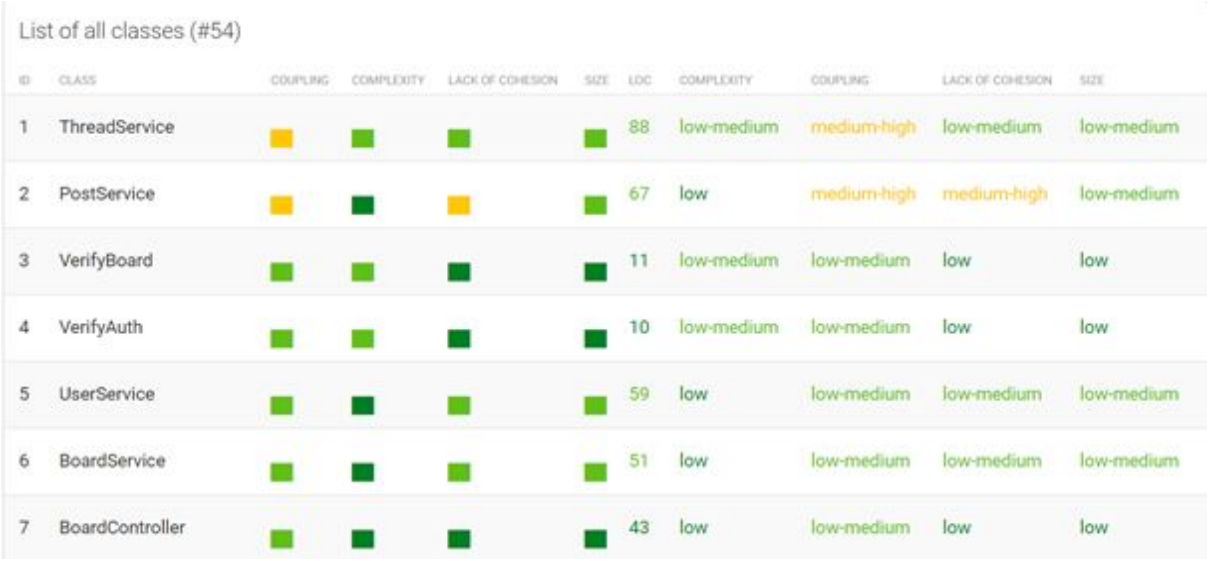
Assignment 2

OP10 - SEM 20

1. Code Analysis Tool

For the first part we use *CodeMR* and *Metric Reloaded* to identify the problematic classes and methods. The analysis from both tools rated the code quality of the whole project highly, making it quite troublesome to choose and identify the best way to continue improving the codebase. Regardless of that, we were still able to slightly improve the ratings on the classes and methods we list further below. The reasoning behind our refactoring of the class is also included below each of the class and methods chosen to be refactored.

Before proceeding with the selected classes and methods, the image and table below shows an overview of the metrics of our classes and methods



List of all classes (#54)

ID	CLASS	COUPLING	COMPLEXITY	LACK OF COHESION	SIZE	LOC	COMPLEXITY	COUPLING	LACK OF COHESION	SIZE
1	ThreadService					88	low-medium	medium-high	low-medium	low-medium
2	PostService					67	low	medium-high	medium-high	low-medium
3	VerifyBoard					11	low-medium	low-medium	low	low
4	VerifyAuth					10	low-medium	low-medium	low	low
5	UserService					59	low	low-medium	low-medium	low-medium
6	BoardService					51	low	low-medium	low-medium	low-medium
7	BoardController					43	low	low-medium	low	low
8	UserController					21	low	low-medium	low	low
9	PostNotFoundExcep...					6	medium-high	low	low	low
10	BoardThreadNotFou...					6	medium-high	low	low	low
11	ThreadIsLockedExc...					6	medium-high	low	low	low
12	BoardIsLockedExce...					6	medium-high	low	low	low
13	BoardNotFoundExce...					6	medium-high	low	low	low
14	AuthorizationFail...					6	medium-high	low	low	low
15	PermissionException					6	medium-high	low	low	low
16	UserNotFoundExcep...					5	medium-high	low	low	low
17	BoardThread					111	low-medium	low	high	low-medium
18	Board					90	low-medium	low	medium-high	low-medium

19	Post					87	low-medium	low	medium-high	low-medium
20	AuthToken					47	low-medium	low	low-medium	low
21	AuthResponse					18	low-medium	low	low	low
22	EditBoardThreadRe...					16	low-medium	low	low	low
23	EditBoardRequest					15	low-medium	low	low	low
24	IsLockedResponse					12	low-medium	low	low	low
25	EditPostRequest					9	low-medium	low	low	low
26	LocalDateTimeDese...					6	low-medium	low	low	low
27	LocalDateTimeSeri...					6	low-medium	low	low	low
28	User					67	low	low	low-medium	low-medium
29	RegisterRequest					57	low	low	low-medium	low-medium
30	ThreadController					42	low	low	low	low
31	StatusResponse					37	low	low	low-medium	low
32	LoginRequest					37	low	low	low	low
33	PostController					36	low	low	low	low
34	ContentController...					23	low	low	low	low
35	CreateBoardThread...					20	low	low	low	low
36	IsLockedRequest					16	low	low	low	low
37	CreateBoardRequest					14	low	low	low	low
38	CreatePostRequest					14	low	low	low	low
39	CheckRequest					13	low	low	low	low
40	HandlerBuilder					12	low	low	low	low

An overview of the chosen methods. Method list was too long so we only show a limited list sorted by highest ev(G):

Method	ev(G)	iv(G)	v(G)
nl.tudelft.sem.group20.authenticationserver.services.UserService.login(String,String)	5.0	5.0	5.0
nl.tudelft.sem.group20.authenticationserver.entities.AuthToken.equals(Object)	4.0	2.0	5.0
nl.tudelft.sem.group20.boardserver.services.BoardService.updateBoard(EditBoardRequest,String)	4.0	1.0	4.0
nl.tudelft.sem.group20.contentserver.services.PostService.createPost(String,CreatePostRequest)	4.0	2.0	5.0
nl.tudelft.sem.group20.authenticationserver.embeddable.LoginRequest.equals(Object)	3.0	3.0	5.0
nl.tudelft.sem.group20.authenticationserver.embeddable.RegisterRequest.equals(Object)	3.0	5.0	7.0
nl.tudelft.sem.group20.authenticationserver.entities.User.equals(Object)	3.0	6.0	8.0
nl.tudelft.sem.group20.authenticationserver.services.UserService.createUser(RegisterRequest)	3.0	2.0	3.0
nl.tudelft.sem.group20.boardserver.entities.Board.equals(Object)	3.0	2.0	4.0

Now we proceed to explain why we selected some classes and methods to be refactored.

Chosen Classes

1. *BoardThread*
    - This class was the worst rated withinCodeMR identifying a high lack of cohesion. This means that the elements do not interact with each other. Methods for example do not use the same attributes.This is mostly because this class is mainly a holder for data and has very little logic in it.

2. *Post*
    - This class showed a medium-high lack of cohesion. This class was chosen since it can be refactored together with the *BoardThread.*

3. *ThreadService*
    - This class exhibited a medium-high level of coupling. As this can have negative effects on code maintenance, such as ripple effects when changing things in classes it is dependent on, we felt this was something we should try to change.

4. *PostThreadService*
    - This class exhibited a medium-high level of coupling and lack of cohesion. We chose this class as well since it can be refactored together with BoardThreadSerice.

5. Custom exceptions handlers we created
    - All our custom exception handles rated medium-high in complexity. As these classes are not that complex we wanted to see if we could change them somehow. We already had a fix in mind when looking at the code and want to try this out.

6. *Board*
    - This entity class had medium-high lack of cohesion and low-medium size, due to the large number of fields and methods that did not have any common attributes, such as getters and setters. As medium-high is one of the highest metrics we have gotten, we wanted to try and improve this.

Chosen Methods

1. AuthenticationService.login
    - This method has a complexity of five. This is too high and makes the code harder to read and more susceptible to bugs. Also fixing this prevents methods from becoming black boxes over time. We will fix this by splitting the method into two smaller methods.
2. BoardService.updateBoard
    - This method has a complexity of four. This is too high and makes the code harder to read and more susceptible to bugs. Also fixing this prevents methods from becoming black boxes over time. We will fix this by splitting the method into two smaller methods.
3. PostService.createPost
    - This method has a complexity of four. This is too high and makes the code harder to read and more susceptible to bugs. Also fixing this prevents methods from becoming black boxes over time. We will fix this by splitting the method into two smaller methods.
4. BoardService.createBoard
    - This method is refactored based on the class itself having medium-high coupling. This means that the method depends too much on other classes. To fix this problem we will extract the code that they have in common and create one method for it.
5. PostService.createPost
    - This method is refactored based on the class itself having medium-high coupling. This means that the method depends too much on other classes. To fix this problem we will extract the code that they have in common and create one method for it.

2. Code Refactoring

Now we will proceed to explain how each class and method was refactored, and how it improved its score in one of the code analysis tools.

2.1 Class Refactoring

BoardThread and Post class Refactoring

First we realized that both *BoardThread* and *Post* resided in the same microservice and shared similar parameters such as: *body information*, *id*, *creation date*, *edited date and username of creator*. This led us to believe that we could create an abstract class from which both of these classes inherited the similar parameters and methods. We proceeded by creating the abstract *@MappedSuperClass* named *Content* which contained the shared parameters, and shared methods such as the getters, setters and hash functions. After the creation we refactored the *BoardThread* and *Post* classes to inherit from this abstract class. With this refactoring we shrunk the size of each class by half, and reduced the complexity of each class with the shared methods. This also improved the metrics of CodeMR as shown in the image below.

As we can see, BoardThread went from high lack of cohesion to medium-high lack of cohesion, while the Post went from low-medium size to low size. We also wish to highlight that these classes are simple data holders, hence the lack of cohesion can be attributed to the fact that these classes don't have much logic within them - rather they help us efficiently store our information and represent the content of our databases.

- Before

ID	CLASS	COUPLING	COMPLEXITY	LACK OF COHESION	SIZE	LOC	COMPLEXITY	COUPLING	LACK OF COHESION	SIZE
17	BoardThread					107	low-medium	low	high	low-medium
19	Post					87	low-medium	low	medium-high	low-medium

- After

18	BoardThread					74	low-medium	low	medium-high	low-medium
19	Post					50	low-medium	low	medium-high	low

Refactoring of Board class

As a class which corresponds to a Spring entity, the *Board* had too many fields and very few methods which shared an attribute. Keeping in mind that the Board Server might be extended in the future to include additional entities (such as a *Course*, for instance), we thought that creating a @MappedSuperclass as a parent for the *Board* class might make the microservice more scalable.

To reduce lack of cohesion in the *Boar*d class, we created two additional superclasses using Spring's @MappedSuperclass annotation: the *BaseEntity* and the *ExtendedBaseEntity*. The *BaseEntity* can be more easily generalized, since it contains only two fields: an *id* and a *name* (most entities would require these two fields). The ExtendedBaseEntity inherits from this class, and contains an embedded TimestampTracker, a separate class with two fields: *created* and *edited*. The Board extends the ExtendedBaseEntity and has additional attributes.

After refactoring, the *Board* class has low-medium lack of cohesion. However, the *ExtendedBaseEntity* now has a medium-high lack of cohesion. The trade-off is that classes which inherit from it will have lower lack of cohesion, and only the *ExtendedBaseEntity* will be slightly more difficult to maintain.

One other improvement is that now each class has a low size.

- Before

ID	CLASS	COUPLING	COMPLEXITY	LACK OF COHESION	SIZE	LOC	COMPLEXITY	COUPLING	LACK OF COHESION	SIZE
3	Board					90	low-medium	low	medium-high	low-medium

- After

ID	CLASS	COUPLING	COMPLEXITY	LACK OF COHESION	SIZE	LOC	COMPLEXITY	COUPLING	LACK OF COHESION	SIZE
3	Board					48	low-medium	low	low-medium	low
4	ExtendedBaseEntity					39	low-medium	low	medium-high	low
6	BaseEntity					42	low	low	low	low

Refactoring of Exception classes

In our project we have used custom exception classes to throw exceptions when a user tries to perform an invalid action. In the beginning, we made all of these 8 classes extend the RuntimeException which turned out not to be a good choice as it implicitly increased the complexity of our code.

Java RuntimeException extends the Exception class already and does not provide any features that we could find useful. Therefore, we decided to choose to inherit from the Exception class instead, which successfully lowered the complexity of the exception classes. To conclude, we should avoid using unnecessarily advanced classes when we can always opt for simpler solutions.

- Before:

ID	CLASS	COUPLING	COMPLEXITY	LACK OF COHESION	SIZE	LOC	COMPLEXITY	COUPLING	LACK OF COHESION	SIZE
9	PostNotFoundExcep...				6	medium-high	low	low	low	
10	BoardThreadNotFou...				6	medium-high	low	low	low	
11	ThreadIsLockedExc...				6	medium-high	low	low	low	
12	BoardIsLockedExce...				6	medium-high	low	low	low	
13	BoardNotFoundExce...				6	medium-high	low	low	low	
14	AuthorizationFail...				6	medium-high	low	low	low	
15	PermissionException				6	medium-high	low	low	low	
16	UserNotFoundExcep...				5	medium-high	low	low	low	

- After:

ID	CLASS	COUPLING	COMPLEXITY	LACK OF COHESION	SIZE	LOC	COMPLEXITY	COUPLING	LACK OF COHESION	SIZE
21	PostNotFoundExcep...				6	low-medium	low	low	low	
22	BoardThreadNotFou...				6	low-medium	low	low	low	
23	ThreadIsLockedExc...				6	low-medium	low	low	low	
24	BoardIsLockedExce...				6	low-medium	low	low	low	
25	BoardNotFoundExce...				6	low-medium	low	low	low	
26	AuthorizationFail...				6	low-medium	low	low	low	
27	PermissionException				6	low-medium	low	low	low	
28	UserNotFoundExcep...				5	low-medium	low	low	low	

Refactoring of PostService and ThreadService classes

These two classes are the core part of our system and are crucial to our application. That is why we decided that they should be refactored, even though their metrics were not as bad as in some other classes.

Two of these functions show similar behaviour, therefore a logical improvement was making them inherit from the class containing elements they both share. The newly created *ContentService* class contains *RestTemplate* and *ThreadRepository* members and methods responsible for user authentication and availability checks. These allowed us to reduce the size of the two bloated classes and remove code duplication.

This operation refactoring two methods which will be mentioned below, lessened the coupling of the PostService method and reduced the number of lines of code in these two classes.

- Before:

ID	CLASS	COUPLING	COMPLEXITY	LACK OF COHESION	SIZE	LOC	COMPLEXITY	COUPLING	LACK OF COHESION	SIZE
1	ThreadService				88	low-medium	medium-high	low-medium	low-medium	
2	PostService				67	low	medium-high	medium-high	low-medium	

- After:

ID	CLASS	COUPLING	COMPLEXITY	LACK OF COHESION	SIZE	LOC	COMPLEXITY	COUPLING	LACK OF COHESION	SIZE
1	ThreadService				64	low-medium	medium-high	low-medium	low-medium	
2	PostService				59	low-medium	low-medium	medium-high	low-medium	
8	ContentService				22	low	low-medium	low	low	

2.2 Method Refactoring

Refactoring of PostService.createPost and PostService.updatePost methods

While refactoring the PostService class these two methods that seemed especially complex and easily reducible in size. Their functionality was also quite ambiguous since they not only created and edited new posts but also checked if the input was correct and handled the threads bound with the posts. Methods should have one specified task and this certainly was not the case there.

The most obvious way to fix them was separating the thread and post parts. Two newly created methods (retrieveThread and updateThread) manage the communication with the thread repository and threads in general which is necessary since posts cannot function without threads.

As you can see in the pictures, doing so greatly reduced their complexity.

- Before (essential cyclomatic complexity, design complexity, cyclomatic complexity):

method	ev(G)	iv(G)	v(G)
nl.tudelft.sem.group20.contentserver.services.PostService.createPost(String,CreatePostRequest)	4	2	5
nl.tudelft.sem.group20.contentserver.services.PostService.updatePost(String,EditPostRequest)	2	2	3

- After:

method	ev(G)	iv(G)	v(G)
nl.tudelft.sem.group20.contentserver.services.PostService.createPost(String,CreatePostRequest)	1	1	1
nl.tudelft.sem.group20.contentserver.services.PostService.updatePost(String,EditPostRequest)	2	1	2
nl.tudelft.sem.group20.contentserver.services.PostService.retrieveThread(long)	2	1	2
nl.tudelft.sem.group20.contentserver.services.PostService.updatePost(String,EditPostRequest)	2	1	2

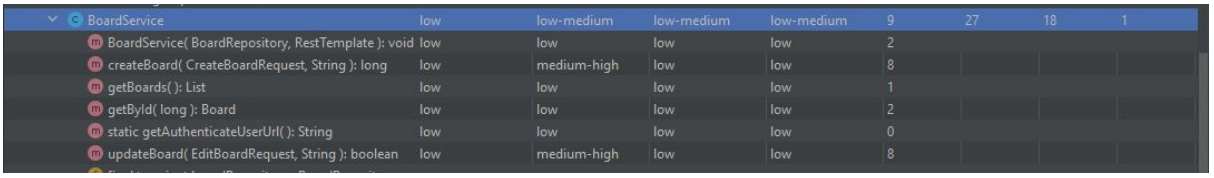
Refactoring of BoardService.createBoard and BoardService.updateBoard methods

After we got the first report from CodeMR, it became clear that there are some problems in the BoardService class, namely two methods, createBoard and updateBoard had medium-high coupling.

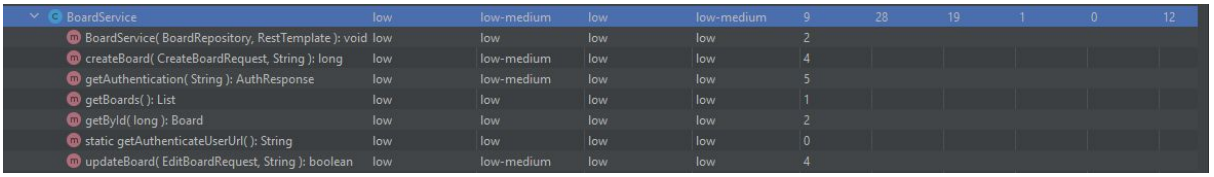
In order to resolve that, it was decided to define distinct functionalities the methods have and split them into smaller ones. However, after a closer look into the methods, it became clear that both methods share similar functionality, namely, they request a

response from the Authentication Server. Hence we moved this logic into its own separate method (getAuthentication).
As a result, createBoard and updateBoard have now low-medium coupling.

- Before:



- After:



Refactoring of UserService.login method

To reduce the essential cyclomatic complexity of this method we decided to split this function into two separate methods. Before, this method was both authenticating and generating a token for a user. Generating a token for the user exercises a different segment of the code and hence it should deserve its own method. This in turn increases the modularity of methods in this class.

To do this we simply proceeded by splitting this method and extracting the token generation logic to form a new separate method called *generateToken*. After this refactoring we notice that our essential complexity has decreased by a fair margin.

- Before:

Method	ev(G)	iv(G)	v(G)
nl.tudelft.sem.group20.authenticationserver.services.UserService.login(String,String)	5.0	5.0	5.0

- After:

Method	ev(G)	iv(G)	v(G)
nl.tudelft.sem.group20.authenticationserver.services.UserService.login(String,String)	3.0	3.0	3.0