

Software Engineering Methods: Lab Assignment 2

Group 18A

January 2022

Chapter 1

Introduction

The main focus of this assignment is to analyze our current code and refactor it such that it becomes more efficient and maintainable. Thus, we are improving the non-functional properties of the software under analysis. We will make use of two powerful tools for computing code metrics at both class and method levels: **CodeMR** and **MetricsTree**.

Chapter 2

Target Code Metrics

In this section, we will outline the code metrics we will target, the thresholds we identify for these metrics, and a justification of our choices. We decided upon the following five code metrics after inspecting the reports generated by **CodeMR** and **Metrics Tree**.

Class Complexity - CodeMR

Several classes in our system were indicated to have **High** complexity. We intend to address this in our refactoring operations.

High complexity not only makes it more difficult for us to maintain but it also complicates detecting potential bugs, devising test cases, and introducing new changes. On other other hand, we also acknowledge that certain methods simply perform complex functionality. Therefore, we believe that the threshold for complexity should be at most **Low Medium**.

Furthermore, to justify reducing complexity, it is established that it can be more difficult for us (developers) to maintain the complex code.[1] As we intend on maintaining and extending our system, we believe that lower complexity will facilitate our development process in the future.

Lack of Cohesion - CodeMR

We believe that the lack of cohesion implies that we grouped unrelated or independent functionality into one class. While this means that we usually need to change one class when adding functionality, it still makes adding new features more difficult.

For example, one class may interact with multiple subdomains. When we wish to alter one subdomain, we need to navigate through the class and edit all instances where the class interacts with the subdomain. This compromises maintainability.

Therefore, we believe that we need to have **Low** or **Low Medium** Lack of Cohesion. Higher lack of cohesion can make our code more difficult to maintain and can result in a large class, two code smells according to Refactoring Guru[2]. Moreover, since we intend on extending our system, it is reasonable to maintain a relatively low lack of cohesion in case we add more functionality.

High Coupling - CodeMR

According to the report generated by CodeMR, we find that several classes in our system have **High** coupling. This results from the various Service classes that require access to the database and communication with other microservices. These service classes therefore have a high number of dependencies.

However, in accordance with software design principles, we acknowledge that high coupling introduces poor maintainability[3]. If one component changes, we will need to alter several other classes. This can quickly become unmanageable when our code base expands.

On the other hand, it is impossible to avoid coupling. This is because certain classes are simply dependent, such as the Service and Repository class. Therefore, we believe that a suitable threshold is at most **Medium-High** coupling.

Number of Method Parameters - MetricsTree

The number of method parameters is an important indicator of code quality. A high number of method parameters can make the method more difficult to unit test and compromises readability. Instead of having a high number of parameters passed to a method, passing an object instance is a better option, as it makes the code easier to read.

According to the rules of the Checkstyle plugin, a tool widely used to assess the quality and readability of code, 7 is specified as an upper bound for number of method parameters [4]. This is also mentioned by Steve McConnell in his book, *Code Complete*.

Cyclomatic Complexity - MetricsTree

Having a low cyclomatic complexity makes the code easier to read, test and modify. A method with a very large cyclomatic complexity may require a significant number of tests to achieve complete coverage and it becomes very cumbersome to modify, as one change could have knockdown effects in the code that follows it. In the book *Code Complete*, Steve McConnell recommends that between 0-5 is acceptable, when a method reaches 6-10, the developer should be aware that the complexity has increased significantly and over 11 should be refactored.

Chapter 3

Class Refactoring

Application Controller

Problem Overview

The Application Controller is responsible for handling all HTTP requests from the client. Prior to the refactoring operations, the Application Controller used the Repository classes directly to interact with the database and handled any errors that may occur during service to service communication. Notably, this class had **Medium High Complexity**.

List of all classes (#14)

ID	CLASS	COUPLING	COMPLEXITY	LACK OF COHESION	SIZE	LOC	COMPLEXITY	COUPLING	LACK OF COHESION	SIZE
1	ApplicationContro...	■	■	■	■	321	medium-high	medium-high	medium-high	medium-high

Figure 3.1: Application Controller: Prior to refactoring

Refactoring Operation

Hence, we decided to refactor this class. We added another class called **Application Service** and acts as the only dependency for the Application Controller class. It is responsible for interacting with the database and checking for constraints enforced by the system. Functionality that was previously in the Application Controller was moved to the Application Service.

However, this is with exception to the error handling. Methods in the Application Service will throw custom exceptions indicating which error occurred and the Application Controller will handle these accordingly while providing the client with a corresponding HTTP response.

We can find the improvement of the metric for before and after refactoring:

2	ApplicationContro...	■	■	■	■	174	low-medium	medium-high	low-medium	low-medium
---	----------------------	---	---	---	---	-----	------------	-------------	------------	------------

Figure 3.2: Application Controller: After refactoring

Motivation

We believe that this refactoring operation was necessary as it decreased the complexity of the Application Controller. In order to scale our system, we must ensure that we can integrate new endpoints without significantly increasing complexity. This can be achieved by reducing the number of concerns and dependencies in the Application Controller. Furthermore, we believe that **Medium High** complexity needs to be reduced to **Low Medium** because this allows us to maintain the complexity at a reasonable level and open to adding more functionality. It is also important to note that this refactoring operation was applied to Controllers in *all* microservices.

Application Service

Problem Overview

The application service had several methods with high cyclomatic complexity. This resulted in a **Medium-High** complexity. Besides the complexity we also had **Medium-High** coupling and lack of cohesion, this was because some classes did jobs that had very little overlap in purpose as well as the data they were handling.

ID	CLASS	COUPLING	COMPLEXITY	LACK OF COHESION	SIZE	LOC	COMPLEXITY	COUPLING	LACK OF COHESION	SIZE
1	ApplicationService					243	low-medium	medium-high	medium-high	low-medium

Figure 3.3: Application Service: Prior to refactoring

Refactoring Operations

The methods with high cyclomatic complexity had a lot of their branches moved to other classes, removed or worked around in some other way, this brought down the cyclomatic complexity to be either **Low** or **Low-Medium**.

We split the methods with high coupling and lack of cohesion up among multiple classes that all handled data that are closely related. This decreased the amount of coupling and nearly completely fixed the lack of cohesion. These classes are:

- **Application Service:** Anything that was not categorized in any of the other services stayed in this service.
- **Application Data Service:** All methods that are related to application data and required retrieving data from the TA service.
- **Filter Service:** All methods related to filtering applications.
- **Select Applicant Service:** All methods related to selecting an applicant as TA, as this requires a unique set of variables and data that other methods don't use.

ID	CLASS	COUPLING	COMPLEXITY	LACK OF COHESION	SIZE	LOC	COMPLEXITY	COUPLING	LACK OF COHESION	SIZE
5	ApplicationService					78	low	low-medium	low	low-medium

Figure 3.4: Application Service: After refactoring

Motivation

The main reason we refactored this service after we created it, is that this is the main service and logic for the entire application. Thus it having complexity, lack of cohesion and coupling of **Medium-High** was not the right thing. Therefore we came to the conclusion that this was a must-do to refactor.

TA Service

Problem Overview

The TA service accesses many data tables, including the TA, Workload, Contract, and Review table. Prior to refactoring, the TA service was responsible for all operations involving these data tables.

However, some methods required only accessing one specific table, for example writing a review only required accessing the Review table. This resulted in a **High** lack of cohesion between the methods within this class. Furthermore, since all Repository classes were dependencies for the TA service, it also had **High** coupling.

ID	CLASS	COUPLING	COMPLEXITY	LACK OF COHESION	SIZE	LOC	COMPLEXITY	COUPLING	LACK OF COHESION	SIZE
1	TaService					361	medium-high	high	high	medium-high

Figure 3.5: TA Service: Prior to refactoring

Refactoring Operations

To increase cohesion and reduce coupling, we introduced more classes: **Contract Service**, **Review Service**, and **Workload Service**. The refactoring operations is as follows:

- Take all methods *only* requiring ***Repository** into ***Service**. For example, if a method only requires the **Workload Repository**, we move the method to the **Workload Service**.
- Inject the ***Service** as a dependency into the **TA Controller** and then call the ***Service** instead of the **TAService** when the corresponding method is called.
- For methods that require the usage of *multiple* Repositories, we allow that the method remains in the **TA Service**.

We find that the amount of coupling and lack of cohesion improved. The lack of cohesion in the newly introduced service classes are **Low**. The lack of cohesion in the TA service is **Low Medium** and the level of coupling is **Medium High**.

ID	CLASS	COUPLING	COMPLEXITY	LACK OF COHESION	SIZE	LOC	COMPLEXITY	COUPLING	LACK OF COHESION	SIZE
1	TaService	medium-high	low	low-medium	73	low	medium-high	medium-high	low-medium	low-medium

Figure 3.6: TA Service: After refactoring

Motivation

We believe that **High** lack of cohesion and **High** coupling requires refactoring.

The lack of cohesion indicates that unrelated functionality are grouped within one class. Not only does this result in a long class, but it also implies that we do not have a clear mapping from class to subdomain. This makes maintaining the system more challenging.

For example, if we want to change how the **Workload** objects are processed within the system, we will need to navigate to specific methods within the class and change them one by one. However, having a **Workload Service** class, we can easily change this by adding one class method. It is also now clear that each service class maps to a subdomain.

Furthermore, we have that one service class communicates with at most two external classes. This reduces the coupling between classes, making our system more scalable and maintainable.

TA Controller

Note that these refactoring operations were applied after the TA Service refactoring (as mentioned above) was done.

Problem Overview

The TA Controller is responsible for handling the HTTP requests of clients to the TA microservice. It now employs many service classes to interact with the database and to provide the client with a valid HTTP response. However, this introduced **Medium High** Lack of Cohesion and Coupling.

The main cause is that one method *always* required *one* Service class. This resulted from how we divided functionality among the service classes for the previous refactoring operation.

2	TaController	medium-high	medium-high	medium-high	379	medium-high	medium-high	medium-high	medium-high	medium-high
---	--------------	-------------	-------------	-------------	-----	-------------	-------------	-------------	-------------	-------------

Figure 3.7: TA Controller: Prior to refactoring

Refactoring Operations

To target these non-desirable properties, we added several more class: **Contract Controller**, **Workload Controller**, and **Review Controller**. The refactoring operations were as follows:

- Move all methods requiring the ***Service** into the ***Controller**. For example, a method requiring the **Contract Service** is moved to the **Contract Controller**.

- Alter the endpoint corresponding to each method, so that it is clear which Controller class handles which endpoint.
- Inject the corresponding service class as a dependency into the controller class.
- Since other microservices communicate with the TA microservice, we also need to change the URLs other microservices use to address the TA microservice.

These refactoring operations allowed for **Low** coupling and **Low** lack of cohesion.



Figure 3.8: TA Controller: After refactoring

Motivation

By reducing the lack of cohesion, we ensure that each controller class corresponds to a central responsibility in the TA microservice. For example, it is now clear that the **Workload Controller** handles all HTTP requests about the workloads of teaching assistants. This allows us to add new functionalities without creating a single large class.

Furthermore, each controller class only uses the necessary service class. This reduces the level of coupling in controller classes, allowing for easier testing and extending of functionalities in individual classes.

Course Service

Problem Overview

The Course service requires access to two data tables, which are the Course and Grade table. Prior to refactoring, the Course Service was responsible for all operations involving both data tables. Since most operations used only one of the two databases it caused **Medium-High** Lack of Cohesion.

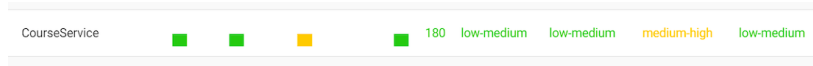


Figure 3.9: Course Service: Prior to refactoring

Refactoring Operations

Therefore, we came to the conclusion that this class required refactoring. We added two other classes called **Grade Service** and **Recruitment Service**.

- All methods using only the Course Repository stayed in the **Course Service**.
- All methods using only the Grade Repository were moved to the **Grade Service**.
- The last method remaining that used both repositories which depends on recruiting, was moved to the **Recruitment Service**.

We find that the lack of cohesion greatly improves. The lack of cohesion in the newly introduced service classes are **Low**. The lack of cohesion in the Course Service is **Low-Medium**.

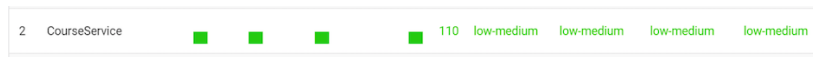


Figure 3.10: Course Service: After refactoring

Motivation

We believe that this service, being responsible for all logic, should not have a **Medium-High** or higher lack of cohesion. The higher lack of cohesion indicates that unrelated functionality are grouped within a class making it more difficult to understand and improve such classes, limiting the scalability and readability of the code.

Total Overview of All Refactoring Operations

Below is an overview of how all the previous five class refactoring operations have affected the three microservices: Application Microservice, TA Microservice, and Course Microservice.

Course Microservice

List of all classes (#14)										
ID	CLASS	COUPLING	COMPLEXITY	LACK OF COHESION	SIZE	LOC	COMPLEXITY	COUPLING	LACK OF COHESION	SIZE
1	CourseController					194	low-medium	medium-high	low-medium	low-medium
2	CourseService					180	low-medium	low-medium	medium-high	low-medium
3	TokenFilter					31	low-medium	low-medium	low	low
4	AuthorizationConfig					16	low-medium	low-medium	low	low
5	EmptyTargetException					4	low-medium	low	low	low
6	InvalidCourseExce...					4	low-medium	low	low	low
7	Course					45	low	low	low	low
8	TaCommunicator					33	low	low	low	low

Figure 3.11: Course Microservice: Prior to class refactoring

List of all classes (#17)										
ID	CLASS	COUPLING	COMPLEXITY	LACK OF COHESION	SIZE	LOC	COMPLEXITY	COUPLING	LACK OF COHESION	SIZE
1	CourseController					192	low-medium	medium-high	low-medium	low-medium
2	CourseService					110	low-medium	low-medium	low-medium	low-medium
3	TokenFilter					31	low-medium	low-medium	low	low
4	AuthorizationConfig					16	low-medium	low-medium	low	low
5	EmptyTargetException					4	low-medium	low	low	low
6	InvalidCourseExce...					4	low-medium	low	low	low
7	Course					45	low	low	low	low
8	GradeService					41	low	low	low	low

Figure 3.12: Course Microservice: After class refactoring

TA Microservice

ID	CLASS	COUPLING	COMPLEXITY	LACK OF COHESION	SIZE	LOC	COMPLEXITY	COUPLING	LACK OF COHESION	SIZE
1	TaService					361	medium-high	high	high	medium-high
2	TaController					379	medium-high	medium-high	medium-high	medium-high
3	TokenFilter					31	low-medium	low-medium	low	low
4	AuthorizationConfig					16	low-medium	low-medium	low	low
5	InvalidStatusExce...					4	low-medium	low	low	low
6	EmptyTargetException					4	low-medium	low	low	low
7	AddRoleFailureExc...					4	low-medium	low	low	low
8	DuplicateObjectEx...					4	low-medium	low	low	low
9	Contract					37	low	low	low	low

Figure 3.13: TA Microservice: Prior to class refactoring

ID	CLASS	COUPLING	COMPLEXITY	LACK OF COHESION	SIZE	LOC	COMPLEXITY	COUPLING	LACK OF COHESION	SIZE
1	TaService					73	low	medium-high	low-medium	low-medium
2	ContractController					156	low-medium	low-medium	low	low-medium
3	ContractService					131	low-medium	low-medium	low	low-medium
4	WorkloadController					102	low-medium	low-medium	low	low-medium
5	WorkloadService					96	low-medium	low-medium	low	low-medium
6	TaController					60	low-medium	low-medium	low	low-medium
7	TokenFilter					31	low-medium	low-medium	low	low
8	AuthorizationConfig					16	low-medium	low-medium	low	low
9	ReviewController					78	low	low-medium	low	low-medium

Figure 3.14: TA Microservice: After class refactoring

Application Microservice

ID	CLASS	COUPLING	COMPLEXITY	LACK OF COHESION	SIZE	LOC	COMPLEXITY	COUPLING	LACK OF COHESION	SIZE
1	ApplicationService					243	low-medium	medium-high	medium-high	low-medium
2	ApplicationContro...					174	low-medium	medium-high	low-medium	low-medium
3	TokenFilter					31	low-medium	low-medium	low	low
4	AuthorizationConfig					16	low-medium	low-medium	low	low
5	InvalidApplicatio...					4	low-medium	low	low	low
6	EmptyTargetElemen...					4	low-medium	low	low	low
7	TaCommunicator					50	low	low	low	low
8	CourseCommunicator					46	low	low	low	low

Figure 3.15: Application Microservice: Prior to class refactoring

List of all classes (#21)

ID	CLASS	COUPLING	COMPLEXITY	LACK OF COHESION	SIZE	LOC	COMPLEXITY	COUPLING	LACK OF COHESION	SIZE
1	ApplicationContro...					186	low-medium	medium-high	low-medium	low-medium
2	FilterService					109	low-medium	low-medium	low	low-medium
3	TokenFilter					31	low-medium	low-medium	low	low
4	AuthorizationConfig					16	low-medium	low-medium	low	low
5	ApplicationService					78	low	low-medium	low	low-medium
6	ApplicationDataSe...					44	low	low-medium	low	low
7	SelectApplicantSe...					35	low	low-medium	low	low
8	InvalidApplicatio...					4	low-medium	low	low	low
9	EmptyTargetElemen...					4	low-medium	low	low	low

Figure 3.16: Application Microservice: After class refactoring

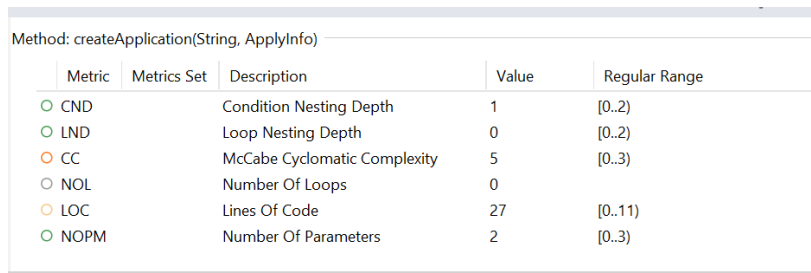
Chapter 4

Method Refactoring

Application Service - createApplication()

Problem Overview

After refactoring the class as a whole, we still needed to check if the method metrics were in accordance with our main goal. To do that, we used the MetricsTree plugin to analyze the methods of this service. At first sight, the method that creates an application was our first target since it had a high **McCabe's Cyclomatic Complexity of 5**, which exceeded the accepted threshold.



The screenshot shows a table titled 'Method: createApplication(String, ApplyInfo)' with the following data:

	Metric	Metrics Set	Description	Value	Regular Range
○	CND		Condition Nesting Depth	1	[0..2)
○	LND		Loop Nesting Depth	0	[0..2)
○	CC		McCabe Cyclomatic Complexity	5	[0..3)
○	NOL		Number Of Loops	0	
○	LOC		Lines Of Code	27	[0..11)
○	NOPM		Number Of Parameters	2	[0..3)

Figure 4.1: Method prior to refactoring

Refactoring Operations

For this method, we had three **if** statements that check the validity of the application and would throw an exception in case of invalidity. We decided to split the checks. The refactoring operations were as follows:

- Create **gradeRestriction()** method - this method checks if the grade for the course is greater than the general passing grade and throws the exception otherwise.
- Create **quarterRestriction()** method - this deals with the limited number of courses that a student can TA for in a quarter (i.e. 3). Otherwise, it throws the **InvalidApplicationException**.
- Create **applicationExists()** method - this checks if the application could be found by the given username and course code.

These refactoring operations brought the **Cyclomatic Complexity to 1** which is now even lower than the accepted threshold and average.

Method: createApplication(String, String, String)				
Metric	Metrics Set	Description	Value	Regular Ra...
<input type="radio"/> CND		Condition Nesting Depth	0	[0..2)
<input type="radio"/> LND		Loop Nesting Depth	0	[0..2)
<input type="radio"/> CC		McCabe Cyclomatic Complexity	1	[0..3)
<input type="radio"/> NOL		Number Of Loops	0	
<input type="radio"/> LOC		Lines Of Code	10	[0..11)
<input type="radio"/> NOPM		Number Of Parameters	3	[0..3)

Figure 4.2: Method after refactoring

Motivation

The main motivation behind the decision of refactoring this method was that with high complexity comes a higher probability of error with increased time for maintenance. We did not give up on the specific exceptions. The difference now is that we handle them separately and only throw them when needed. This makes the process of testing easier.

Application Service - autoReject()

Problem Overview

One method done - onto the other one! As a team, we decided to implement this extra useful feature of auto rejection given some minimum values. Sadly, it had a long parameter list which makes the method complicated and closed to extensions. Initially, the method took as input **six parameters**. Since a method is considered a long parameter list if it has more than four parameters, we concluded that this was also the case. Thus, we started the refactoring process.

Method: autoReject(String, String, String, String, String, String)				
Metric	Metrics Set	Description	Value	Regular Range
<input type="radio"/> CND		Condition Nesting Depth	1	[0..2)
<input type="radio"/> LND		Loop Nesting Depth	0	[0..2)
<input type="radio"/> CC		McCabe Cyclomatic Complexity	5	[0..3)
<input type="radio"/> NOL		Number Of Loops	0	
<input type="radio"/> LOC		Lines Of Code	41	[0..11)
<input type="radio"/> NOPM		Number Of Parameters	6	[0..3)

Figure 4.3: Method prior to refactoring

Refactoring Operations

As we have seen in the lectures, the solution was straightforward: we needed to create a Parameter Object, which would take some of the parameters and merge them into an object that shares the same logic. The refactoring operations were as follows:

- Create **FilterParameters** class - this class merges four parameters from the initial method: **minGrade**, **minRating**, **minAvgRating** and **minReqTa**.
- Update method signature - for each of the parameters in the data clump, remove it from the signature; update method callers and method body to use the new values.
- Update test cases accordingly.

These refactoring operations brought the number of parameters to **three parameters** which is now in accordance with the average.

Method: autoReject(String, FilterParameters, String)

Metric	Metrics Set	Description	Value	Regular Ra
<input checked="" type="radio"/> CND		Condition Nesting Depth	1	[0..2]
<input checked="" type="radio"/> LND		Loop Nesting Depth	1	[0..2]
<input checked="" type="radio"/> CC		McCabe Cyclomatic Complexity	5	[0..3]
<input type="radio"/> NOL		Number Of Loops	1	
<input checked="" type="radio"/> LOC		Lines Of Code	33	[0..11]
<input checked="" type="radio"/> NOPM		Number Of Parameters	3	[0..3]

Figure 4.4: Method after refactoring

Motivation

The main motivation behind the decision of refactoring this method was that we had four parameters that shared the same logic and deserved to be part of the same object. Having that many parameters made the extension of the method impossible. Thus, if we wanted to optimize it in the future by adding new parameters, we would have a very unreadable and poorly maintained method.

Application Service - applyAlgorithm()

Problem Overview

Finally, the method to apply the algorithm, which generates a grade recommendation based on specific conditions, had the same problem as the one above: long parameter list. The method signature was composed of **six parameters**, which was beyond our expectations. Thus, we started the refactoring process.

Method: applyAlgorithm(String, String, String, String, String, String)

Metric	Metrics Set	Description	Value	Regular Range
<input checked="" type="radio"/> CND		Condition Nesting Depth	1	[0..2]
<input checked="" type="radio"/> LND		Loop Nesting Depth	0	[0..2]
<input checked="" type="radio"/> CC		McCabe Cyclomatic Complexity	2	[0..3]
<input type="radio"/> NOL		Number Of Loops	0	
<input checked="" type="radio"/> LOC		Lines Of Code	22	[0..11]
<input checked="" type="radio"/> NOPM		Number Of Parameters	6	[0..3]

Figure 4.5: Method prior to refactoring

Refactoring Operations

Since we have already created a helper Parameter Class, we decided that we can further use it to optimize this code bad smell. The refactoring operations were as follows:

- Use **FilterParameters** class - substitute the four parameters with their newly created class.
- Update method signature - for each of the parameters in the data clump, remove it from the signature; update method callers and method body to use the new values.
- Update test cases accordingly.

These refactoring operations brought the number of parameters to **three parameters** which is now in accordance with the average.

Method: applyAlgorithm(String, FilterParameters, String)

	Metric	Metrics Set	Description	Value	Regular Ra.
<input checked="" type="radio"/>	CND		Condition Nesting Depth	1	[0..2)
<input checked="" type="radio"/>	LND		Loop Nesting Depth	0	[0..2)
<input checked="" type="radio"/>	CC		McCabe Cyclomatic Complexity	2	[0..3)
<input type="radio"/>	NOL		Number Of Loops	0	
<input type="radio"/>	LOC		Lines Of Code	22	[0..11)
<input type="radio"/>	NOPM		Number Of Parameters	3	[0..3)

Figure 4.6: Method after refactoring

Motivation

The main motivation behind the decision of refactoring this method was that again it was blocking the addition of new parameters. With the updated method, our system is now ready for extra features and functionalities!

TA Service - Constructor

Problem Overview

While we have refactored the TA Service already, the number of dependencies within this service is still relatively high. Several repository classes are used to access the different data tables, and communicator classes are used to communicate with other microservices. This totals 7 dependency injections for the constructor of the TA Service.

Method: TaService(TaRepository, ContractRepository, WorkloadRepository, ReviewRepository, CourseCommunicator, ...)

	Metric	Metrics Set	Description	Value	Regular Ra...
<input checked="" type="radio"/>	CND		Condition Nesting Depth	0	[0..2)
<input checked="" type="radio"/>	LND		Loop Nesting Depth	0	[0..2)
<input checked="" type="radio"/>	CC		McCabe Cyclomatic Complexity	1	[0..3)
<input type="radio"/>	NOL		Number Of Loops	0	
<input type="radio"/>	LOC		Lines Of Code	28	[0..11)
<input type="radio"/>	NOPM		Number Of Parameters	7	[0..3)

Figure 4.7: Method prior to refactoring

Refactoring Operations

We intend on reducing the number of method parameters. However, we cannot target the repository classes since they handle independent tables. Therefore, we combined the communicator class into **one** class - Central Communicator. The refactoring operations are as follows:

- Transfer all communication methods from **Authentication Communicator**, **Course Communicator**, and **Notification Communicator** to the **Central Communicator**.
- Remove the previous communicator classes as dependencies from the TA Service. Add the Central Communicator as the only dependency related to communicating with external microservices.

This reduces the number method parameters from 7 to 4. It also centralises the communication point with external microservices.

Method: TaService(TaRepository, ContractRepository, WorkloadRepository, CentralCommunicator)					
	Metric	Metrics Set	Description	Value	Regular Ra.
	CND		Condition Nesting Depth	0	[0..2)
	LND		Loop Nesting Depth	0	[0..2)
	CC		McCabe Cyclomatic Complexity	1	[0..3)
	NOL		Number Of Loops	0	
	LOC		Lines Of Code	17	[0..11)
	NOPM		Number Of Parameters	4	[0..3)

Figure 4.8: Method after refactoring

Motivation

This method refactoring operation allows for fewer method parameters. Moreover, since it combined the communicator classes, it also reduces the number of dependency injections. Combining the communicator classes is also justified since no communicator class had more than three methods and the complexity of the **Central Communicator** remains low.

TA Service - averageTa()

Problem Overview

When further analyzing this service, trying to eliminate all code smells, the method that calculates the average TA hours drew our attention for a high **McCabe's Cyclomatic Complexity of 5**. In contrast to all other methods, the complexity was beyond average (i.e. around 2-3 Complexity), so the decision was fairly easy: we had to refactor this method to decrease McCabe's number.







Method: averageTa(String, String)					
	Metric	Metrics Set	Description	Value	Regular Ra...
	 CND		Condition Nesting Depth	1	[0..2)
	 LND		Loop Nesting Depth	1	[0..2)
	 CC		McCabe Cyclomatic Complexity	5	[0..3)
	 NOL		Number Of Loops	1	
	 LOC		Lines Of Code	23	[0..11)
	 NOPM		Number Of Parameters	2	[0..3)

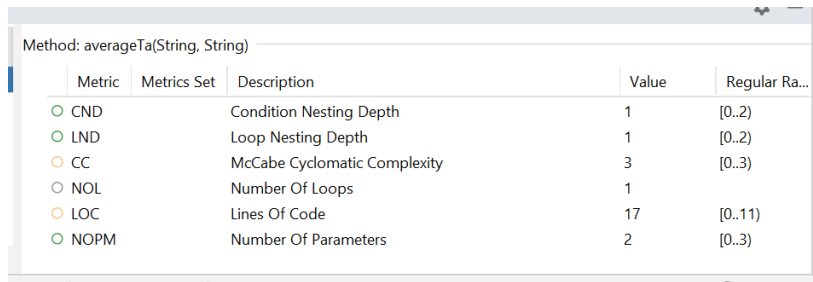
Figure 4.9: Method prior to refactoring

Refactoring Operations

In order to get to a lower complexity, we had to draw the control flow graph. From the graph, we immediately figured that we had two conditions that were not supposed to be there. They were throwing some custom errors, but they would not have been reached either way because of the condition. The refactoring operations were as follows:

- Remove if statement checking if the workload object is empty - this would not have been the case since we took into consideration only TAs that actually worked for that specific course.
- Remove if statement checking if the average hours are equal to zero - this would only be possible if the TA did not work at all or also had negative hours, which is not possible for our system.

These refactoring operations brought the **Cyclomatic Complexity to 3** which is now in accordance with the average.



	Metric	Metrics Set	Description	Value	Regular Ra...
<input type="radio"/>	CND		Condition Nesting Depth	1	[0..2)
<input type="radio"/>	LND		Loop Nesting Depth	1	[0..2)
<input type="radio"/>	CC		McCabe Cyclomatic Complexity	3	[0..3)
<input type="radio"/>	NOL		Number Of Loops	1	
<input type="radio"/>	LOC		Lines Of Code	17	[0..11)
<input type="radio"/>	NOPM		Number Of Parameters	2	[0..3)

Figure 4.10: Method after refactoring

Motivation

We believe that the key to an efficient and long-lasting system is represented by low cyclomatic complexity. The main motivation behind the decision of refactoring this method was that we had paths in the graph that will never be accessed, and this meant useless test cases which increase the overall complexity of the product. The new McCabe's number makes it easier to achieve better test branch coverage.

References

1. <https://martinfowler.com/articles/developer-effectiveness.html>
2. <https://refactoring.guru/smells/large-class>
3. <https://medium.com/clarityhub/low-coupling-high-cohesion-3610e35ac4a6>
4. https://checkstyle.sourceforge.io/config_sizes.htmlParameterNumber