2022

TAHS – Assignment 2 Refactoring

GROUP 13A

AUTHORS:

- ANDY LI
- TANESHWAR PARANKUSAM
- MEHMET SÖZÜDÜZ
- MIHNEA TOADER
- JEGOR ZELENJAK
- JEROEN BASTENHOF

Table of Contents

Abbreviations

Abbreviation	Description
CBO	**C**oupling **B**etween **O**bjects Classes
LCAM	**L**ack of **C**ohesion **A**mong **M**ethods
LOC	**L**ines **o**f **C**ode
LTCC	**L**ack of **T**ight **C**lass **C**ohesion
MC	**M**ethod **C**ount
MLOC	**M**ethod **L**ines **o**f **C**ode
NBD	**N**ested **B**lock **D**epth
NOM	**N**umber **o**f **M**ethods

1 Analyzing code metrics

We have chosen to use the recommended code metric tool: *CodeMR. CodeMR* is a tool that is capable of statically analyzing the codebase based on a certain query configuration. The configuration of this query influences the severity of the results in the final report that is generated by *CodeMR*. As has been concluded in the study by Bigonha et al, thresholds of metrics are *"significantly effective in supporting the detection of bad smells"* and *"may be useful instruments in evaluating software quality"* (Bigonha, et al., 2019). Therefore, we as a group have collaboratively decided to tailor these thresholds to our needs by altering the default query parameters, see Figure 1.

Enable	Level	Quality Attribute	Condition
☑	low	Coupling	(CBO <= 5)
☑	low-medium	Coupling	(CBO > 5 AND CBO <= 10)
☑	medium-high	Coupling	(CBO > 10 AND CBO <= 20)
☑	high	Coupling	(CBO > 20 AND CBO <= 30)
☑	very-high	Coupling	(CBO > 30)
☑	low	Complexity	(WMC <= 20) OR (RFC <= 50) OR (DIT <= 1)
☑	low-medium	Complexity	(WMC > 20 AND WMC <= 50) OR (RFC > 50 AND RFC <= 100) OR (DIT > 1 AND DIT <= 3)
☑	medium-high	Complexity	(WMC > 50 AND WMC <= 101) OR (RFC > 100 AND RFC <= 150) OR (DIT > 3 AND DIT <= 10)
☑	high	Complexity	(WMC > 101 AND WMC <= 120) OR (RFC > 150 AND RFC <= 200) OR (DIT > 10 AND DIT <= 20)
☑	very-high	Complexity	(WMC > 120) OR (RFC > 200) OR (DIT > 20)
☑	low	Lack of Cohesion	(LCAM <= 0.6)
☑	low-medium	Lack of Cohesion	(LCAM > 0.6 AND LCAM <= 0.7)
☑	medium-high	Lack of Cohesion	(LCAM > 0.7 AND LCAM <= 0.8)
☑	high	Lack of Cohesion	(LCAM > 0.8 AND LCAM <= 0.9)
☑	very-high	Lack of Cohesion	(LCAM > 0.9)
☑	low	Size	(LOC <= 50) OR (NOM <= 20)
☑	low-medium	Size	(LOC > 50 AND LOC <= 300) OR (NOM > 20 AND NOM <= 30)
☑	medium-high	Size	(LOC > 300 AND LOC <= 900) OR (NOM > 30 AND NOM <= 40)
☑	high	Size	(LOC > 900 AND LOC <= 1500) OR (NOM > 40 AND NOM <= 50)
☑	very-high	Size	(LOC > 1500) OR (NOM > 50)

Figure 1: Default query settings for CodeMR.

The customized version of the query settings for *CodeMR* can be found in Figure 2. Please note that *CodeMR* does not allow us to change the metrics for method-level metrics, hence they are explained separately were applicable.

Let us now explain the reasoning behind changing/keeping certain query settings by discussing the changes per *quality attribute* (third column in both images).

Enable	Level	Quality Attribute	Condition
☑	low	Coupling	(CBO <= 5)
☑	low-medium	Coupling	(CBO > 5 AND CBO <= 10)
☑	medium-high	Coupling	(CBO > 10 AND CBO <= 15)
☑	high	Coupling	(CBO > 15 AND CBO <= 25)
☑	very-high	Coupling	(CBO > 25)
☑	low	Complexity	(WMC <= 20) OR (RFC <= 50) OR (DIT <= 1)
☑	low-medium	Complexity	(WMC > 20 AND WMC <= 50) OR (RFC > 50 AND RFC <= 100) OR (DIT > 1 AND DIT <= 3)
☑	medium-high	Complexity	(WMC > 50 AND WMC <= 101) OR (RFC > 100 AND RFC <= 150) OR (DIT > 3 AND DIT <= 10)
☑	high	Complexity	(WMC > 101 AND WMC <= 120) OR (RFC > 150 AND RFC <= 200) OR (DIT > 10 AND DIT <= 20)
☑	very-high	Complexity	(WMC > 120) OR (RFC > 200) OR (DIT > 20)
☑	low	Lack of Cohesion	(LCAM <= 0.6)
☑	low-medium	Lack of Cohesion	(LCAM > 0.6 AND LCAM <= 0.7)
☑	medium-high	Lack of Cohesion	(LCAM > 0.7 AND LCAM <= 0.8)
☑	high	Lack of Cohesion	(LCAM > 0.8 AND LCAM <= 0.9)
☑	very-high	Lack of Cohesion	(LCAM > 0.9)
☑	low	Size	(LOC <= 40) OR (NOM <= 20)
☑	low-medium	Size	(LOC > 40 AND LOC <= 225) OR (NOM > 20 AND NOM <= 25)
☑	medium-high	Size	(LOC > 225 AND LOC <= 450) OR (NOM > 25 AND NOM <= 30)
☑	high	Size	(LOC > 450 AND LOC <= 800) OR (NOM > 30 AND NOM <= 35)
☑	very-high	Size	(LOC > 800) OR (NOM > 35)

Figure 2: Customized query settings for CodeMR.

Coupling

High coupling prohibits, in most cases, future changes to a component that is used by a particular class. In other words, the modifications that happen to the dependent classes propagate back to the class that is making use of them. Our microservice architecture already aims to reduce the coupling in general.

Since we consider coupling to be quite an important metric within our microservices, we decided to lower the thresholds for the coupling between objects (**CBO**) by a constant value of 5. As much as this lowered threshold works for most classes, we are also more likely to encounter 'false positives' (from our perspective). An example of such a 'false positive' is our customizable validation chain which we use to validate requests. The composition of a validation chain may vary between endpoints and, therefore, requires to be manually constructed using our validation chain builder. Due to the varying number of single-responsibility classes that we offer to construct a validation chain, the coupling between objects increases for the controllers that make use of these chains. We do not directly consider this coupling to be bad, as it contributes to a **highly flexible** approach for validating requests.

Complexity

Complexity is semi-related to understanding the code. A higher complexity generally means that the code is harder to understand/grasp, which might cause misunderstanding and/or the creation of new bugs. Lower complexity can also be tied to an environment with high cohesion and low coupling.

Since we mainly want to focus on the coupling and the size, we have decided to not lower/higher the default values offered by *CodeMR*. Due to the nature of the Spring framework, as well as our current code base, we are not coping with severe complexity issues that can be effectively recognized by *CodeMR*.

Lack of Cohesion

Cohesion mostly tells us how much the methods of a class are related to one and another. For this metric, we have decided to keep the default values as well. The main reason for this is that our largest classes (controllers and services) usually do not contain methods that interact with methods from the same class, causing the lack of cohesion to increase.

Size

The group collaboratively decided that the threshold for the size of a class (**LOC**) should be approximately halved, and that the number of methods (**NOM**) should be re-scaled by collaboratively lowering all values by a reasonable amount that differs per severity. A large class does not necessarily have to be bad, but it is certainly undesirable in terms of testing, as well as keeping the class up to date. Reducing the size of a class by e.g., transferring methods to other/new classes, ensures that a class is much easier to manage/reason about. Not only will it improve on the aforementioned fields, but it will also make it easier for other members to review each other's work, according to an article about reviewing code from (SmartBear, sd).

As *CodeMR* does not allow the query settings to influence the result based on the methods inside a class, it was decided to manually analyze the value-based metrics that *CodeMR* shows within the IntelliJ plugin after generating a report. The group mainly looked at the methods that exceeded a maximum **MLOC**, method lines of code, value of *40*.

1.1 Classes that require refactoring

The classes on which is decided that they require refactoring, can be found below.

Attribute	Value
Class Name	UserController.java
Quality Attribute	Size & Coupling
Threshold and reason	10 <= CBO <= 20 & NOM <= 20; this class was huge and highly coupled. Introducing inheritance and splitting this class would address this issue.
How to improve	Extract class: Introduce inheritance with the base abstract class, splitting the functionality in terms of endpoints into different child classes.

Attribute	Value
Class Name	RecommendationController.java
Quality Attribute	Coupling
Threshold and reason	CBO <= 20; this class was highly coupled and was doing the creation of strategies which it should not be responsible for.
How to improve	Extract class: Move creation of strategies into a factory class.

Attribute	Value
Class Name	RecommendationStrategy.java HoursStrategy.java GradeStrategy.java TimesSelectedStrategy.java
Quality Attribute	Coupling
Threshold and reason	10 <= CBO <= 20; Even though the metric is not very alarming, we wanted to reduce the coupling in the Strategy classes.
How to improve	Extract class: Introduce inheritance by creating a parent class and making the four strategies the children of that class.

Attribute	Value
Class Name	HourDeclaration.java
Quality Attribute	Coupling
Threshold and reason	LCAM <= 0.5; There were endpoints that had nothing in common, but still were in the same controller. Considering how early in development it was, splitting the controller into three helped with scalability and eliminated potential high coupling.
How to improve	Extract class: Split endpoints belonging to statistics and greetings into their own class

Attribute	Value
Class Name	CourseController.java
Quality Attribute	complexity & Lack of Cohesion
Threshold and reason	LCAM < 0.7 & WMC < 0.3; Controller class contained endpoints not directly related to courses and thus could be split. Furthermore, endpoints (class methods) didn't make use of certain class attributes.
How to improve	Extract Class: The controller class is split into 2 controllers, one containing grade related endpoints and other containing courses related endpoints.

1.2 Methods that require refactoring

The methods on which is decided that they require refactoring, can be found below. As *CodeMR* mainly focusses on the severity of classes in the final report, the methods that require refactoring were based on **manual** analysis of the value-based metrics returned by *CodeMR*.

Attribute	Value
Class Name	SubmissionController.java
Method Name	*getContract(…)*
Quality Attribute	Size & Complexity
Threshold and reason	MLOC <= 40 & NBD < 5; to increase the readability of the code, we want to reduce the MLOC as well as the NBD.
How to improve	Extract Method: Reduce duplication of code/ re-use code by applying the extract method refactoring operation.

Attribute	Value
Class Name	SubmissionController.java
Method Name	*hireTa(…)*
Quality Attribute	Coupling
Threshold and reason	MC <= 9; To improve the coupling of the method, we would like to reduce the number of methods called by at least 3.
How to improve	Extract Class: repeated boilerplate into separate Director class.

Attribute	Value
Class Name	SubmissionController.java
Method Name	*reject(…)*
Quality Attribute	Coupling
Threshold and reason	MC <= 8; To improve the coupling of the method, we would like to reduce the number of methods called by at least 3.
How to improve	Extract Class: repeated boilerplate into separate Director class.

Attribute	Value
Class Name	CourseService.java
Method Name	*getMultipleUserGrades(…)*
Quality Attribute	Coupling & Complexity
Threshold and reason	MC <= 8 & NBD < 4; it is important for methods to be clear.
How to improve	Extract method: extract some functionality of the method to repository as a complex query.

Attribute	Value
Class Name	AsyncCourseCandidacyValidator.java
Method Name	*validate(…)*
Quality Attribute	MLOC
Threshold and reason	MLOC >= 40; it is important to keep methods short and concise. They should delegate more advanced functionality to more specialized method. MLOC value before refactoring: 47 lines.
How to improve	Extract method: separate functionality across multiple new member functions.

2 Refactoring

The following subsections describe the classes/methods that were refactored as part of this assignment. All refactoring operations are briefly described and accompanied by analyzed metrics derived from *CodeMR*.

2.1 Refactored classes

The following subsections describe all refactoring operations on classes. All refactored classes are prefixed with the name of the microservice.

2.1.1 [Users] UserController

2.1.1.1 Output of the metrics before/after refactoring

Before refactoring:

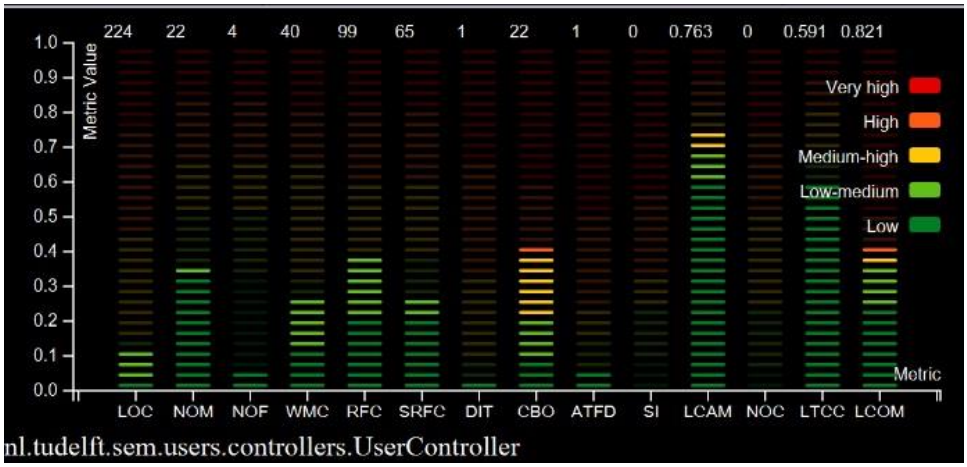

*Figure 3: Metrics of the UserController **before** refactoring.*

After refactoring:

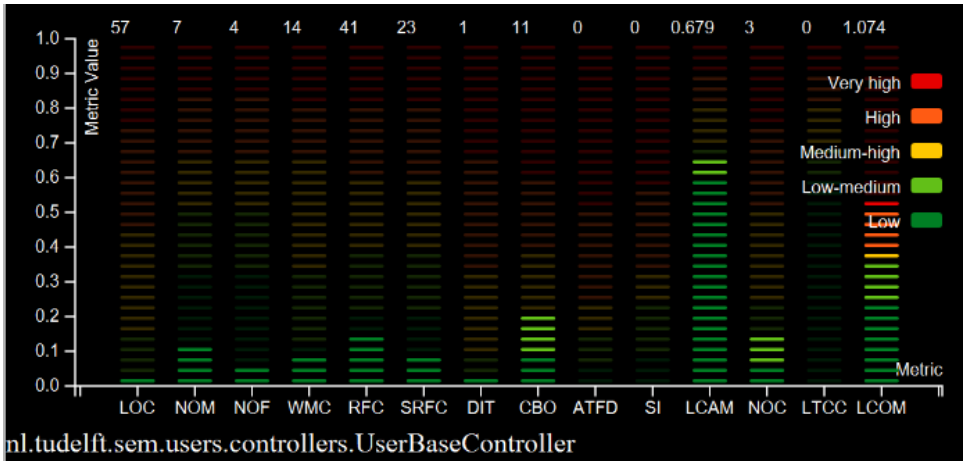

*Figure 4: Metrics of the UserBaseController **after** refactoring.*

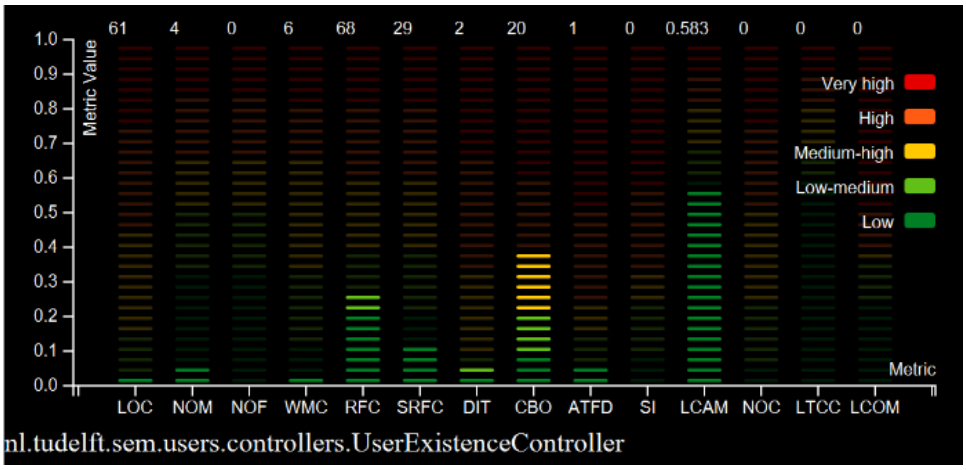

*Figure 5: Metrics of the UserExistenceController **after** refactoring.*

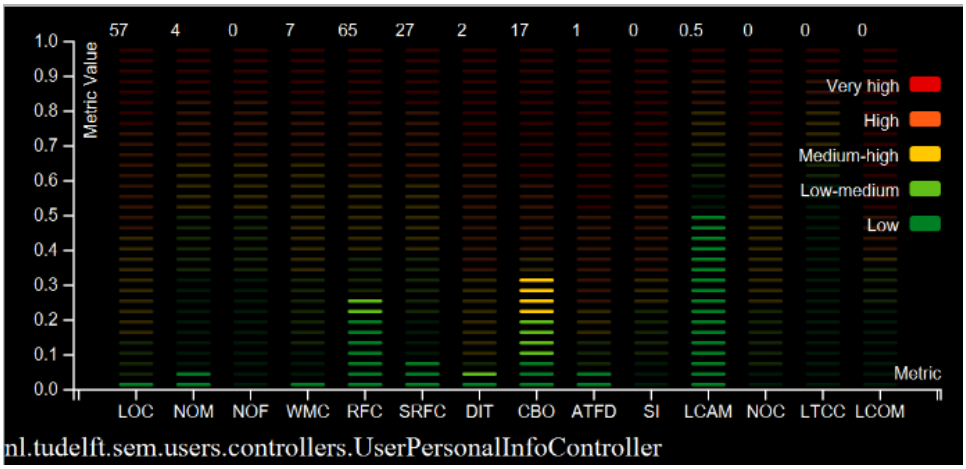

*Figure 6: Metrics of the UserPersonalInfoController **after** refactoring.*

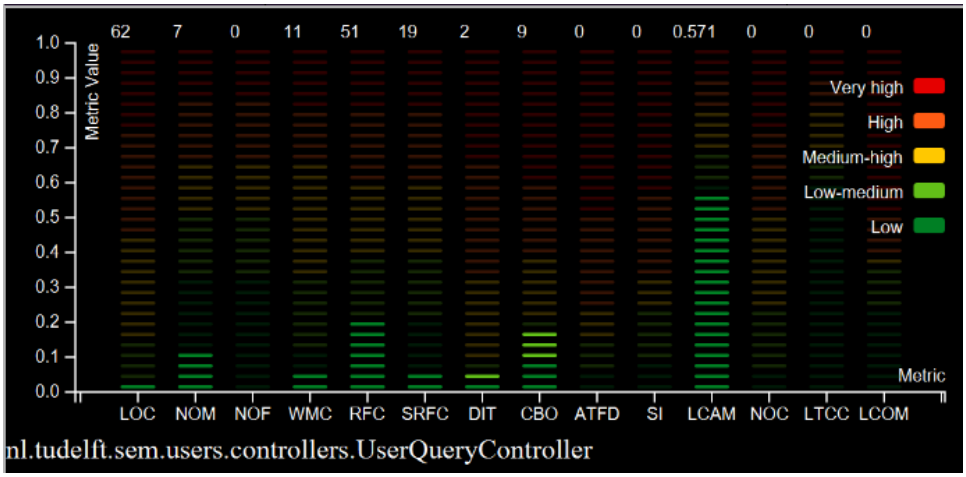

*Figure 7: Metrics of the UserQueryController **after** refactoring.*

2.1.1.2 Documentation of the types of refactoring operations

A part of the coupling issue within the UserController class was the related to its size (everything was in one class) and due to some previously extracted methods being used very rarely. Therefore, to solve this issue, we created a base abstract parent class *UserBaseController* which stores the common attributes and helper methods, and split the 10 endpoint methods into the following child classes: *UserExistenceController* (Create and Delete operations), *UserPersonalInfoController* (Update operations) and *UserQueryController* (Retrieve operations).

In addition, some methods could also have been modified. The method *createJson* was used quite little places (and only was used if all the values were strings) and thus has been removed (the functionality has been moved inside the method, which was not more than 2 extra lines of code). *attemptToRegister* was used only once and thus has been removed; instead, an exception handler has been added to the *UserExistenceController* (the only class this method has been called). The method *validateAdmin* consisted of one line, so we just replaced the call with this line.

As can be seen from the graphs, the size metric (Lines of Code) went lower, same holds for coupling (CBO, from high to medium and this only in one class). Cohesion metric (Lack of Cohesion Among Methods, LCAM) also went lower. It can be seen that the Lack of Cohesion of Methods metric increased; however, we are using LCAM as a metric for cohesion which did go lower. We think that the lack of cohesion among the members of the base class is caused by the fact that they serve as shared functions used by the derived classes.

For further information see this merge request (!115).

2.1.2 [Hiring Procedure] RecommendationController

2.1.2.1 Output of the metrics before/after refactoring

Before refactoring:

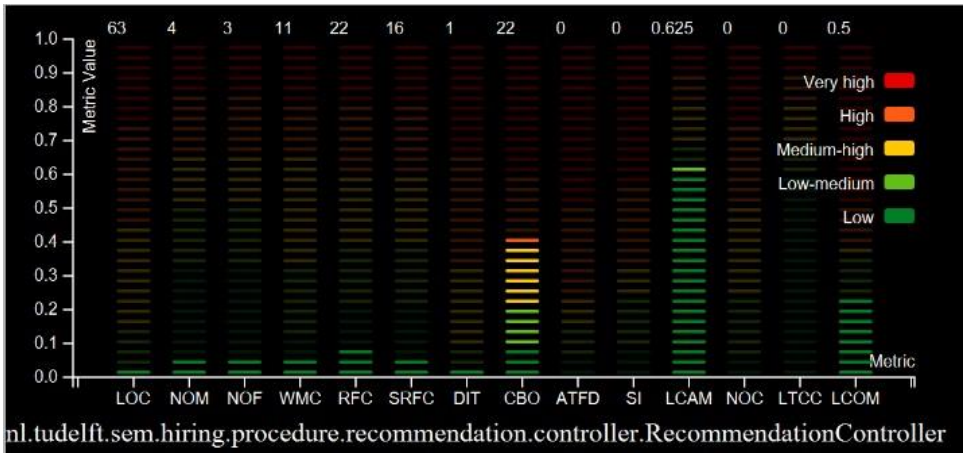

*Figure 8: Metrics of the RecommendationControlle **before** refactoring.*

After refactoring:

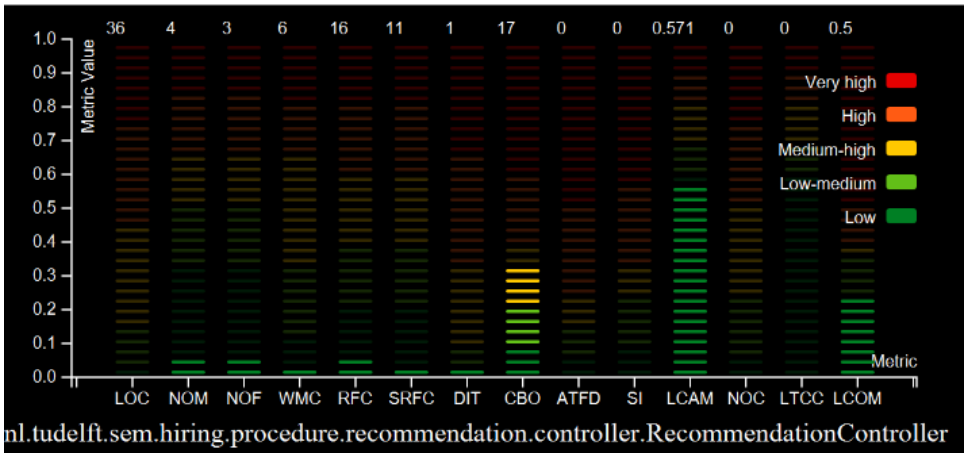

*Figure 9: Metrics of the RecommendationController **after** refactoring.*

2.1.2.2 Documentation of the types of refactoring operations

Inside the private *recommend* method of *RecommendationController* there was a switch statement that created a recommendation strategy based on the desired strategy type, see Figure 10. We thought that creating a suitable strategy is not a responsibility of *RecommendationController*, so we have decided to move this creation to a *RecommenderFactory* class (we have decided to rename RecommendationStrategy interface to Recommender to make it more explicit and remove the class Recommender since it was not useful and only increased coupling). Figure 11 and Figure 12 show the newly implemented classes.

```
private Mono<List<Recommendation>> recommend(long courseId, int amount,
                                             double minValue, StrategyType strategy,
                                             String authorization) {
    switch (strategy) {
        case TOTAL_TIMES_SELECTED:
            RecommendationStrategy strategy1 =
                    new TotalTimesSelectedStrategy(submissionRepository);
            return new Recommender(strategy1).recommend(courseId, amount, minValue);
        case TIMES_SELECTED:
            RecommendationStrategy strategy2 =
                    new TimesSelectedStrategy(submissionRepository, gatewayConfig,
                            authorization);
            return new Recommender(strategy2).recommend(courseId, amount, minValue);
        case GRADE:
            RecommendationStrategy strategy3 =
                    new GradeStrategy(submissionRepository, gatewayConfig, authorization);
            return new Recommender(strategy3).recommend(courseId, amount, minValue);
        //HOURS
        default:
            RecommendationStrategy strategy4 =
                    new HoursStrategy(submissionRepository, gatewayConfig, authorization);
            return new Recommender(strategy4).recommend(courseId, amount, minValue);
    }
}
```

*Figure 10: Code snippet of the recommend method **before** refactoring.*

```
private Mono<List<Recommendation>> recommend(RecommendationRequest req, String jwt) {
    Recommender strategy = RecommenderFactory
        .create(repo, gatewayConfig, jwt, req.getStrategy());
    return strategy.recommend(req.getCourseId(), req.getAmount(), req.getMinValue());
}
```

*Figure 11: Code snippet of the recommend method **after** refactoring.*

```
public class RecommenderFactory {

    private RecommenderFactory() {

    }

    /** Creates a recommender based on a certain strategy, from the given parameters. ...*/
    public static Recommender create(SubmissionRepository repo, GatewayConfig config,
                                     String jwt, StrategyType strategyType) {
        switch (strategyType) {
            case TOTAL_TIMES_SELECTED:
                return new TotalTimesSelectedStrategy(repo, config, jwt);
            case TIMES_SELECTED:
                return new TimesSelectedStrategy(repo, config, jwt);
            case GRADE:
                return new GradeStrategy(repo, config, jwt);
            //HOURS
            default:
                return new HoursStrategy(repo, config, jwt);
        }
    }
}
```

*Figure 12: Code snippet of the RecommendationFactory class **after** refactoring.*

In addition, to reduce the number of parameters in the private *recommend* method we introduced a new object *RecommendationRequest* (Introduce Parameter Object technique). This does introduce a new object, meaning a slight increase in coupling, however, since 5 parameters is too much, we still have decided to do it. Furthermore, some of the exceptions are now handled by *ObjectMapper*, which let us remove some manual deserialization and some catch blocks. As can be seen from the table, the results turned out to be positive, nevertheless.

```
private Mono<List<Recommendation>> parseBodyAndRecommend(String body, String authorization) {
    try {
        JsonNode node = new ObjectMapper().readTree(body);
        long courseId = Long.parseLong(node.get("courseId").asText());
        int amount = Integer.parseInt(node.get("amount").asText());
        double minValue = Double.parseDouble(node.get("minValue").asText());
        StrategyType strategy = StrategyType
                .valueOf(node.get("strategy").asText().toUpperCase(Locale.ROOT));

        return recommend(courseId, amount, minValue, strategy, authorization);
    } catch (JsonProcessingException e) {
        return Mono.error(new ResponseStatusException(HttpStatus.BAD_REQUEST,
                "Invalid request body format"));
    } catch (NumberFormatException e) {
        throw new ResponseStatusException(HttpStatus.BAD_REQUEST,
                "Cannot parse the number provided");
    } catch (IllegalArgumentException e) {
        return Mono.error(new ResponseStatusException(HttpStatus.BAD_REQUEST,
                "The specified strategy does not exist"));
    } catch (Exception e) {
        // Will be NullPointerException. Since, PMD complains about catching NPE,
        //  a general exception put here
        return Mono.error(new ResponseStatusException(HttpStatus.BAD_REQUEST,
                "Missing fields in the body"));
    }
}
```

*Figure 13: Code snippet of the parseBodyAndRecommend method **before** refactoring.*

```
@Data
@NoArgsConstructor
public class RecommendationRequest {
    private long courseId;
    private int amount;
    private double minValue;
    private StrategyType strategy;
}
```

*Figure 14: Code snippet of the RecommendationRequest **after** refactoring.*

```
private Mono<List<Recommendation>> parseBodyAndRecommend(String body, String authorization) {
    try {
        ObjectMapper mapper = new ObjectMapper();
        RecommendationRequest req = mapper.readValue(body, RecommendationRequest.class);
        return recommend(req, authorization);
    } catch (JsonProcessingException e) {
        return Mono.error(new ResponseStatusException(HttpStatus.BAD_REQUEST,
                "Invalid request body format"));
    } catch (Exception e) {
        // Will be NullPointerException. Since, PMD complains about catching NPE,
        //  a general exception put here
        return Mono.error(new ResponseStatusException(HttpStatus.BAD_REQUEST,
                "Missing fields in the body"));
    }
}
```

*Figure 15: Code snippet of the parseBodyAndRecommend method **after** refactoring.*

As can be seen from the bar charts in 2.1.1.1, the coupling metric (Coupling Between Object Classes, CBO) went down from high to medium-high.

For further information see this merge request (!113).

2.1.3 [Hiring Procedure] RecommendationStrategy (now Recommender) and other Strategies

2.1.3.1 Output of the metrics before/after refactoring

Before refactoring:

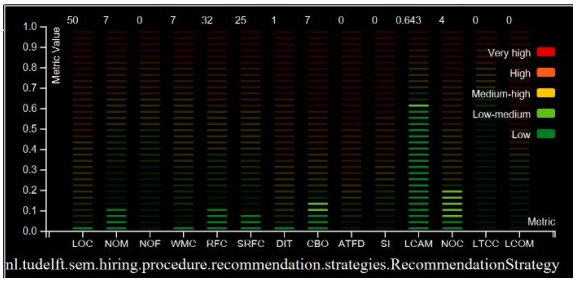

*Figure 16: Metrics of the RecommendationStrategy interface **before** refactoring.*

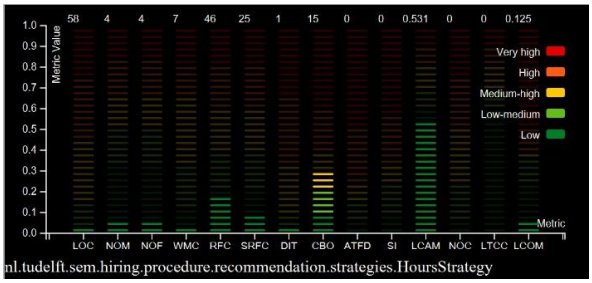

*Figure 17: Metrics of the HoursStrategy class **before** refactoring.*

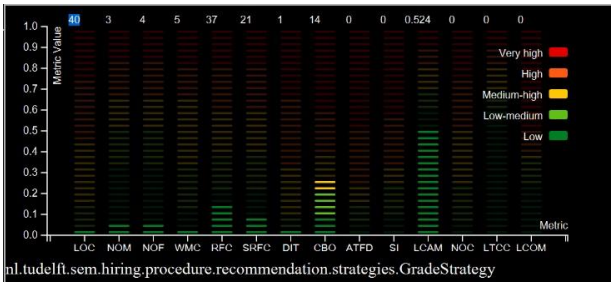

*Figure 18: Metrics of the GradeStrategy class **before** refactoring.*

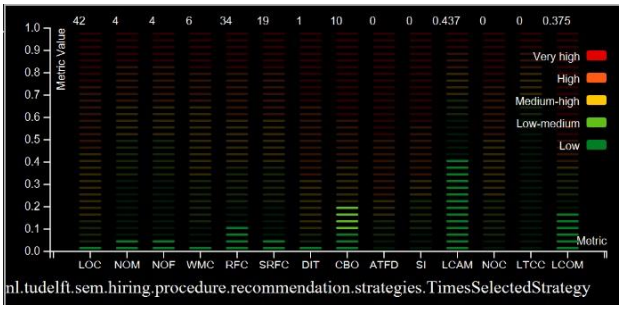

*Figure 19: Metrics of the TimesSelectedStrategy class **before** refactoring.*

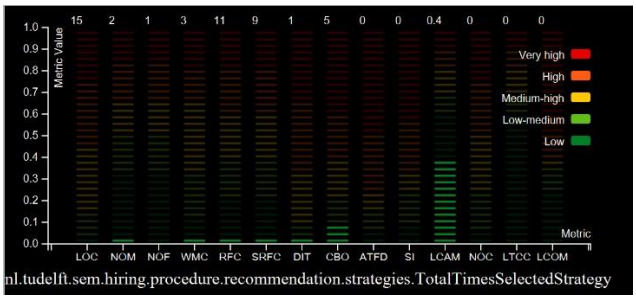

*Figure 20: Metrics of the TotalTimesSelectedStrategy class **before** refactoring.*

After refactoring:

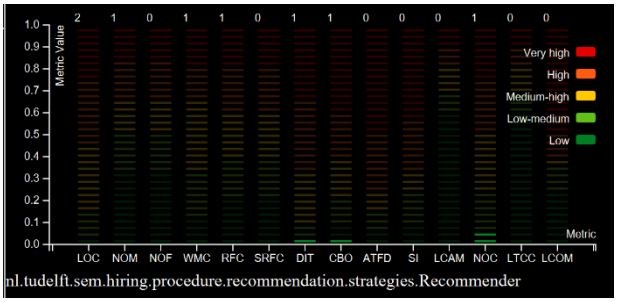

*Figure 21: Metrics of the Recommender interface **after** refactoring.*

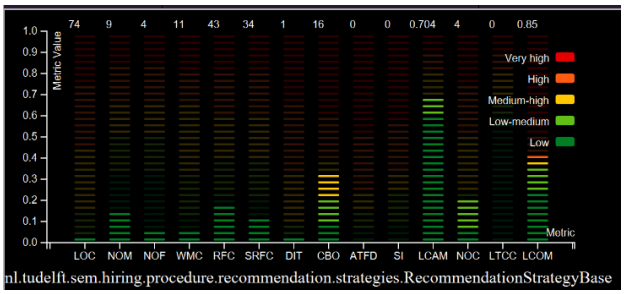

*Figure 22: Metrics of the RecommendationStrategyBase class **after** refactoring.*

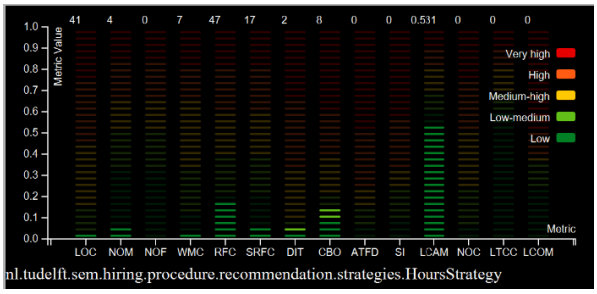

*Figure 23: Metrics of the HoursStrategy class **after** refactoring.*

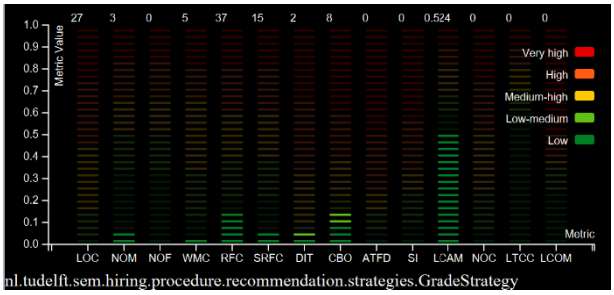

*Figure 24: Metrics of the GradeStrategy class **after** refactoring.*

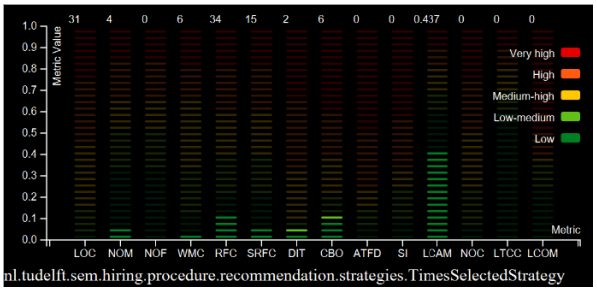

*Figure 25: Metrics of the TimesSelectedStrategy class **after** refactoring.*

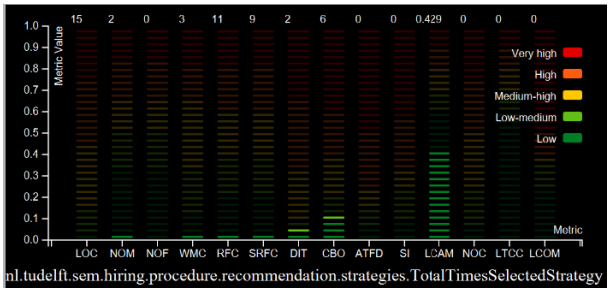

*Figure 26: Metrics of the TotalTimesSelectedStrategy class **after** refactoring.*

2.1.3.2 Documentation of the types of refactoring operations

Initially, we had an interface *RecommendationStrategy* with a lot of default methods, and each *Strategy* object had the same attributes. We have decided to create a base abstract parent class for the four strategies called *RecommendationStrategyBase* to store common attributes, make it implement *Recommender* (previously, *RecommendationStrategy*) interface and move the default methods from the interface to this class.

In addition, in some places in three of the strategies there were *get* and *post* requests that were essentially the same (*post* code was used in *GradeStrategy* and *HoursStrategy*; *get* code was in *TimesSelectedStrategy* and *HoursStrategy*). Thus, we have decided to move them in the parent class to reduce the lines of code in the method in the following way (Move Method technique: only the relevant code snippets are shown):

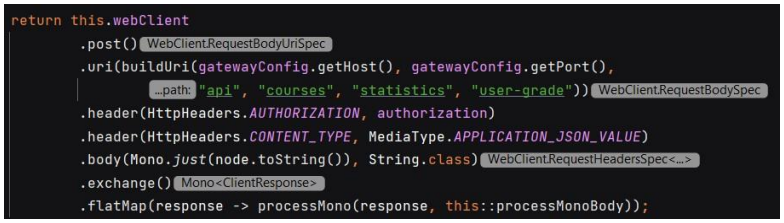

*Figure 27: Code snippet of the method with POST request **before** refactoring.*

*Figure 28: Code snippet of the method with POST request **after** refactoring.*

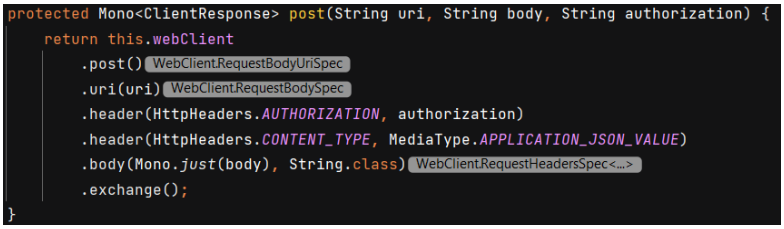

*Figure 29: Code snippet of the separate POST request method **after** refactoring.*

*Figure 30: Code snippet of method with GET request **before** refactoring.*

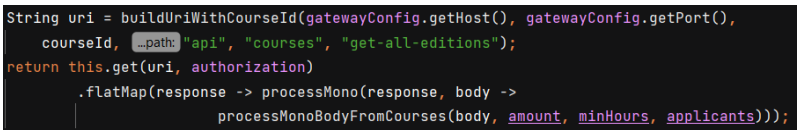

*Figure 31: Code snippet of the method with GET request **after** refactoring.*

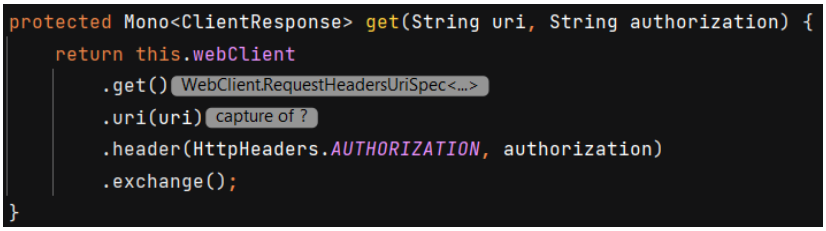

*Figure 32: Code snippet of the separate GET request method **after** refactoring.*

It can be seen from the graphs above that the coupling metric (CBO) for *HoursStrategy*, *GradeStrategy* and *TimesSelectedStrategy* went down (for the former two from medium-high to low-medium). However,

the coupling for *RecommendationStrategyBase* went slightly up, remaining medium-high. Nevertheless, this is in one class instead of two, and we think that semantically this is clearer now, and easy to extend (since we already have the base parent class). As for LCOM having increased for *RecommendationStrategyBase*, the comment is the same as was in section 2.1.1.2: this is not our metric for cohesion. Other than that, all other metrics are still within the boundaries.

For further information see this merge request (!108).

2.1.4 [Hour Management] HourDeclarationController

2.1.4.1 Output of the metrics before/after refactoring

Before refactoring:

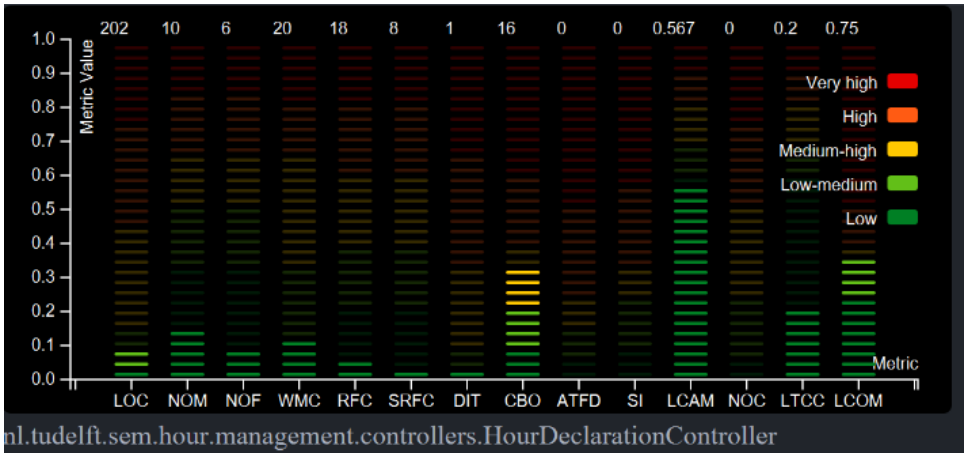

*Figure 33: Metrics of the HourDeclarationController **before** refactoring.*

After refactoring:

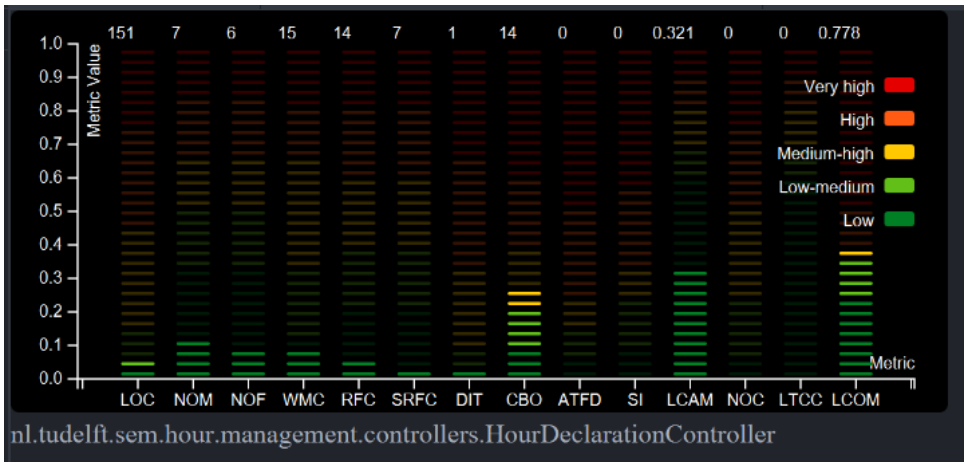

*Figure 34: Metrics of the HourDeclarationController **after** refactoring.*

2.1.4.2 Documentation of the types of refactoring operations

See this merge request (!75) for more information.

The problem in the old HourDeclarationController was that it had contained multiple methods that had no relationship with each other. This lack of relationship between class methods have been highlighted by the LCAM metric. It basically meant that, some of these methods do not belong to the same class.

Therefore, the refactoring operation was to move endpoints associated with (statistics) to their own special controllers. For instance, some of the endpoint specific dependencies: *NotificationService* for accept/reject and *StatisticsService* for statistical endpoints would not be imported into a general class.

This means that an error in one of these services wouldn't affect other methods/endpoints that had nothing to do with these dependencies. The improvement to the granularity of the classes can be seen in the decrease of LCAM and LTCC. (Lack of tight cohesion)

As this decision was made before adding further endpoints to both controllers, its real impact is much higher than what is indicated by these metrics. By combining and separating those three controllers in the current codebase and measuring the metrics individually, should portrait a more dramatic decrease in metrics.

2.1.5 [Courses] CourseController

2.1.5.1 Output of metrics before/after refactoring

Before refactoring:

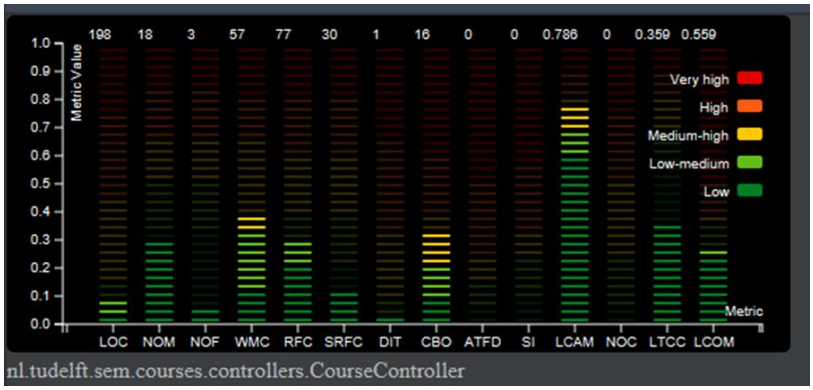

*Figure 35: Metrics of the CourseController **before** refactoring.*

After refactoring:

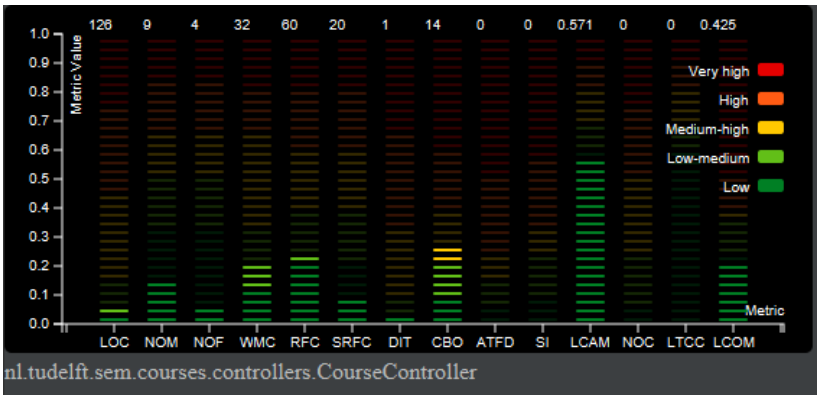

*Figure 36: Metrics of the CourseController **after** refactoring.*

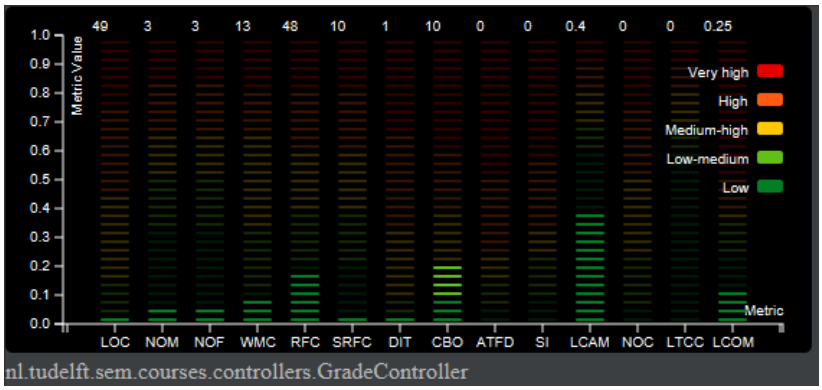

*Figure 37: Metrics of the GradeController **after** refactoring.*

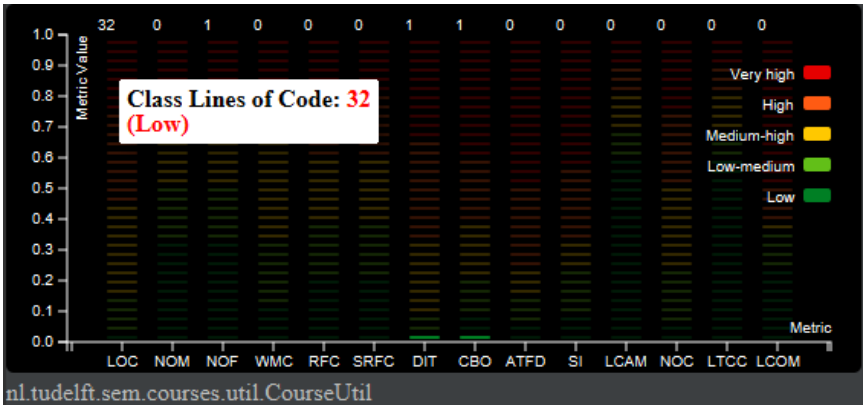

*Figure 38: Metrics of the CourseUtil class **after** refactoring.*

2.1.5.2 Documentation of the types of refactoring operations

The course controller initially contained all the endpoints for the courses microservice including those related to courses, grades, and lecturers. To increase the maintainability of the system, we decided to split the controller into two sections, one containing all endpoints relating to grades and the other for courses and lecturers. The lecturer endpoints remained in the *CourseController* as there were only a few. The move method refactoring operation reduced complexity and lack of cohesion significantly as shown by the graphs above. Moreover, we extracted authentication helper methods into a utility class, *CourseUtil*, which is imported by the controller classes. Though importing a class can increase coupling, the tradeoff made for better cohesion proved to be effective, with even coupling-related metrics such as CBO, coupling between objects, being lowered for all classes. The refactoring changes made can be found in this merge request (!114).

2.2 Refactored methods

The following subsections describe all refactoring operations on class methods. All refactored methods are prefixed with the name of the microservice, as well as the class they reside in.

2.2.1 [Hiring Procedure] SubmissionController – hireTa

2.2.1.1 Output of the metrics before/after refactoring

Before refactoring:

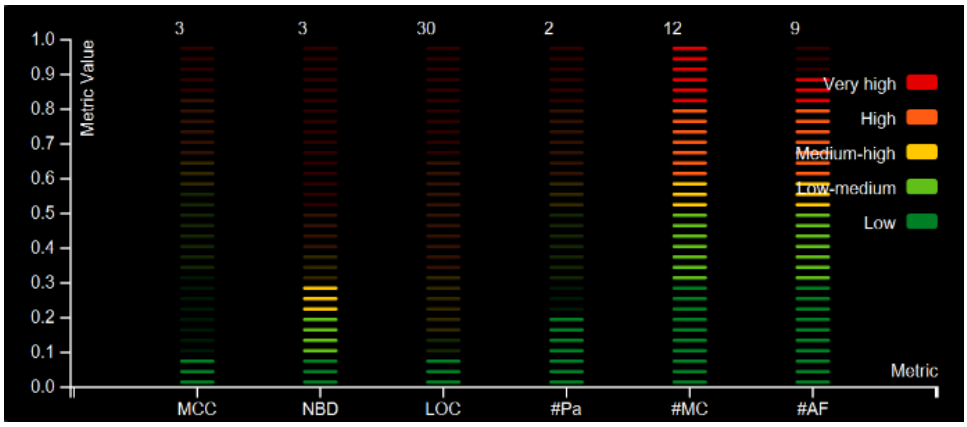

*Figure 39: Metrics of the hireTa method **before** refactoring.*

After refactoring:

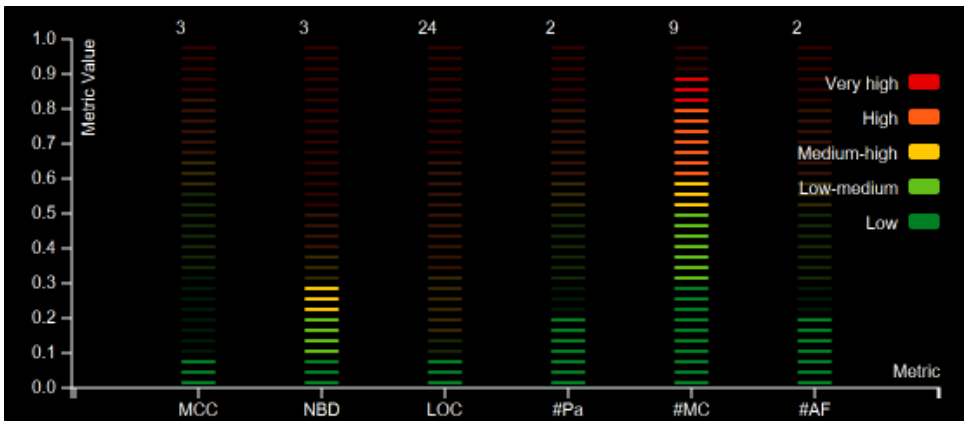

*Figure 40: Metrics of the hireTa method **after** refactoring.*

2.2.1.2 Documentation of the types of refactoring operations

A big part of the coupling issue within the *hireTa* method was the Validation Chain building. Therefore, to solve this issue, a new Director class was introduced to the Builder, as described in the Design Patterns lecture. As a result of this change, multiple methods within the *SubmissionController* class were also improved.

In the pre-refactoring version of the code, 6 validators were called and added to the Validation Chain. To fix this and reduce the amount of boilerplate code across all methods within the *SubmissionController*

class, the most used validation chains builder patterns were moved to a Director class. Therefore, lines 185-193 from the previous version were reduced to just one line in the refactor, see Figure 41.

Figure 41: Code snippet of the refactored replacements for validation chain construction.

We managed to arrive at an MC value of 7 by passing the functionality of extracting the Submission from the Optional wrapper to the *SubmissionService*, but ultimately decided against this. We decided to sacrifice points of the metric in favor of readability and maintaining the standard. To see our discussion, please check the corresponding merge request.

See this merge request (!117) for more information.

2.2.2 [Hiring Procedure] SubmissionController – rejectTa

2.2.2.1 Output of the metrics before/after refactoring

Before refactoring:

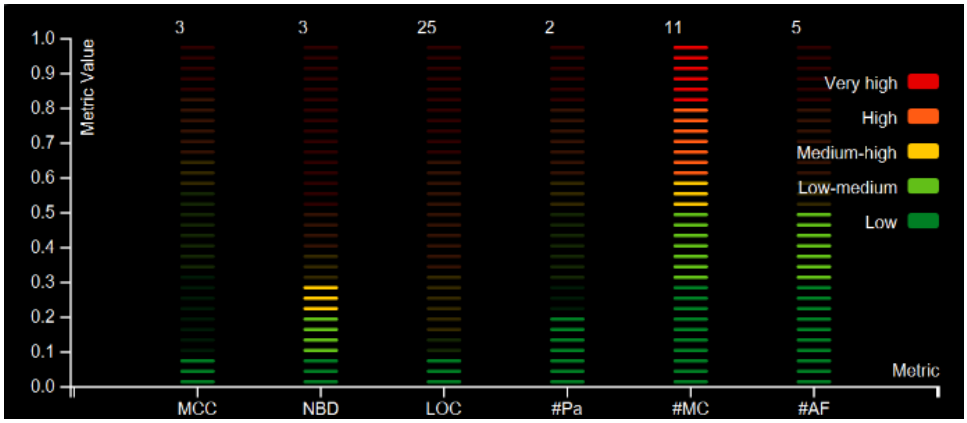

*Figure 42: Metrics of the rejectTa method **before** refactoring.*

After refactoring:

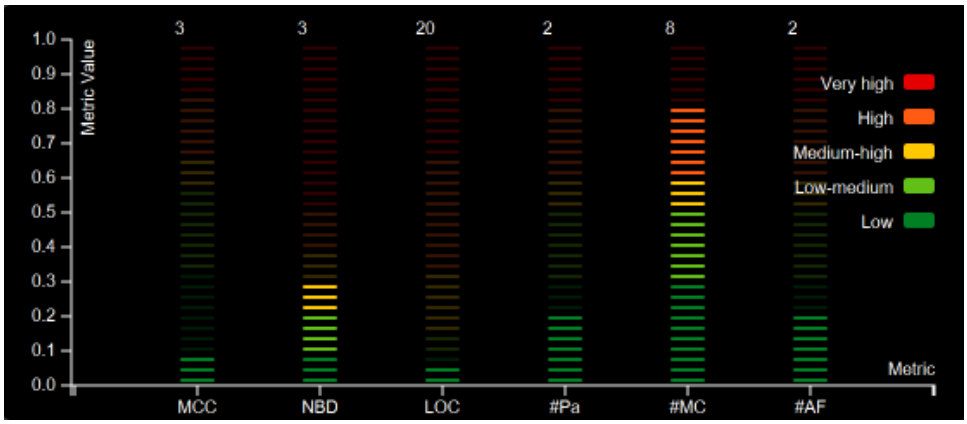

*Figure 43: Metrics of the rejectTa method **after** refactoring.*

2.2.2.2 Documentation of the types of refactoring operations

As presented previously, the solution to this coupling issue was introducing a Director to the *AsyncValidator* Builder.

In the pre-refactoring version of the code, 3 validators were called and added to the Validation Chain. To fix this, the Validation Chain building was moved to the Director class, under the name *chainAdminResponsibleLecturer*. Ultimately, this method was used in multiple endpoints. Therefore, lines 232-238 from the previous version were reduced to just one line in the refactor, see Figure 44.

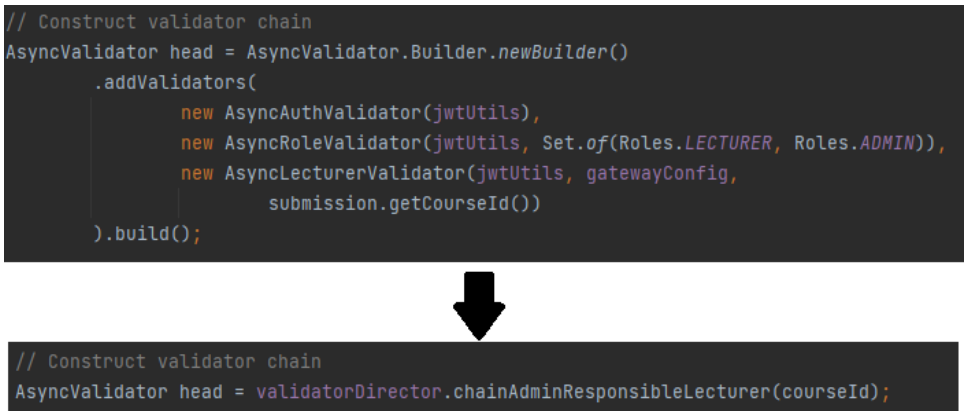

Figure 44: Code snippet of the refactored replacements for validation chain construction.

As with *hireTa()*, we could have arrived at a lower metric value, but decided against this in favor of readability. To see our discussion, please check the corresponding merge request.

See this merge request (!117) for more information.

2.2.3 [Hiring Procedure] SubmissionController – getContract

2.2.3.1 Output of the metrics before/after refactoring

Before refactoring:

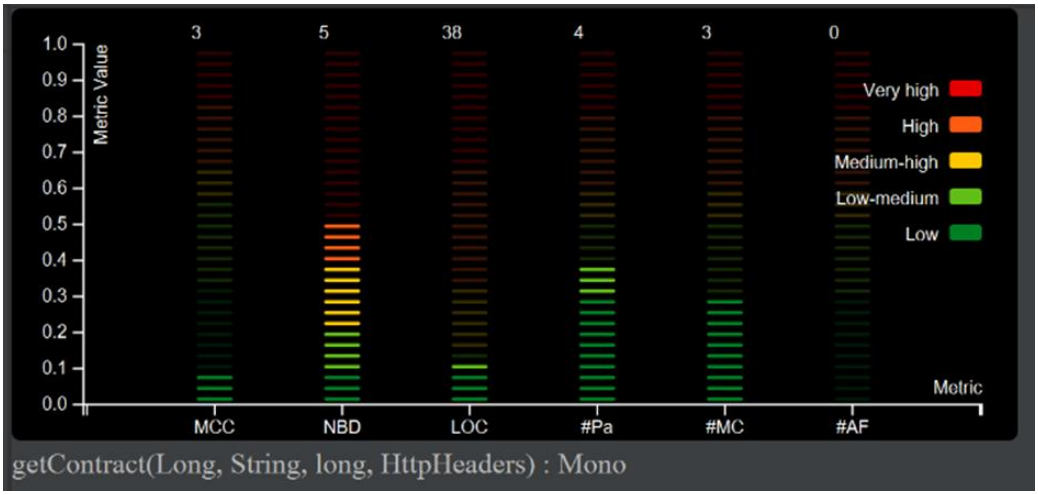

*Figure 45: Metrics of the getContract method **before** refactoring.*

After refactoring:

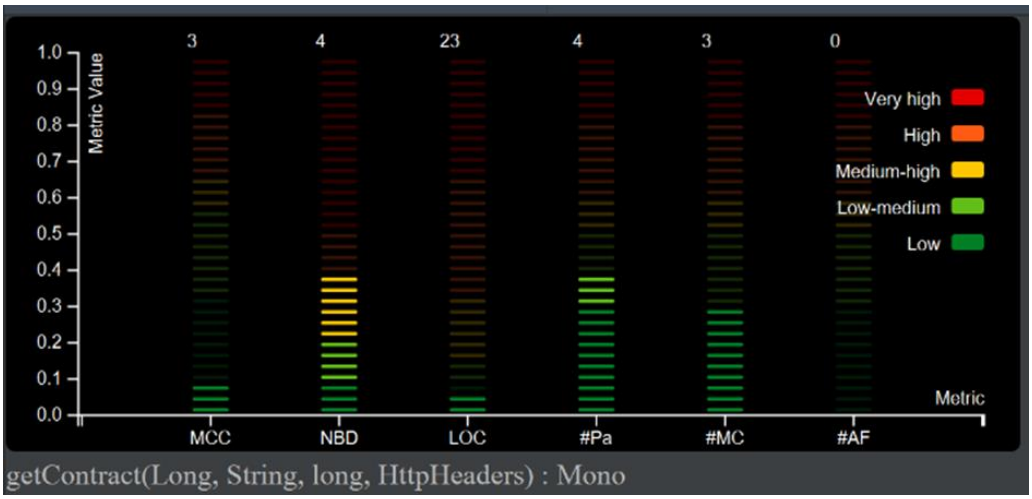

*Figure 46: Metrics of the getContract method **after** refactoring.*

2.2.3.2 Documentation of the types of refactoring operations

See this commit for the changes & this merge request (!111) for more information.

As we can see, the code from lines 334-341 & lines 345-353 of the old version (in red) are almost the same. The main difference between the two is the parameter '*userId*' or '*ownUserId*' in the method '*setContractParams*(…)'. So, we want to move this code to an external method [extract method refactoring] keeping this difference in mind.

Due to extracting the code to the method '*getContractHelper*(...)' (which we later changed the naming to '*convertContractToMonoDto*(…)' to reduce confusion) the duplication of the code is essentially removed. This is also possible due to making the '*userId*' or '*ownUserId*' a parameter of the method. The number of lines of code is therefore reduced, as well as the Nested Block Depth.

2.2.4 [Courses] CourseService – getMultipleUserGrades

2.2.4.1 Output of the metrics before/after refactoring

Before Refactoring

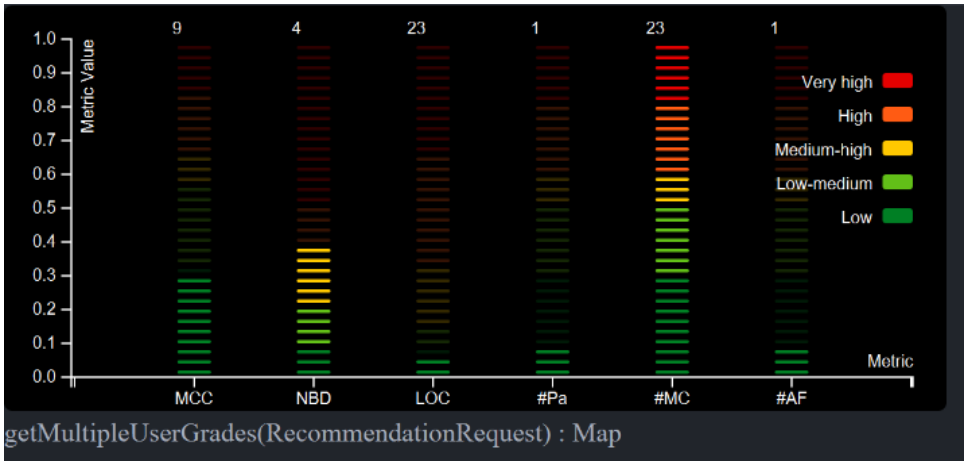

*Figure 47: Metrics of the getMultipleUserGrades method **before** refactoring.*

After Refactoring:

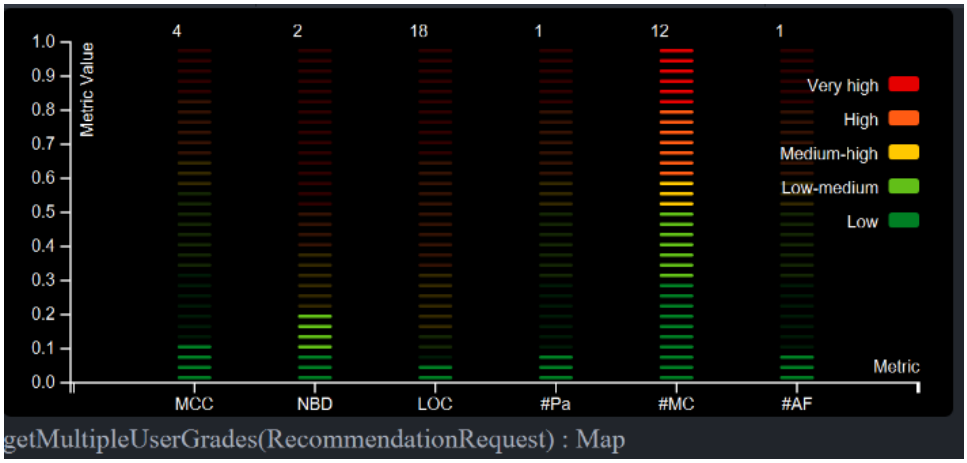

*Figure 48: Metrics of the getMultipleUserGrades method **after** refactoring.*

2.2.4.2 Documentation of the types of refactoring operations

See this merge request (!110) for changes.

As we can see, the code before refactoring is quite hard to understand, there are two sorting operations on the result at lines 235-236, and there are lots of access operations to the database.

The refactor aimed to reduce the load on the method by passing most of the functionality to the database, through use of a more complex query. The effects of delegating operations to database can be seen by the reduction of MC by half. (11 operations to be exact) The reason for the MC still being very high after refactoring is due to what is wanted of the method. The method has to limit the number of values based

on a provided value, then it has to extract the data into a desired format. Also, the error handling within the class is done through null/checks, which further adds onto MC value. Any further reduction on MC value could potentially mean compromising the functionality system offers; therefore, the method will not be refactored further.

Another metric the refactoring targeted was NBD (nested block depth). The reason this metric was high was because method fetched information about each individual course edition. This has been mitigated by modifying the SQL query to accept information about course editions. This way the repository checks whether the grade belongs to an intended course or not. Though refactoring, the outer loop (that checked each course edition individually) was removed which reduced NBD value.

2.2.5 [Hiring Procedure] AsyncCourseCandidacyValidator – validate

2.2.5.1 Output of the metrics before/after refactoring

Before refactoring:

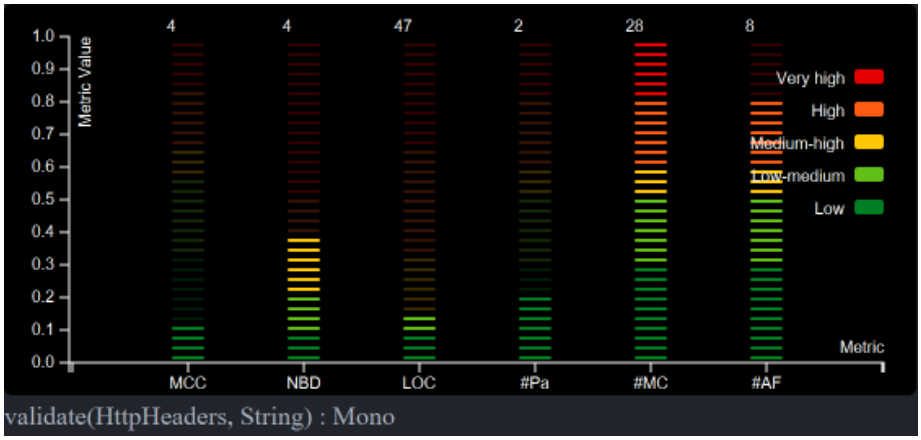

*Figure 49: Metrics of the validate method **before** refactoring.*

After refactoring:

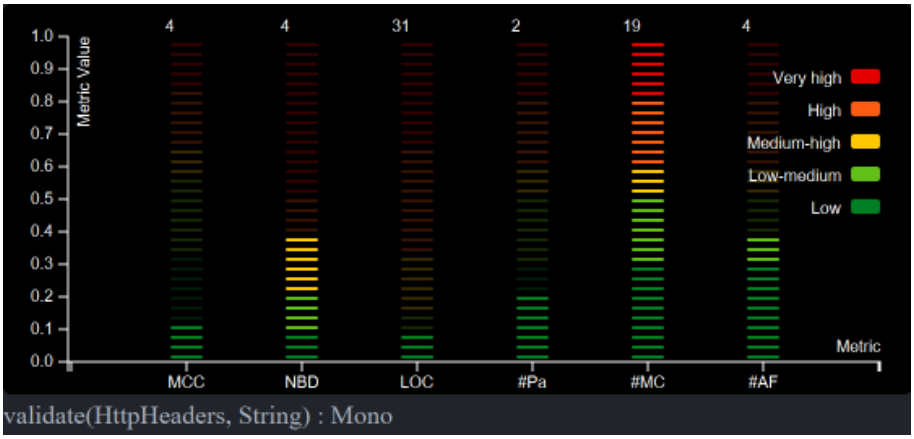

*Figure 50: Metrics of the validate method **after** refactoring.*

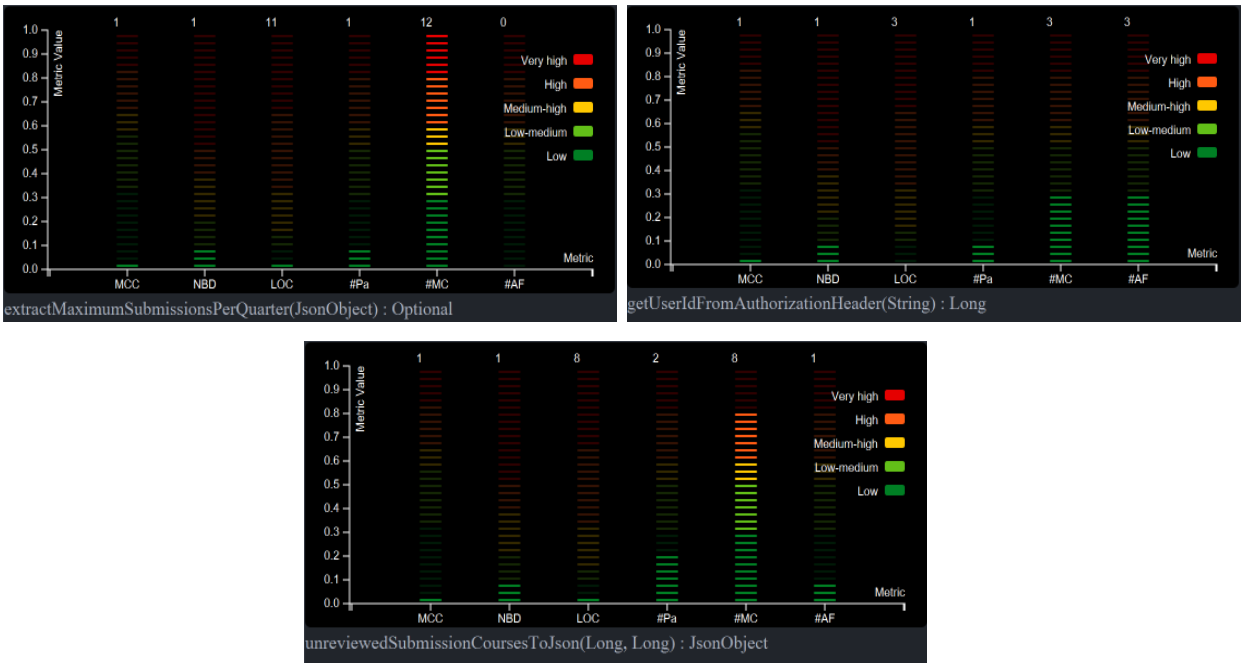

*Figure 51: Metrics of the extracted methods **after** refactoring.*

NOTE: The IntelliJ *CodeMR* plugin does not allow a threshold to be set/changed for, e.g., **#MC** (number of method count).

2.2.5.2 Documentation of the types of refactoring operations

See this merge request (!112) or this commit for changes.

By looking at the former approach in the referenced merge request, we can quickly conclude that the method is too big. The method being too big makes it harder for reviewers or other team member to grasp what the method does. To make this easier, we decided to reduce the number of lines of code for our method by using **extract method** refactoring. This allowed us to extract sequences of code that could be viewed separately, which we then documented extensively (using JavaDoc).

From the bar charts in 2.2.5.1, we can deduce that we managed to decrease the **MLOC**, method lines of code, metric by *16*. A decent amount judging by the fact that we now have three new methods that also contain proper documentation as opposed to comments only. The same holds for the number of accessed fields (**#AF**) that got improved by using these fields *only* in the methods where they are of most use, e.g., *JwtUtils* to parse JWT tokens.

Although we can see that the number of method calls (**#MC**) is lower, but still high, there is no need to worry. The reason why this number is so high has to do with a variant of the builder notation used by the *WebClient* class (for initiating HTTP requests). Furthermore, we also make use of the stream API from Java to group and aggregate the responses returned by the course microservice. There are known ways to refactor this by segmenting the build and/or stream operations. However, this has a high chance of working the opposite way and create more confusion for the reader, because the segmented parts were all needed for a single purpose (e.g., performing an HTTP request).

3 References

Bigonha, M., Ferreira, K., Souza, P., Sousa, B., Januário, M., & Lima, D. (2019). The usefulness of software metric thresholds for detection of bad smells and fault prediction. *Information and Software Technology, 115*, 79-92. doi:https://doi.org/10.1016/j.infsof.2019.08.005

SmartBear. (n.d.). *Best Practices for Code Review*. Retrieved 2022, from smartbear.com: https://smartbear.com/learn/code-review/best-practices-for-peer-code-review/#:~:text=Review%20fewer%20than%20400%20lines,ability%20to%20find%20defects%20diminishes