# Class Refactoring:

The initial report generated by CodeMR was surprisingly good. With less than 20% of the classes reaching “medium-high” level on the four main quality attributes, complexity, coupling, lack of cohesion and size. It was therefore, a surprisingly challenging task to further improve those. In order to achieve this task we focused on the classes whose metrics lowered our project’s average the most. These classes were: LectureScheduleService found in database module, SysInteractController class found in the gateway authentication module, the Generator class in the schedule generate module, and the DatabaseService classes which both are present in the viewer and in the rules module.

## LectureScheduleService in Database module:

Initially on the codeMR report, LeactureScheduleService class had the worst code quality. The attributes that could be significantly improved were Complexity, Coupling and Lack of Cohesion. The metrics contributing to these attributes are CBO (11), RFC (45), SRF (29), WMC (23), LCOM (0.708) and LCAM (0.711). The main issue with this class was that it contained many methods and many different dependencies. In order to improve the quality of this class we have decided to apply Move Method Refactoring. Which we used to move a specific method (getStudentsInLecture) into a different Service class (UserService) that was already using the dependencies used by the moved method. This allowed us to greatly reduce the coupling of the target class without hurting the quality of the new host class, as pictured on the figure A in the Appendix. The overall refactoring of this class has resulted in the improvement of the before specified quality metrics: CBO from 11 to 9, RFC from 45 to 34, SRF from 29 to 26, WMC from 23 to 20, LCOM from 0.708 to 0.571, LCAM from 0.711 to 0.656, figure B in the Appendix.

## DatabaseService in Viewer module:

The initial CodeMR report for the databaService of the viewer module had issues with coupling, which can be seen from the figure C. After checking the code, we noticed that this was mainly caused by the methods using the webclient and many chained methods which most likely return different object types one after another. This was done by most of the methods of the class, therefore, a helper method is created to reduce length and avoid repetition. In order to reduce coupling, we added a method called “ReturnList" which takes an uri and an id as input and returns a list of objects using the webclient and puts the method in a separate class. *Coupling* decreased from “medium-high” to “low” and *lack of cohesion* and *size* from “low-medium” to “low”.

SysInteractController in Gateway-Authentication module:

SysInteractController is a class that has had a lot of dependencies and inconsistent/bad code quality. This class is the main controller of the gateway to the system therefore it was expected to have problems with coupling. Before we refactored the class, it had to parse authentication tokens and parse requests with specific entity parameters. These two alone

generated 10+ dependencies. To refactor this, we decided to use the Extract Method Refactoring type. We moved token parsing as well as specific entity parsing to a new class called TokenParsing. Making this new simple class, with good quality, figure D allowed us to improve the quality of the target class. This improvement can be quantified by comparing the metrics of: CBO from 13 to 7, SRFC from 30 to 21, WMC from 25 to 18, LCOM from 0.738 to 0.467, LAM from 0.472 to 0.469, LTCC from 0.417 to 0.25, figure E.

## DatabaseService in Rules module:

The DatabaseService class was another class that was recognized as having high coupling, this was due to a CBO of 13. We realized that the DatabaseService has such a high coupling because it uses the webclient and then returns values of many methods that are different entity objects. In order to reduce the coupling, at first we noticed a few dispensable methods so we removed them, this helped to simplify the problem at hand. In order to reduce the coupling even further we decided to use the move method refactoring strategy. For move method refactoring we first started by moving the webclient to a bean, created in RulesConfiguration. Furthermore, we decided that we should use a default header for all the requests, namely: “Content type: Application/JSON”, because previously whenever we created a new request we would have to set this header manually. Finally we made sure to split up the class into three classes relating to their return type or their operation on the database (these new classes can be found in the services package). The changes described above definitely helped improve the readability of the classes. Especially because the webclient is now simplified to require fewer methods to run.

## Generator in Schedule-generate module:

The generator class is responsible for assigning students and rooms to newly created lectures, and is one of the largest classes in our whole project due to the complexity of its methods (of the entire algorithm). As such, it is important that the class can be read and interpreted easily and by developers who might not have worked on the class initially. It’s also important that the methods be easy to test. Upon analysing the code and reading the generated metrics, we found that our class had a weighted method count (WMC) of 51, an access to foreign data (ATFD) of 4 and a Lack of tight class cohesion (LTCC) of 0.945. We decided that the WMC made the class too big, and the LTCC was a sign that we had many methods that would better fit a util class, as static methods. So our goals were to reduce the LTCC under 0.8 at least, because that seemed like an achievable goal, and WMC to under 40. In order to reduce these values, we made use of the extract method refactoring type. To do this we created a class named Util, and moved any methods which did not change the attributes of the Generator class into this Util class. Doing this allowed us to remove a lot of clutter from the Generator class, especially due to the fact that as we performed method refactoring on the methods in the class as well, there were even more methods being made, which also needed to be moved into Util. As well as reducing the WMC to 31, refactoring in this manner also reduced the ATFD from 4 to 3 as we were able to move a call to the API controller from Generator into Util, which is ideal as the content of the Generator class should primarily be the logic of the scheduling algorithm, not the details about how to communicate with the other modules. We also greatly reduced the

LTCC, from 0.945 to 0.76. The LTCC is an indicator of the cohesion between public methods, and reducing this makes the class much more robust.

## Method refactoring:

After computing and reviewing the code metrics at a method level, we found several methods in the Generator class of the scheduleGenerate module whose metrics were unsatisfactory. While there were methods in other classes which had room for improvement as well, we decided to focus on methods in the Generator class as these methods form the backbone of the scheduling system. Some of them were also presenting problems when it came to testing, because they were too big, long, and complicated. Since the priorities of the client may change over time, the way we schedule lectures may have to change to suit their needs, and as such it is imperative that the methods in Generator are both readable for developers who have not worked with it before, and can be easily altered to match the clients' needs. We've also noticed during testing that the methods herein proved to be difficult to test, because they performed a lot of different actions at once. The methods we worked on were: **scheduling**, **scheduleGenerate** ,**findRoom**, **isFree** and **getEarliestTime**. All in the **Generator** class.

We used extract method refactoring to reduce the number of method calls and the nested block depth of all methods we worked on. Having a large number of method calls indicated that our methods were performing many different tasks at once, which makes the purpose of each method more ambiguous. Not all method calls are necessarily problematic however, some method calls, such as getters, are inevitable and as such the **number of method calls** for the scheduling method is still relatively high. It's an unavoidable reality that this algorithm has a lot of pieces highly dependent on each other, and the usage of the Timestamp and Calendar class makes us bound to use a lot of method calls. A high **nested block depth** is also an indicator of a method being too complex; trying to read a method with four levels of indentation can become confusing, and as such we used extract method refactoring in order to simplify the nested for loops into several more simple methods, each performing a simple function. The **findRoom**, **isFree** and **getEarliestTime** methods all had a very high **number of parameters**, which makes the Javadoc less readable, and was a sign that we weren't setting enough objects as class parameters, especially since many of these parameters were being repeated by many methods. To solve this issue, we added these parameters as class attributes. Since these parameters were commonly used, and were being passed between many different methods, this reduced the number of parameters for many methods, and greatly reduced the amount of redundant information that a new developer would have to read. For all of these changes our goal was to reduce **nested block depth** to a maximum of 3 (three seemed readable and reasonable), and for **number of methods called** we were aiming for around 6, but in some cases it did not make sense to keep simplifying the code (the whole method was already doing a single, well-defined thing) and we ended up having some exceptions to the rule. Here were the improvements:

---

*[**method name**] - old **NBL** value -> new **NBL** value | old **NMC** -> new **NMC***

| | | |
|---|---|---|
| [scheduling] - | 8 -> 1 | \| 62 -> 3, 17, 5, 8 (decomposed) |
| [scheduleGenerate]- | | \| 11 -> 6 |
| [findRoom]- | 4 -> 3 | \| 11 -> 6 |
| [isFree]- | | \| 11 -> 8 |
| [getEarliestTime]- | | \| 13 -> 6 |

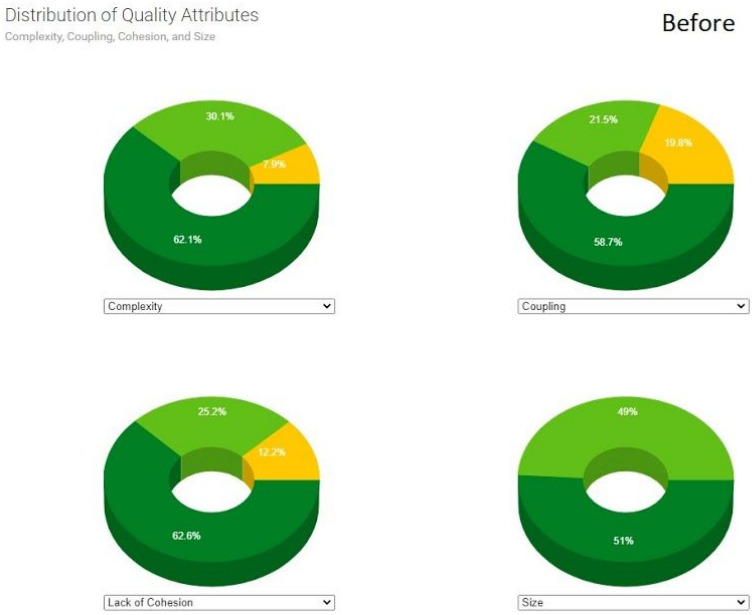
**number of methods called** (**NOMC**) improvements:

| | Old **NOMC** value -> new **NOMC** value |
|---|---|
| [findRoom]- | 6 -> 2 |
| [isFree]- | 6 -> 4 |
| [getEarliestTime]- | 5 -> 2 |

(if a metric is not specified that's because the value was already acceptable)

---

Before




After


The overall project improvement

# Appendix:

Figure A:


Figure B:


Figure C:

Before:


After:


Figure D:


Figure E: