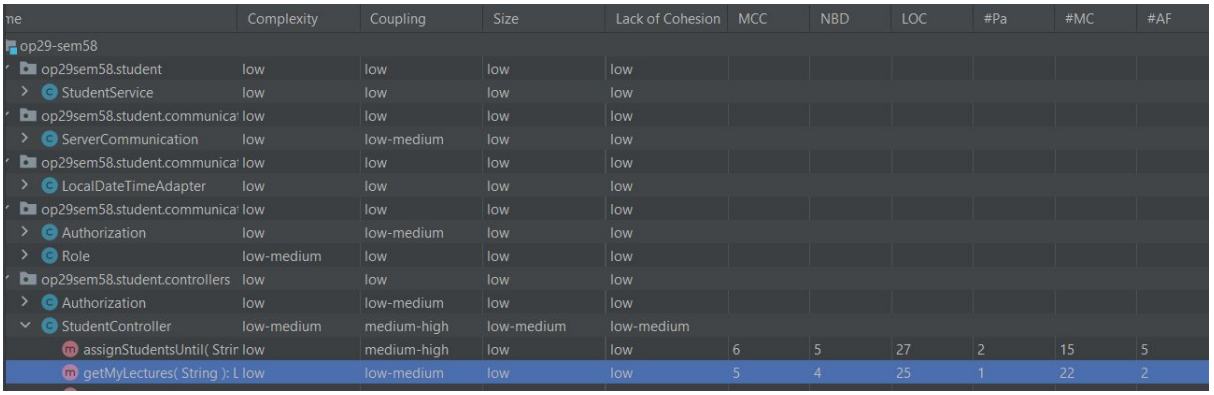
Size before:



So here you can see a major improvement, we went from 59% to 67.1% in regards to low class size. Which is great, because that is exactly what we wanted. Also logical from the changes we made.

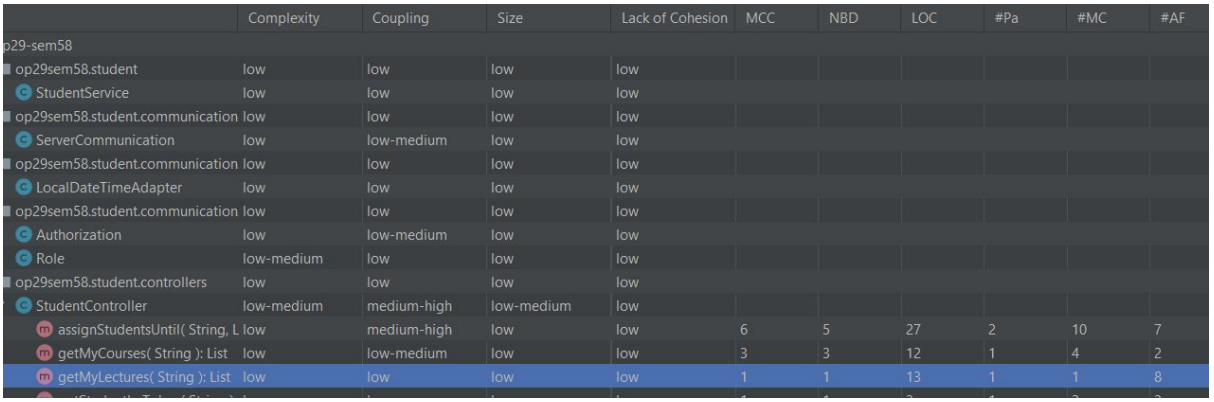
Method: *getMyLectures*

When deciding on the method that needed to change, we mainly focussed on the complexity as that is important for the readability and thus maintainability of the code. We noticed that one of the methods, *getMyLectures,* had a bad complexity. In the selected row in the image below shows the metrics for this method. The column MCC (McCabe's Cyclomatic Complexity) shows this complexity. We decided to choose this method over *assignStudentsUntil*, because this refactor paired well together with the class refactor.



me	Complexity	Coupling	Size	Lack of Cohesion	MCC	NBD	LOC	#Pa	#MC	#AF
op29-sem58										
op29sem58.student	low	low	low	low						
StudentService	low	low	low	low						
op29sem58.student.communica	low	low	low	low						
ServerCommunication	low	low-medium	low	low						
op29sem58.student.communica	low	low	low	low						
LocalDateTimeAdapter	low	low	low	low						
op29sem58.student.communica	low	low	low	low						
Authorization	low	low-medium	low	low						
Role	low-medium	low	low	low						
op29sem58.student.controllers	low	low	low	low						
Authorization	low	low-medium	low	low						
StudentController	low-medium	medium-high	low-medium	low-medium						
assignStudentsUntil(Strir	low	medium-high	low	low	6	5	27	2	15	5
getMyLectures(String): L	low	low-medium	low	low	5	4	25	1	22	2

To solve the complexity of this method, we split the method into several smaller methods and moved it to a new class that manages everything regarding getting lectures for a specific student: *LectureManager*. Since these methods are closely related to each other, this new class had a good score on the cohesion metric while also fixing the complexity issue we had in the original method to begin with. In the image below you can see how the complexity metric was improved from 22 to 1 while also improving the other metrics shown in the image.



	Complexity	Coupling	Size	Lack of Cohesion	MCC	NBD	LOC	#Pa	#MC	#AF
p29-sem58										
op29sem58.student	low	low	low	low						
StudentService	low	low	low	low						
op29sem58.student.communication	low	low	low	low						
ServerCommunication	low	low-medium	low	low						
op29sem58.student.communication	low	low	low	low						
LocalDateTimeAdapter	low	low	low	low						
op29sem58.student.communication	low	low	low	low						
Authorization	low	low-medium	low	low						
Role	low-medium	low	low	low						
op29sem58.student.controllers	low	low	low	low						
StudentController	low-medium	medium-high	low-medium	low						
assignStudentsUntil(String, L	low	medium-high	low	low	6	5	27	2	10	7
getMyCourses(String): List	low	low-medium	low	low	3	3	12	1	4	2
getMyLectures(String): List	low	low	low	low	1	1	13	1	1	8