

Assignment 2

SEM group 34B

Introduction

For assignment 2 we as a group assessed the quality of our code in the light of a set of metrics, and subsequently applied refactoring operations to improve these metrics.

The metrics we used were calculated by an IntelliJ IDEA plugin called 'MetricsTree', this plugin gave us a quantifiable measure of code quality both on method and class levels.

Selected methods / classes

We selected the following methods to be refactored:

- joinTraining(), a method of the TrainingService class
- joinCompetition(), a method of the CompetitionService class
- checkGender(), a method of the CompetitionService class
- getSuitableCompetition(), a method of the CompetitionService class
- sendDecisionOfOwnerToRequester(), a method of the UserController class

We also selected the following classes to be refactored:

- The User class
- The UserController class
- The CompetitionService class
- The TrainingService class
- The TrainingController class

The operations we applied to these classes / methods and the reasoning behind the changes will be discussed in further detail in the [refactoring section](#).

Thresholds

Classes

- For the Weighted Methods per Class metric, we used a threshold of twelve (for regular range), thirty-five (for high range) ,and forty-five (very high range) (source 2).
- For the Number of Accessor Methods per Class metric, we used a threshold of four, seven and thirteen (for regular, high, very-high ranges respectively).

- For the Coupling between Objects metric, we used a threshold of fourteen, seventeen and twenty-three (for the regular, high and very-high ranges respectively) (source 1).
- For the Lack of Cohesion (LCOM), we decided to use both CodeMR and the Metrics Tree for reference. We chose a threshold of 1 (source 3), as it represents a cohesive class. Anything above 2 or equal to 2 represents a problem in cohesion (source 3). For example, initially the TrainingController had an index of 2 and after refactoring, the splitted classes have an index of 1, thus a clear improvement (these scores were shown on Metrics Tree; notice that CodeMR showed pie charts, for which we took screenshots as you can see below. Green is of course a score of 1) .

Methods

- For the CC (cyclomatic complexity), we set the threshold to be 7: anything above 7 needs to be changed as it has too many branches. The joinCompetition method for example had very high CC (11), but it was also quite a long and complicated method on its own; we definitely had to lower its CC. Other methods however, didn't necessarily have a CC over 7. But in the context of their functionality, they had some of the highest branches per line of code, and considering the scope of them, lowering the CC seemed optimal. One such method was checkGender. The threshold of 7 was chosen because it had a very score (source 1). The threshold of 5 is high (source 1), so everything below it is optimal.
- For the LOC (lines of code), we decided to set the threshold to 20, as 20 is considered to be too many lines of code (source 1). In our screenshots, we include javadocs as lines of code, but the actual metric shouldn't do that. For example, one method which had many lines of code was sendDecisionOfOwnerToRequester so we reduced the number of lines, as we will show below.

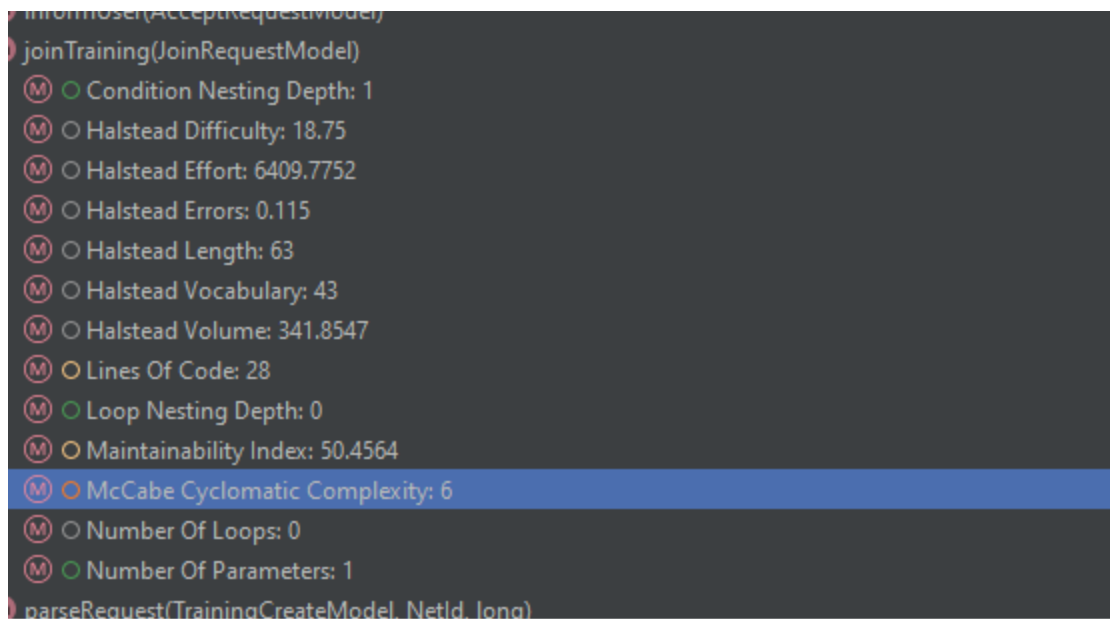
Refactoring

Methods

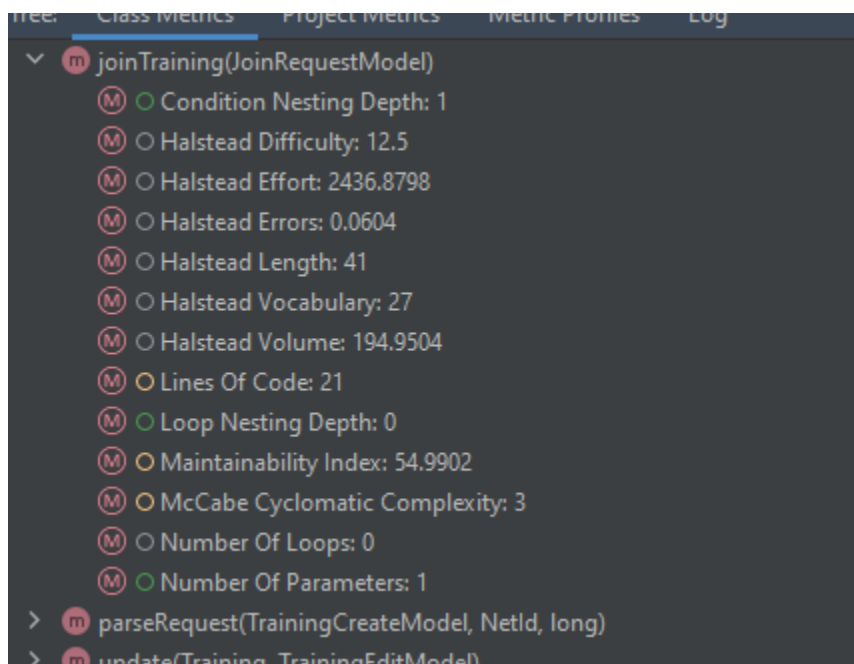
joinTraining

In class Training Service, CC for the joinTraining() is too high. The reason is that there are so many conditions in this method. Refactor code in method joinTraining(), what I did is that I created another method just for checking and moved the if-condition into that method.

Before:



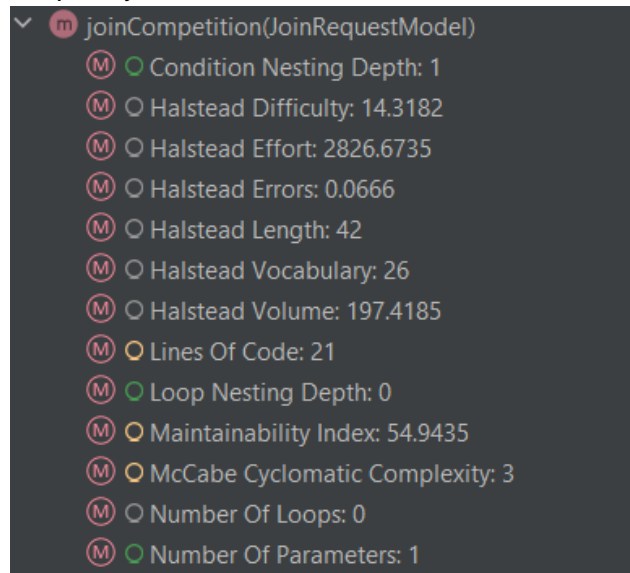
After:



joinCompetition

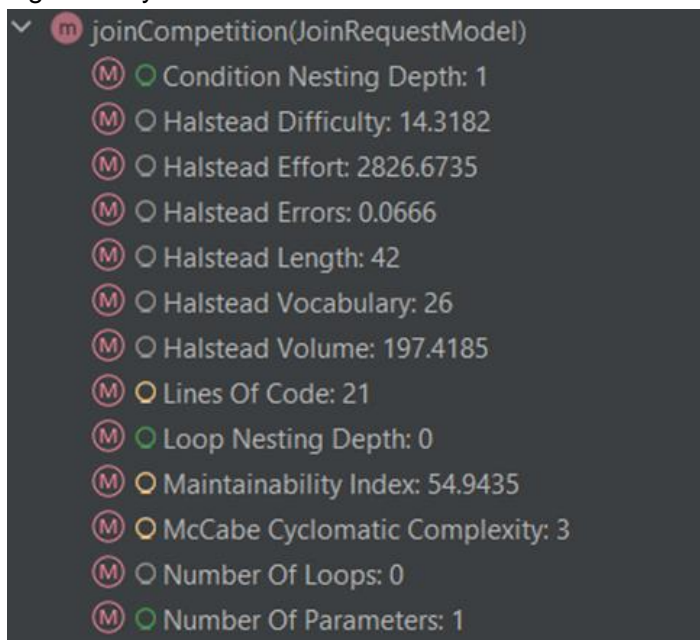
The joinCompetition method is a method which has an extremely high cyclomatic complexity of 11. This is a critical issue that needs to be addressed, and hence we decided to refactor this method. The reason joinCompetition had such a high cyclomatic complexity was the fact that there were a lot of criteria checks to be completed - the competition existing, the competition not starting too close to application, and the user data related checks - the user data existing (and not null), the user having required certification (if they applied as a COX), and the user's details matching the competition creator's set constraints.

As can be seen in the following screenshot, MetricsTree highlighted that we had a cyclomatic complexity of 11.



In order to refactor this method, we created separate methods that evaluate specific checks - a method to check validity of user data, one to check certification, one to check constraints, and on the competition side, methods to check competition validity, and a valid start time.

After this change, the cyclomatic complexity, as seen in the following screenshot, reduces very significantly to 3



This change was important to make, since it drastically improved the cyclomatic complexity metric, which was at an extreme level of 11 prior to the change.

checkGender

The checkGender method is a method which has high cyclomatic complexity, though not the highest. Nonetheless, we decided that it would be a good idea to change this, because it had only 5 lines of actual code, but 2 if statements (one of the highest branches per lines of code in the whole project, if you also include the fact that there were also additional conditions in the ifs using ands). Not only that, but this method is supposed to be a simple checker, thus having so many possible paths is impractical and shows bad design. What we did was to modify the enums behind the method: Gender and GenderConstraint. Each enum would also have a value field, and this way, instead of checking the enums to be identical 2 times (for example, if we need to check that there is a constraint and only females can participate, we need to check that the gender is female and that the gender constraint is female as well), we would simply check whether their value is equal. Of course, having no constraint at all would be a separate if statement, thus the cyclomatic complexity of 2. This is what the Metrics Tree showed before the refactoring:

Method: checkGender(Gender, GenderConstraint)

	Metric	Metrics Set	Description	Value	Regular Ra...
○	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	16	[0..11)
○	NOPM		Number Of Parameters	2	[0..3)
○	HVL	Halstead M...	Halstead Volume	98.9912	
○	HD	Halstead M...	Halstead Difficulty	4.5	
○	HL	Halstead M...	Halstead Length	26	
○	HEF	Halstead M...	Halstead Effort	445.4605	
○	HVC	Halstead M...	Halstead Vocabulary	14	
○	HER	Halstead M...	Halstead Errors	0.0194	
○	MMI	Maintainab...	Maintainability Index	59.5743	[0.0..19.0]

This is what the Metrics Tree showed after the refactoring:

Method: checkGender(Gender, GenderConstraint)					
	Metric	Metrics Set	Description	Value	Regular Ra...
	○ CND		Condition Nesting Depth	1	[0..2)
	○ LND		Loop Nesting Depth	0	[0..2)
	○ CC		McCabe Cyclomatic Complexity	2	[0..3)
	○ NOL		Number Of Loops	0	
	○ LOC		Lines Of Code	14	[0..11)
	○ NOPM		Number Of Parameters	2	[0..3)
	○ HVL	Halstead M...	Halstead Volume	62.9075	
	○ HD	Halstead M...	Halstead Difficulty	3.8571	
	○ HL	Halstead M...	Halstead Length	17	
	○ HEF	Halstead M...	Halstead Effort	242.6431	
	○ HVC	Halstead M...	Halstead Vocabulary	13	
	○ HER	Halstead M...	Halstead Errors	0.013	
	○ MMI	Maintainab...	Maintainability Index	62.3548	[0.0..19.0]

Despite the fact that the cyclomatic complexity in the first place was not extreme, in our case, a cyclomatic complexity of 2 is definitely desirable (taking into account the actual size of the method and its purpose).

getSuitableCompetition

The problem was that in the following metric, the Cyclomatic Complexity of this method is high. We used the threshold of three, supported by “A Catalogue of Thresholds for Object-Oriented Software Metrics”.

For the refactoring, we separated the checks of amateur and organization to make them separate methods.

McCabe Cyclomatic Complexity

Calculated Metrics Value: ○ 4

Metrics Level: Method Level

Regular Range: [0..3)

Metrics Set:

McCabe Cyclomatic Complexity (CC)

McCabe Cyclomatic Complexity (CC) is a measure of the control structure complexity of software. It is the number of linearly independent paths and therefore, the minimum number of independent paths when executing the software.

And we now have the value of 2, which is lower than the threshold mentioned above.

	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 checked="" type="radio"/>	LOC		Lines Of Code	26	[0..11)
<input checked="" type="radio"/>	NOPM		Number Of Parameters	1	[0..3)
<input type="radio"/>	HVL	Halstead M...	Halstead Volume	309.2971	
<input type="radio"/>	HD	Halstead M...	Halstead Difficulty	23.8214	
<input type="radio"/>	HI	Halstead M...	Halstead Length	57	

sendDecisionOfOwnerToRequester

The problem with this method is that it was too long (high LOC) and too complex (high CC). We define the CC as too high when it exceeds 3, this might seem low, but this method is in a controller and methods in a controller should not handle any complex logic. Controller methods are designed to receive requests and delegate the handling of the request to a service, thus this is why we want such a low CC.

Furthermore we define a high LOC in this particular case as being anything above 11, again this might seem low, but the reasoning for this is the same as for the CC.

Before our refactoring, these were the values of our metrics:

Method: sendDecisionOfOwnerToRequester(UserAcceptanceUpdateModel)					
	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	3	[0..3)
<input type="radio"/>	NOL		Number Of Loops	0	
<input checked="" type="radio"/>	LOC		Lines Of Code	33	[0..11)
<input checked="" type="radio"/>	NOPM		Number Of Parameters	1	[0..3)
<input type="radio"/>	HVL	Halstead M...	Halstead Volume	322.8473	

To fix this, a part of the method pertaining to creating a Message object from the model we received was delegated to the UserService. This new method is called

saveAcceptanceMessage. By Doing this we still adhere to the principle of letting the services do most of the logic.

This reduced our CC and LOC to the following values below our threshold.

Method: sendDecisionOfOwnerToRequester(UserAcceptanceUpdateModel)

	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	2	[0..3)
<input type="radio"/>	NOL		Number Of Loops	0	
<input checked="" type="radio"/>	LOC		Lines Of Code	10	[0..11)
<input checked="" type="radio"/>	NOPM		Number Of Parameters	1	[0..3)
<input type="radio"/>	UVM	Unlabeled M...	Unlabeled Value...	74.3300	

Classes

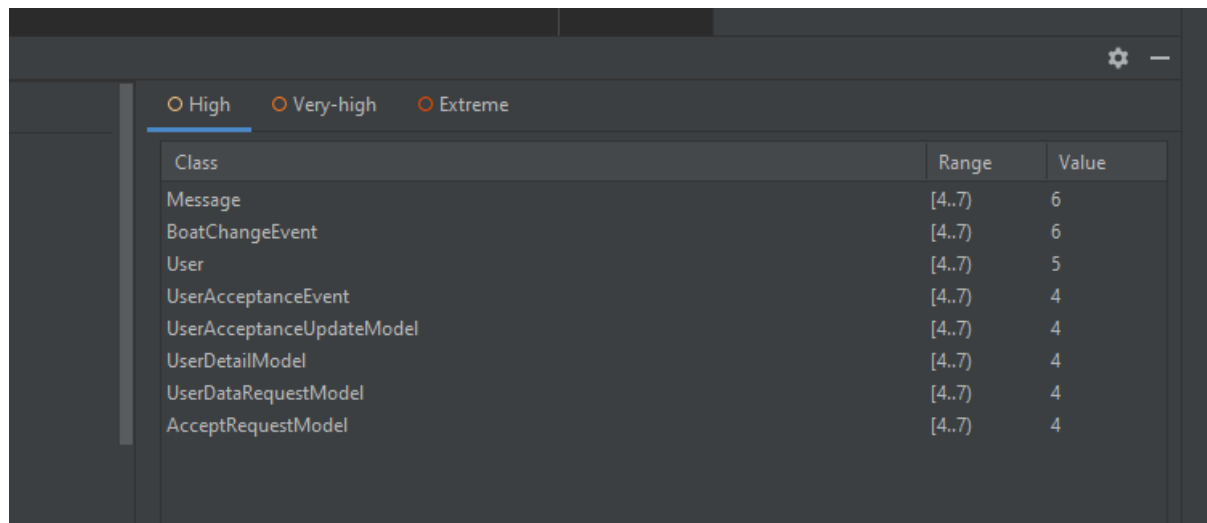
User

In class User, the number of accessor methods metrics for this class is too high. The reason is that there are so many getters and setters in the class and we don't actually use these methods. Refactor code in class User: what I did is that I check which method is not used I deleted that one.

Before:

			⚙	—
<input type="radio"/> High <input checked="" type="radio"/> Very-high <input type="radio"/> Extreme				
Class		Range	Value	
User		[7..13)	10	
Competition		[7..13)	8	
CompetitionEditModel		[7..13)	7	
CompetitionCreateModel		[7..13)	7	

After:

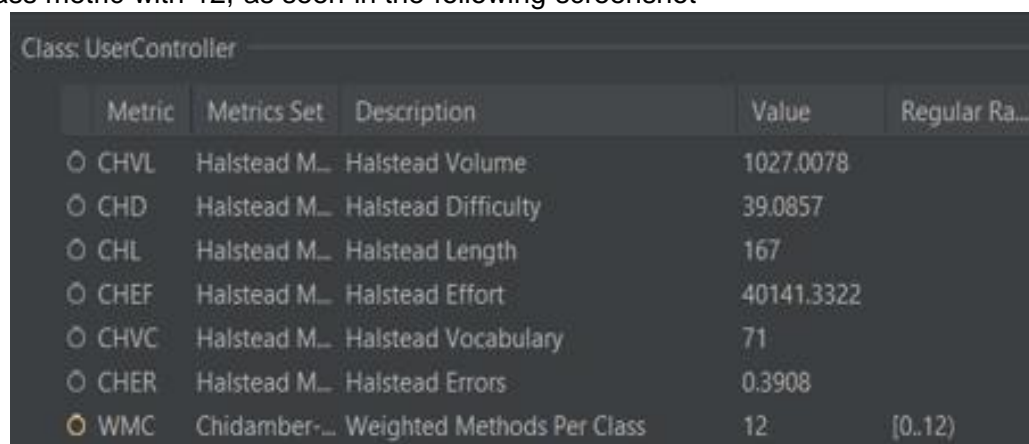


Class	Range	Value
Message	[4..7]	6
BoatChangeEvent	[4..7]	6
User	[4..7]	5
UserAcceptanceEvent	[4..7]	4
UserAcceptanceUpdateModel	[4..7]	4
UserDetailModel	[4..7]	4
UserDataRequestModel	[4..7]	4
AcceptRequestModel	[4..7]	4

UserController

Initially, the UserController class was a class that took care of every single user-related aspect. It was invoked to create a user, find the user given the details in a model, send the application message to the activity owner when a participant decides to partake, send the application result message to the participant when the activity owner has made their decision, and to retrieve the notifications 'inbox' of the user with all the messages they retrieved so far.

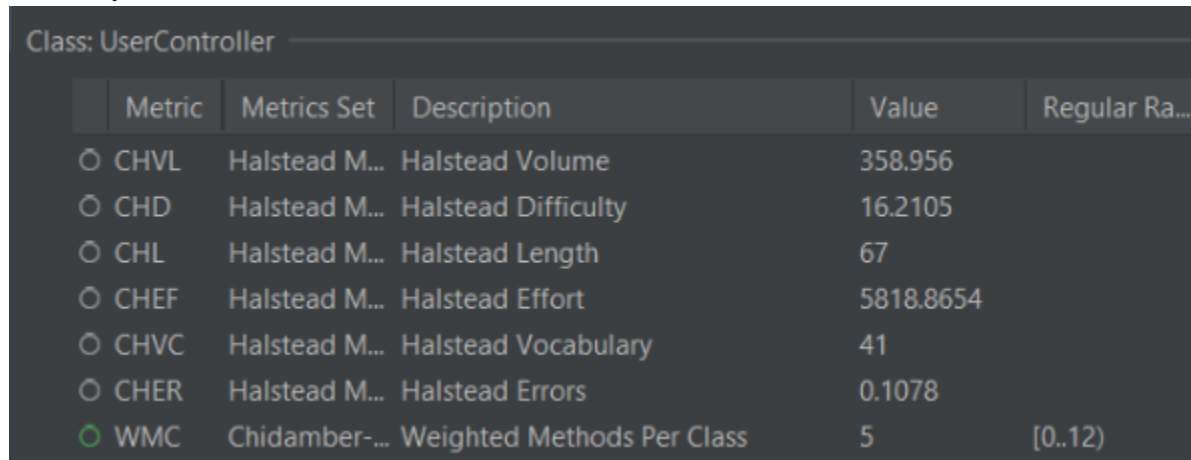
In this case, we opted for the metric of Weighted Methods per Class (WMC). As a result of the overarching characteristic of the UserController class, it suffered from a high Weighted Methods per Class metric with 12, as seen in the following screenshot



Metric	Metrics Set	Description	Value	Regular Ra...
CHVL	Halstead M...	Halstead Volume	1027.0078	
CHD	Halstead M...	Halstead Difficulty	39.0857	
CHL	Halstead M...	Halstead Length	167	
CHEF	Halstead M...	Halstead Effort	40141.3322	
CHVC	Halstead M...	Halstead Vocabulary	71	
CHER	Halstead M...	Halstead Errors	0.3908	
WMC	Chidamber-...	Weighted Methods Per Class	12	[0..12)

So, it was decided that the UserController class be split into 2 classes, with a new class 'NotificationController' emerging out of it. The notification controller handled methods relating to the delivery of messages between the activity owner and participant, both ways - as well as for the user to view their inbox. The UserController class now would only have essential methods - the method to fill in the details of the user, and the method to retrieve the details of the user (as used by the activity microservice).

As seen in the following screenshot, the WMC metric of the UserController class has reduced to a healthy 5.



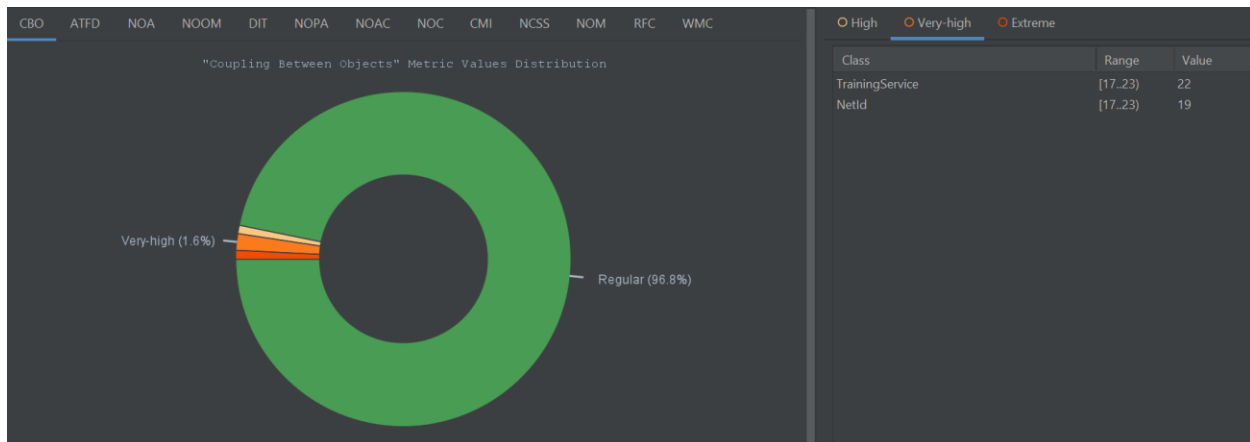
The screenshot shows a table of metrics for the class 'UserController'. The WMC (Weighted Methods Per Class) metric is highlighted with a green circle and a value of 5, which is within the regular range of [0..12].

Class: UserController					
	Metric	Metrics Set	Description	Value	Regular Ra...
<input type="radio"/>	CHVL	Halstead M...	Halstead Volume	358.956	
<input type="radio"/>	CHD	Halstead M...	Halstead Difficulty	16.2105	
<input type="radio"/>	CHL	Halstead M...	Halstead Length	67	
<input type="radio"/>	CHEF	Halstead M...	Halstead Effort	5818.8654	
<input type="radio"/>	CHVC	Halstead M...	Halstead Vocabulary	41	
<input type="radio"/>	CHER	Halstead M...	Halstead Errors	0.1078	
<input checked="" type="radio"/>	WMC	Chidamber-...	Weighted Methods Per Class	5	[0..12)

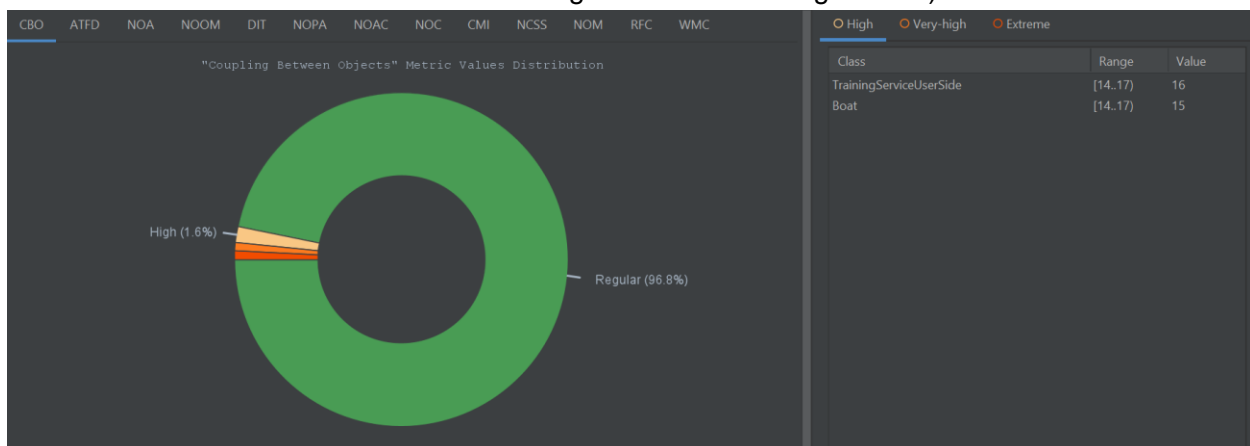
TrainingService

Both the TrainingService and CompetitionService are classes which have very high (or extreme coupling). In this section we will focus on the TrainingService refactoring.

This class adds all of the functionality for a user to join, create, delete etc a training activity. Not only that the code is quite long on its own (having 220 lines of code) but it handles the server part of this service (of creating/deleting competitions) as well as the user part (showing the user the information he wants on the front-end side of the app). Doing multiple things in a class is not a good idea, as this also adds to the lack of cohesion. So we decided to split the class into 2: the TrainingServiceUserSide, which is responsible for showing the user the available trainings, as well as informing the user of joining a certain training (which would directly communicate with the front-end) and the TrainingServiceServerSide, which is directly responsible for changing the database (creating/deleting/modifying trainings). This way, the 2 classes have a clear scope and each has a lower CBO score. We notice that 2 fields (the eventPublisher and the currentTimeProvider) are not needed on the server side are not needed, so we can remove them (which further decreases the coupling index in this class). This is the Metrics Tree before the refactoring (notice that this screenshot was taken on the training service's branch, therefore the CompetitionService class score is still extreme):



This is the Metrics Tree after the refactoring (notice that the CompetitionService is still extreme, as this screenshot was taken on the TrainingService refactoring branch):



Notice that the TrainingServiceServerSide doesn't even appear on the high category in the pie chart (as it has a value below 14). The user side has a value of 16. Splitting the user side furthermore to decrease the coupling index would be impractical, as creating or deleting a boat are closely related and there would be a lot of code duplication. A service is also supposed to have a higher index for coupling compared to other classes, because many methods from different classes are called there: the main logic resides there and it connects the controllers to the repositories. Thus, in our case, an index of 16 is more than desirable.

TrainingController

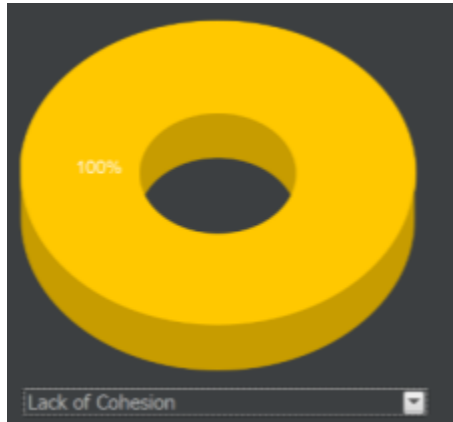
The TrainingController class had both a medium-high coupling, as well as a medium-high lack of cohesion.

This class employed the use of three other classes: AuthManager, TrainingServiceUserSide and TrainingServiceUserSide. To decrease both of the aforementioned metrics, we decided to split the class into three other controller classes: TrainingControllerCreate, TrainingControllerEdit and TrainingControllerUser, each containing the original functionalities from TrainingController dependent on the UserSide and ServerSide TrainingServices respectively.

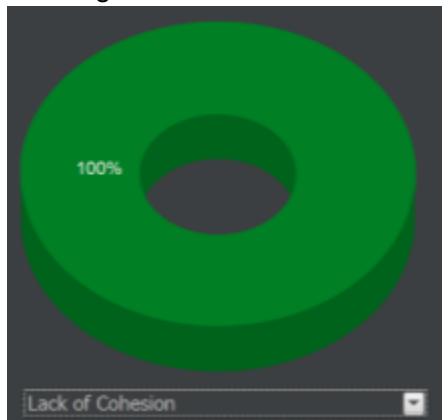
By dividing the functionalities into two different classes, it meant that the two resulting classes would not need as many dependencies (which means lower coupling), as well as being more focused on performing a smaller number of tasks (which means higher cohesion).

The result, as expected, was a decrease in both the coupling and the lack of cohesion. More exactly, the three final controller classes now have low/medium-low coupling and lack of cohesion, as opposed to the original medium-high.

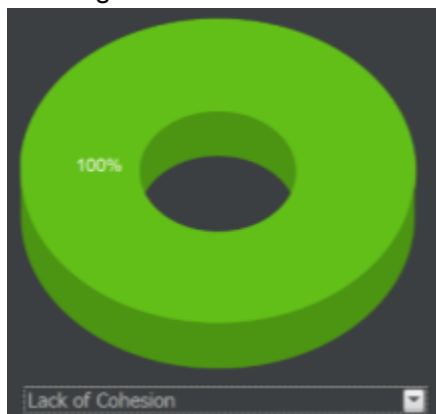
Here we can see the TrainingController class' lack of cohesion chart (from CodeMR) before the refactoring (note that these changes also lower the CBO index):



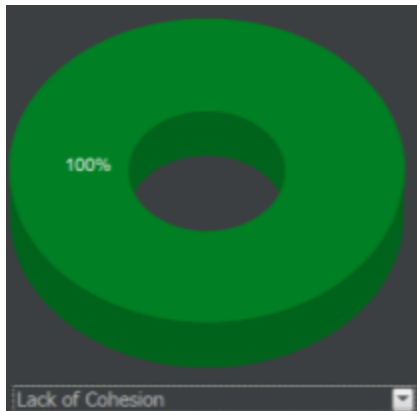
Here we can see the improvement for every single newly created class:
TrainingControllerCreate



TrainingControllerEdit:



TrainingControllerUser:



CompetitionService

As we have mentioned above, competitionService has an extremely high coupling score. In this part, we will shrink the coupling of competitionService. (The picture below shows that the score is 25, which is considered as level “extreme” by MetricsTree)

<input type="radio"/> High	<input type="radio"/> Very-high	<input checked="" type="radio"/> Extreme
Class	Range	Value
CompetitionService	[23..∞)	25

To solve this problem, we set up the threshold of 18, according to the paper “A comparison and evaluation of variants in the coupling between objects metric” written by Mike Child, Peter Posner, and Steve Cunsell. For the part of actually solving the problem, we choose to split the service into two parts: the user side one which includes all the user-related methods and the server side one which only includes the service that happens in the server side. By implementing our refactoring method, we managed to get rid of the extreme coupling problem of the competitionService. The pictures below show the new coupling score of the two new service classes.

☐ High☒ Very-high☐ Extreme

Class	Range	Value
CompetitionServiceServerSide	[14..17)	15
Boat	[14..17)	15

28:2 CRLF UTF-8 4 spaces refactor competitionService

☐ High☒ Very-high☐ Extreme

Class	Range	Value
TrainingService	[17..23)	22
NetId	[17..23)	19
CompetitionServiceUserSide	[17..23)	18

31:2 CRLF UTF-8 4 spaces refactor competitionService

Sources

1. Ferreira, K. *A catalogue of thresholds for object-oriented software metrics*. An analysis of threshold values in Software Engineering metrics. 2015. *thinkmind*,
https://www.thinkmind.org/articles/softeng_2015_3_10_55070.pdf.
2. S. R. Chidamber and C. F. Kemerer, "A metrics suite for object-oriented design," in IEEE Transactions on Software Engineering, vol. 20, no. 6, pp. 476-493, June 1994, doi: 10.1109/32.295895.
3. <https://objectscriptquality.com/docs/metrics/lack-cohesion-methods-lcom4>