SEM Assignment 2

Group 14B

January 2022

Chapter 1

Selected Metrics

The code metric tools chosen were **CodeMR** and **MetricsTree**. CodeMR was initially used to find the problematic classes – classes that were frequently marked as severe by many of the criteria. Those classes were then further inspected using MetricsTree, which could give more detailed insight of metrics at the class level and method level. Below, we will motivate our choices of the classes and methods we chose to refactor. In **Chapter 2**, we will present the metrics for classes before and after refactoring and outline the refactoring operations performed. In **Chapter 3**, we will do the equivalent for the methods refactored. In **Chapter 4**, we list all the references used in this document.

Number Of Attributes Or Methods

We have categorized the **Number of Attributes (NOA)** and **Number of Methods (NOM)** metrics together since we found that the values of these metrics were positively correlated for classes in our code base. A class with many instance variables is indicative of duplicate code in the class and thus requires closer inspection or reduction [6].

Firstly, a large number of methods could indicate that the class has too many responsibilities, which violates the *Single-Responsibility Principle* [1] and hinders testing of the class. Secondly, it may mean that we have dispensable methods in the code base, and it would be inefficient to allocate resources for maintaining them. In both cases, we should refactor the classes that score significantly above the regular range of MetricsTree.

Access To Foreign Data

Accesses to Foreign Data (ATFD) was identified as extremely severe for one of the classes, namely Notification Service, which had a value of 9 (4 in excess). This was the result of its heavy reliance on communication to external services, such as S3Bucket, PDFBox and SendGrid.

This metric indicates that it may have too many external dependencies. More importantly, this suggests that the class has too many responsibilities, which violates the *Single-Responsibility Principle* [1].

Cyclomatic Complexity

Many methods in the service classes of the Submission microservice was found to have high **McCabe Cyclomatic Complexity (MCC)**. Though the recommended range was [0, 3), we decided that this needs to be reduced for all service methods that had MCC > 7, since we wanted to keep the methods below high complexity [7].

We have chosen to focus on the MCC metric since reducing cyclomatic complexity will facilitate the process of achieving high branch coverage in testing. Moreover, having a high cyclomatic complexity also reduces the readability of the methods, which may hinder code maintainability.

Lines Of Code

Similarly to outlined above, several service methods have a high number of **Lines of Code (LOC)**. On average, service methods in the Submission service and Notification service had exceeded the recommended regular range of [0, 11) by a factor of 5.

We have selected this metric due to its level of severity in many of the service methods. Moreover, we believe that this results in blob code that will be hardly maintainable in the future. It will also be more difficult to understand for other developers who would like to contribute to this project.

Coupling Between Objects

We believe that high **Coupling Between Objects (COB)** is a metric that should be improved since it is a predictor of software failure [2]. It was shown that a coupling level of 9 is optimal, thus we want to make sure that most methods will have a value below or equal to 9 [2].

By inspecting the code base, we realised that this was the result of some methods requesting data from repositories and other microservices. Not only does high coupling reduce maintainability, but also increases the complexity of test cases that needed to test the method. This should be avoided since it may discourage testing of the system.

Number Of Parameters

The **Number of Parameters (NOP)** should be reduced below 5 as recommended [4], though the MetricsTree indicated a range of [0, 3). We believe that the number of parameters should be reduced as a greater number of parameters means that it is difficult to write a method call as the developer must remember the order, meaning, and type of each parameter. This could easily lead to client faults for users of our system and other microservices.

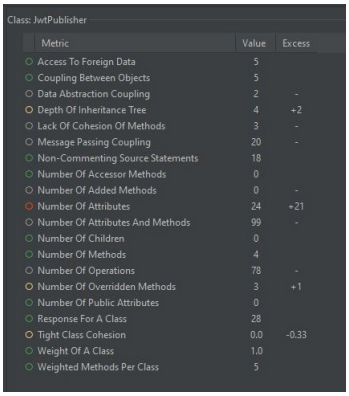
To enhance the usability of our system, we will improve this metric for methods expecting more than 5 parameters. [4]

Chapter 2

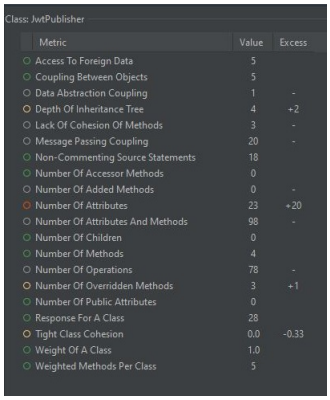
Class Refactoring

2.1 JwtPublisher

Metrics



(a) JwtPublisher metrics before refactoring



(b) JwtPublisher metrics after refactoring

Figure 2.1: Refactoring of JwtPublisher

Code Metric	Before Refactoring	After Refactoring
Number of Attributes	24	23

Table 2.1: Code metrics before and after refactoring of `JwtPublisher`

Motivation

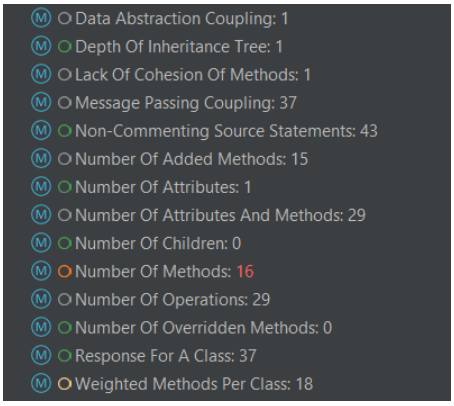
The `JwtPublisher` provides functionality to generate JWT which is one of the most important components of any system: security. As users rely on our security techniques to keep their data private, we want to improve the maintainability of the class by increasing cohesion and reducing number of parameters to only use the necessary ones. This should allow for faster security checks as well as increase code readability.

Operations to refactor

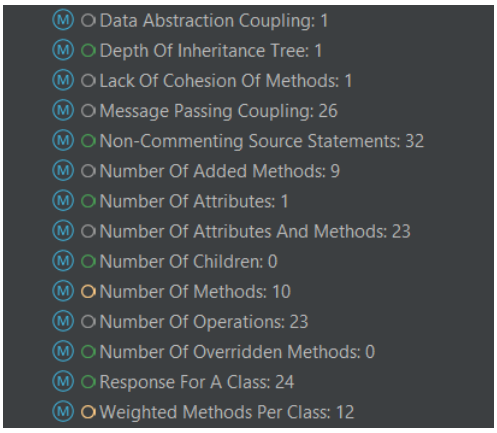
MetricsTree indicated that `JwtPublisher` class contained too many attributes. After thorough investigation of the `JwtPublisher` class, we came to a conclusion, that this is because our class extends from many other classes which are core components of Spring Framework. However, we also found a redundant attribute, which was increasing complexity as well as consuming more memory, and was thus deleted.

2.2 Course Controller

Metrics

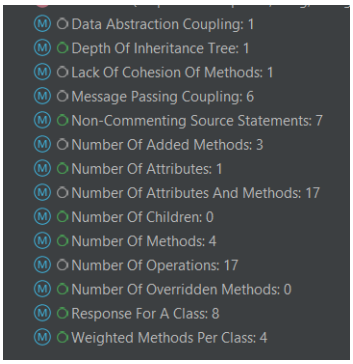


(a) CourseController metrics before refactoring

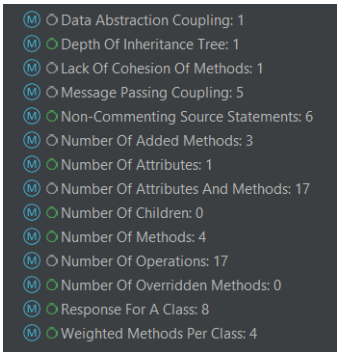


(b) CourseController metrics after refactoring

Figure 2.2: Refactoring of Course Controller



(a) AdditionController metrics



(b) RemovalController metrics

Figure 2.3: Classes that are created out of refactoring

Code Metric	Before Refactoring	After Refactoring
Number of Methods	16	10

Table 2.2: Code metrics before and after refactoring of `CourseController`

Classes Extracted	AdditionController	RemovalController
Number of Methods	4	4

Table 2.3: Code metrics of classes extracted from `CourseController`

Motivation

The `CourseController` class contains all publicly accessible endpoints which then are used to retrieve, alter, add and delete some resources. To improve testability of controller classes, thus we decided to reduce number of methods to at most 9 methods per class (including constructor), above the recommended range of [0, 7). The idea was to divide methods in logically cohesive categories based on their responsibility.

Operations to refactor

To reduce the number of methods, the `CourseController` class was divided into three different controller classes based on different categories: retrieving, adding, or removing a resource. Separating endpoints into their respective controller classes : `AdditionController` and `RemovalController` was the most efficient solution since now they are more cohesive and maintainable. After the refactoring, `CourseController` has only 9 methods (excluding constructor).

2.3 Notification Service

Metrics

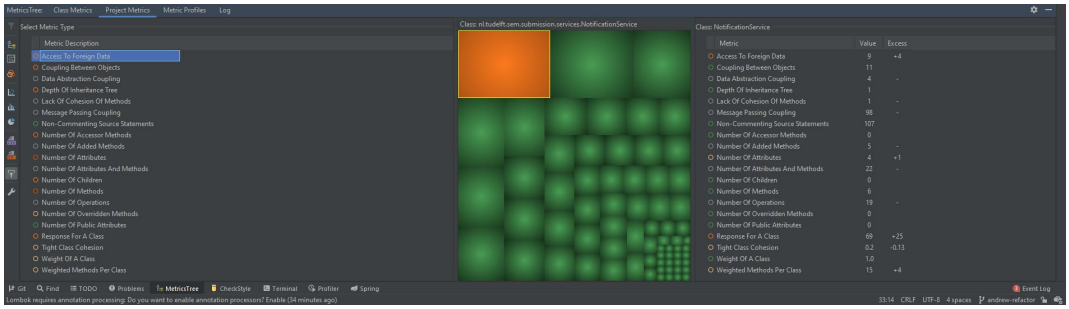


Figure 2.4: NotificationService metrics before refactoring

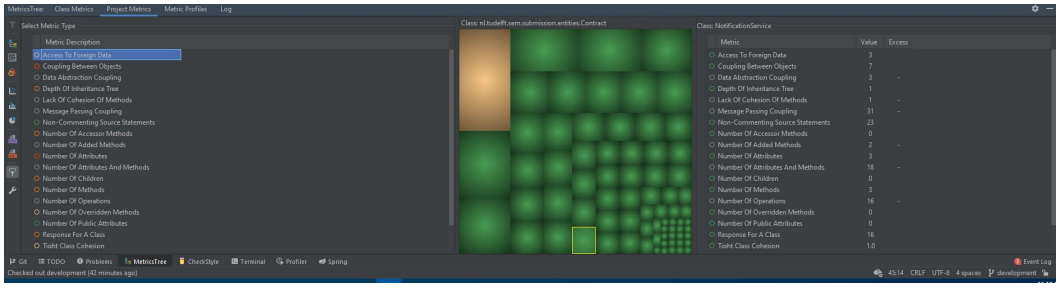


Figure 2.5: NotificationService metrics after refactoring

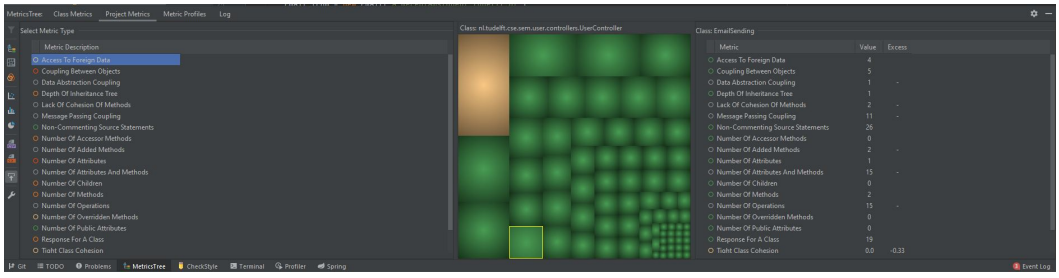


Figure 2.6: EmailSending metrics after refactoring

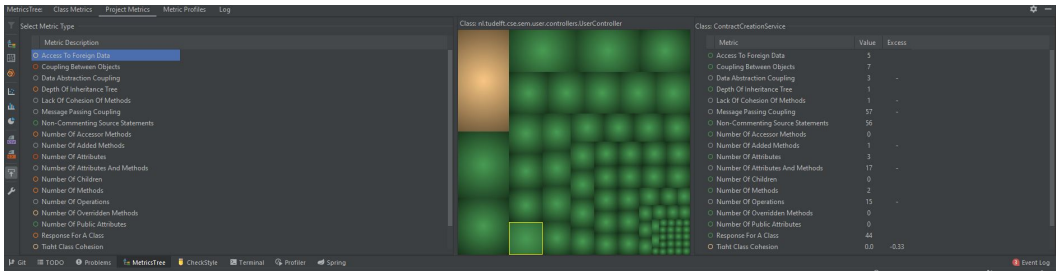


Figure 2.7: ContractCreationService metrics after refactoring

Code Metric	Before Refactoring	After Refactoring
Access to Foreign Data	9	3

Table 2.4: Code metrics of class `NotificationService`

Code Metric	Before Refactoring	After Refactoring
Access to Foreign Data	9	3

Table 2.5: Code metrics of class `EmailSending`

Code Metric	Before Refactoring	After Refactoring
Access to Foreign Data	9	5

Table 2.6: Code metrics of class `ContractCreationService`

Motivation

The `NotificationService` class is responsible for sending notifications to Students when they have been hired as a TA, which is a critical component of the system, as it completes the application process. After analyzing `NotificationService` class with MetricsTree, we found too many foreign data accesses. This was due to the class being responsible for generating the contract, for storing it in the correct database, and sending it via email. Clearly, this violates the *Single Responsibility Principle*, so we reduced foreign data accesses to at most 5 as well as to decrease coupling.

Operations to refactor

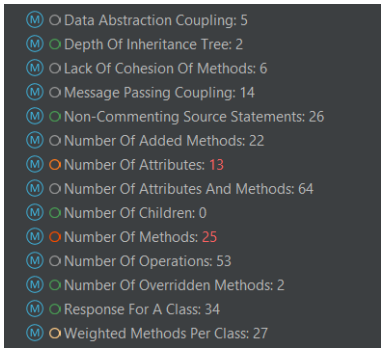
Our approach was to split up `NotificationService` into multiple services, each with it's own single responsibility. The resulting classes with corresponding responsibilities are:

- ContractCreationService is responsible for creating a .pdf document as well as saving it at the S3 bucket (AWS cloud storage).
- `NotificationService` is responsible for keeping the database in the correct state as well as for the logic, thus it can be called a load balancer or a gateway, it decides which class to call and what action to perform.
- `EmailSending` main task is to send email to correct student with relevant information.

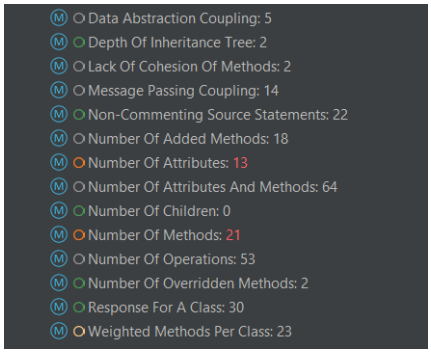
With higher cohesion and lowered access to foreign data, the code base is easier to read, refactor, test and extendable. Each class (mentioned above) adhere to the *Single Responsibility Principle*.

2.4 Student

Metrics



(a) Student metrics before refactoring



(b) Student metrics after refactoring

Figure 2.8: Refactoring of Student Class

Code Metric	Before Refactoring	After Refactoring
Number of Methods	25	21

Table 2.7: Code metrics of `Student`

Code Metric	Before Refactoring	After Refactoring
Number of Methods	13	9

Table 2.8: Code metrics of `Lecturer`

Code Metric	Before Refactoring	After Refactoring
Number of Methods	8	12

Table 2.9: Code metrics of `User`

Motivation

Duplicate code increase likeliness of errors and blob code. `Student` and `Lecturer` extend the same class. Using MetricsTree, we found that `Student` scored high on **Number Of Methods** and decided to decrease amount of methods for each class by at least 4, because we spotted exactly 4 redundant methods.

Operations to refactor

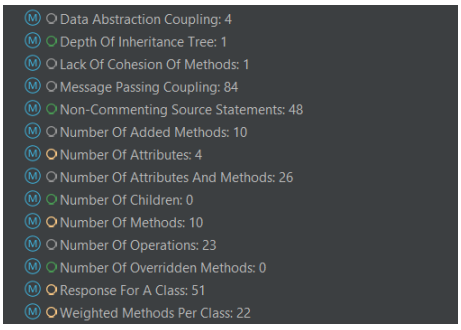
`Student` is an entity class inside the User Micro-Service. It has too many methods(25) and too many attributes (13), but all of them are critical in security or in database management. Therefore, we had limited options as to the methods or attributes that could be removed.

4 methods that are shared both by the `Student` and `Lecturer` class is carried to their parent class `User`. Hence, the number of methods for both the `Student` and `Lecturer` both decreased. Though the amount of methods for the `User` class increased, it was a reasonable trade-off as it increased code reusability.

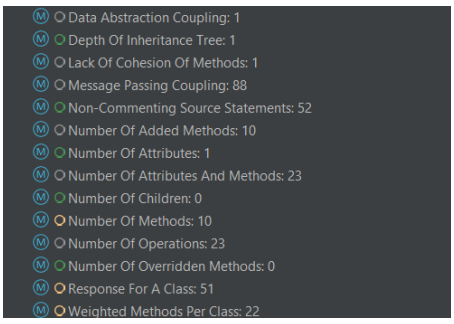
In an attempt to refactor, we tried to extract all fields related to `Rating` and create an object that would be used by `Student` class as an attribute. Unfortunately, it caused some various errors that was not easy to fix and thus led us deciding to only trying to reduce the number of methods that wouldn't cause such problem.

2.5 Course Communication

Metrics



(a) CourseCommunication metrics before refactoring



(b) CourseCommunication metrics after refactoring

Figure 2.9: Refactoring of Course Communication

Code Metric	Before Refactoring	After Refactoring
Number of Attributes	4	1

Table 2.10: Code metrics of `CourseCommunication`

Motivation

The `CourseCommunication` class had a high Number of Attributes, which could indicate low cohesion. This indicates a code smell since it is solely responsible for communication with the Course microservice.

We found that many fields were used by only a few methods. Furthermore, some fields, such as URLs were constants and should have been moved to `util`.

Operations to Refactor

Most of the attributes of `CourseCommunication` class, such as the URL for other Micro-Services, were used in only the generic GET and POST methods, so the attributes were removed as fields of the class.The fields were then created in the methods that used them directly. The Number of Attributes in the class was reduced to 1. Other values were either moved to `util` or provided independently to the methods that used them.

Chapter 3

Method Refactoring

3.1 Contract Creation

Metrics

Code Metric	Before Refactoring	After Refactoring
Condition Nesting Depth	1	0
Lines Of Code	79	46
Loop Nesting Depth	0	0
McCabe Cyclomatic Complexity	3	1
Number of Loops	0	0
Number of Parameters	6	2

Table 3.1: Code metrics before and after refactoring operations for `createContract` method

Code Metric	After Refactoring
Condition Nesting Depth	1
Lines Of Code	32
Loop Nesting Depth	0
McCabe Cyclomatic Complexity	3
Number of Loops	0
Number of Parameters	3

Table 3.2: Code metrics after refactoring for new method `saveContract`

Code Metric	After Refactoring
Condition Nesting Depth	0
Lines Of Code	5
Loop Nesting Depth	0
McCabe Cyclomatic Complexity	1
Number of Loops	0
Number of Parameters	1

Table 3.3: Code metrics after refactoring for new method `lineFormatter`

Motivation

We decided to refactor this method because it had many lines of code (79). This was leading to a hard to follow code and a high probability of error. We observed that we could make the method more reliable and testable if we extracted it into three methods. Besides, the high number of parameters made it quite challenging to be called, complicated and close to extensions, so this also needed to be changed.

Types of Operations

In order two reduce the **lines of code** two extractions were applied.

- First, saving the contract was extracted to a separate method, instead of being embedded in the `createContract` method.
- Secondly, a `lineFormatter` method was created. Instead of calling the stream.newline() method a bunch of times in a row, the `lineFormatter` method is called, which calls the stream.newline() method a specified number of times consecutively.

This brought the lines of code to 46 from 79, which is a great improvement.

In order to reduce the **number of parameters** used we introduced parameter objects. The `createContract` now accepts now an entire Submission object, instead of all the necessary details about the Submission, since the Submission object contains all these attributes. These attributes are extracted within the method by the use of getters. So we managed to achieve a reduction from 6 parameters to 2.

3.2 Create Submission

Metrics

Code Metric	Before Refactoring	After Refactoring
Condition Nesting Depth	1	0
Lines Of Code	53	27
Loop Nesting Depth	0	0
McCabe Cyclomatic Complexity	4	1
Number of Loops	0	0
Number of Parameters	5	4

Table 3.4: Code metrics before and after refactoring operations for `createSubmission`

Code Metric	After Refactoring
Condition Nesting Depth	1
Lines Of Code	16
Loop Nesting Depth	0
McCabe Cyclomatic Complexity	2
Number of Loops	0
Number of Parameters	2

Table 3.5: Code metrics after refactoring for new method `checkForDuplicate`

Code Metric	After Refactoring
Condition Nesting Depth	1
Lines Of Code	26
Loop Nesting Depth	0
McCabe Cyclomatic Complexity	2
Number of Loops	0
Number of Parameters	4

Table 3.6: Code metrics after refactoring for new method `checkSubmissionCredentials`

Code Metric	After Refactoring
Condition Nesting Depth	1
Lines Of Code	17
Loop Nesting Depth	0
McCabe Cyclomatic Complexity	2
Number of Loops	0
Number of Parameters	2

Table 3.7: Code metrics after refactoring for new method `checkSubmissionDeadline`

Motivation

Our motivation behind refactoring this method was that it had a high cyclomatic complexity and too many lines of code, thus, it was hard to understand and change. It is clear that this method included many checks for the validity of the submission made, which naturally could be extracted into different methods.

Types of Operation

The metrics we wanted to improve was the **Lines Of Code (LOC)** metric and the **McCabe Cyclomatic Complexity (MCC)**.

- The MCC metric was improved by extracting the checking for the validity of the submission. This included checking whether it is duplicate, the student had sufficient grade, and the date of submission to `checkForDuplicate`, `checkSubmissionCredentials`, and `checkSubmissionDeadline` respectively. By extracting these checks to their own independent methods, the MCC metric decreased from 4 to 1.
- The LOC metric of the method also decreased from 53 lines of code to 27 lines, with the extraction of the checks for validity of submission,
- The Number of Parameters was also reduced by 1 by taking `SubmissionCreateRequest` as a parameter rather than all its contents.

3.3 Accept Submission

Metrics

Code Metric	Before Refactoring	After Refactoring
Condition Nesting Depth	1	0
Lines Of Code	45	17
Loop Nesting Depth	0	0
McCabe Cyclomatic Complexity	7	1
Number of Loops	0	0
Number of Parameters	3	3

Table 3.8: Code metrics before and after refactoring operations for `acceptSubmission`

Code Metric	After Refactoring
Condition Nesting Depth	1
Lines Of Code	15
Loop Nesting Depth	0
McCabe Cyclomatic Complexity	2
Number of Loops	0
Number of Parameters	1

Table 3.9: Code metrics after refactoring for new method `findSubmission`

Code Metric	After Refactoring
Condition Nesting Depth	0
Lines Of Code	14
Loop Nesting Depth	0
McCabe Cyclomatic Complexity	1
Number of Loops	0
Number of Parameters	3

Table 3.10: Code metrics after refactoring for new method `undateAsTA`

Code Metric	After Refactoring
Condition Nesting Depth	1
Lines Of Code	20
Loop Nesting Depth	0
McCabe Cyclomatic Complexity	4
Number of Loops	0
Number of Parameters	3

Table 3.11: Code metrics after refactoring for new method `checkValidAccept`

Motivation

The method had a very high number of lines of code due to its communication to other microservices. Moreover, it performed actions that are not exclusive to marking a submission as accepted – for example, finding whether the submission exists. These parts can be extracted to new methods that can be used by other methods in the class. So, it was a hard to follow and test method, that had many tasks that needed a clear separation.

Types of Operation

The metrics we worked on for `acceptSubmission` was **McCabe Cyclomatic Complexity (MCC)** and **Lines of Code (LOC)**.

- To reduce the MCC metric, we separated the lines of code that checked whether the submission being accepted actually exists to a method called `findSubmission`. This method can be reused for many other methods in the `SubmissionService` class. Moreover, code responsible for communicating with other microservices to a method called `updateAsTa`. Finally to check the status and whether the lecturer is authorized, a new method called `checkValidAccept` was created. In the end we managed to get to a cyclomatic complexity of 1 from 7.
- The extraction of methods that communicate with other microservices and that can be reused consequently decreased the LOC metric of the method from 45 to 17 lines.

3.4 Send Email

Metrics

Code Metric	Before Refactoring	After Refactoring
Condition Nesting Depth	1	0
Lines Of Code	44	34
Loop Nesting Depth	0	0
McCabe Cyclomatic Complexity	5	4
Number of Loops	0	0
Number of Parameters	4	2

Table 3.12: Code metrics before and after refactoring operations for `sendEmail`

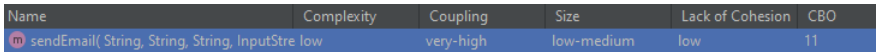


Figure 3.1: metrics of original method sendEmail via CodeMR before refactoring

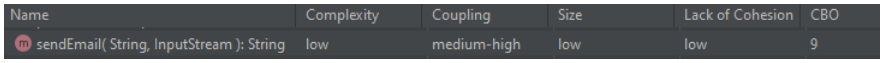


Figure 3.2: metrics of refactored method sendEmail via CodeMR after refactoring

Motivation

The main motivation to refactor this method was that it contain many unnecessary code, left from testing. So, the lines of code were too many and this made the method prone to errors and hard to read. Besides, we have observed that the number of parameters were also a problem, as it made the method too complicated.

Types of Operations

The metrics we worked on for `sendEmail` were primarily the **Number of Parameters**, **Lines of Code** and **McCabe Cyclomatic Complexity**. We thought it valuable to cut down the number of parameters as many were of the same type, String, and could therefore be merged together into a single String array named data which we accessed using their default assigned index value within the data array. With regards to the lines of code, we thought it important to remove any unnecessary comments and clutter as the method was messy which managed to resolve this problem. Finally, to reduce the high value of McCabe Cyclomatic Complexity, we removed an unnecessary try-catch block previously used for testing.

3.5 Retract Submission

Metrics

Code Metric	Before Refactoring	After Refactoring
Condition Nesting Depth	1	1
Lines Of Code	39	27
Loop Nesting Depth	0	0
McCabe Cyclomatic Complexity	5	4
Number of Loops	0	0
Number of Parameters	4	2

Table 3.13: Code metrics before and after refactoring operations for `retractSubmission`

Code Metric	After Refactoring
Condition Nesting Depth	0
Lines Of Code	19
Loop Nesting Depth	0
McCabe Cyclomatic Complexity	2
Number of Loops	0
Number of Parameters	3

Table 3.14: Code metrics after refactoring the new method `canRetractSubmission`

Code Metric	After Refactoring
Condition Nesting Depth	0
Lines Of Code	11
Loop Nesting Depth	0
McCabe Cyclomatic Complexity	1
Number of Loops	0
Number of Parameters	2

Table 3.15: Code metrics after refactoring the new method `deleteSubmission`

Code Metric	After Refactoring
Condition Nesting Depth	1
Lines Of Code	14
Loop Nesting Depth	0
McCabe Cyclomatic Complexity	2
Number of Loops	0
Number of Parameters	1

Table 3.16: Code metrics after refactoring the new method `getSubmission`

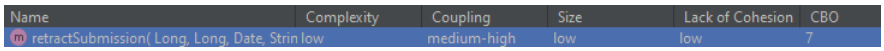


Figure 3.3: Metrics of original method retractSubmission via codeMR **before** refactoring

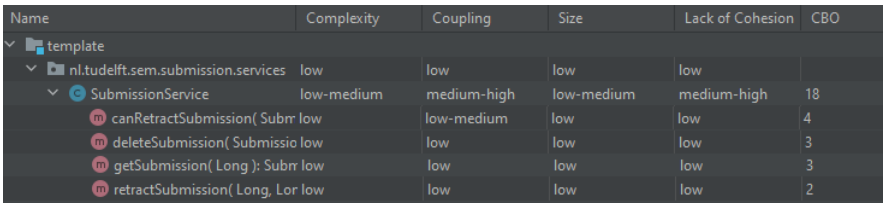


Figure 3.4: Metrics of refactored method retractSubmission via CodeMR **after** refactoring

Motivation

This method had a high coupling because it was communicating with both the database and the course microservice. Besides, there were too many lines, and a high cyclomatic complexity due to the number of checks that need to be done when a submission is retracted and that could have also been a problem in the long term (hard to understand the code, challenging to test it properly). So, we proceeded to fix all these problems with and extract method procedure.

Types of Operations

On `retractSubmission` we worked on **Coupling Between Object (CBO)**, **Lines Of Code (LOC)** and **Cyclomatic Complexity (CC)**. The method was improved by separating it into four methods using the extract method refactoring.

- The first method is called `canRetractSubmission` and it checks all the requirements that a submission should have in order to be eligible for retraction (more than three weeks until the start of the course, the submission has not been accepted).
- `deleteSubmission` deletes the submission from the database and removes the student from the candidate list of a course.
- `getSubmission` extracts a submission from the database based on a submission id.

So, the coupling has been reduced from medium-high to low, as the method has now a reduced number of dependencies. The lines of code, were also reduced from 39 to 27 and the cyclomatic complexity went down as it was also spread to four methods.

Chapter 4

References

1. https://www.baeldung.com/java-single-responsibility-principle
2. https://docs.microsoft.com/en-us/visualstudio/code-quality/code-metrics-class-coupling?view=vs-2022
3. https://pvs-studio.com/en/blog/terms/0086/
4. https://www.ndepend.com/docs/code-metrics#NbParameters
5. https://www.codemr.co.uk/case-reports/bde/html_report_b-all-libraries/htmlx/lbd/metricdescriptions.html
6. https://refactoring.guru/smells/large-class
7. https://www.brandonsavage.net/code-complexity-and-clean-code/