

DELFT UNIVERSITY OF TECHNOLOGY
GROUP 51

Assignment 2

15.01.2021

1) Introduction

In order to compute code metrics in our project, we have chosen to make use of the functionalities offered by CodeMR. Thanks to the HTML reports generated by this tool, as well as of all the useful graphs similarly generated, we were able to closely analyse our project and make an analysis of the methods and classes which need improvement.

One of the most important findings we have come across is the fact that our project is already in a good state, as for most of the metrics we chose to compute, indicated that they were “low” or “low-medium”, and we didn’t have any “high” or “very-high” results.

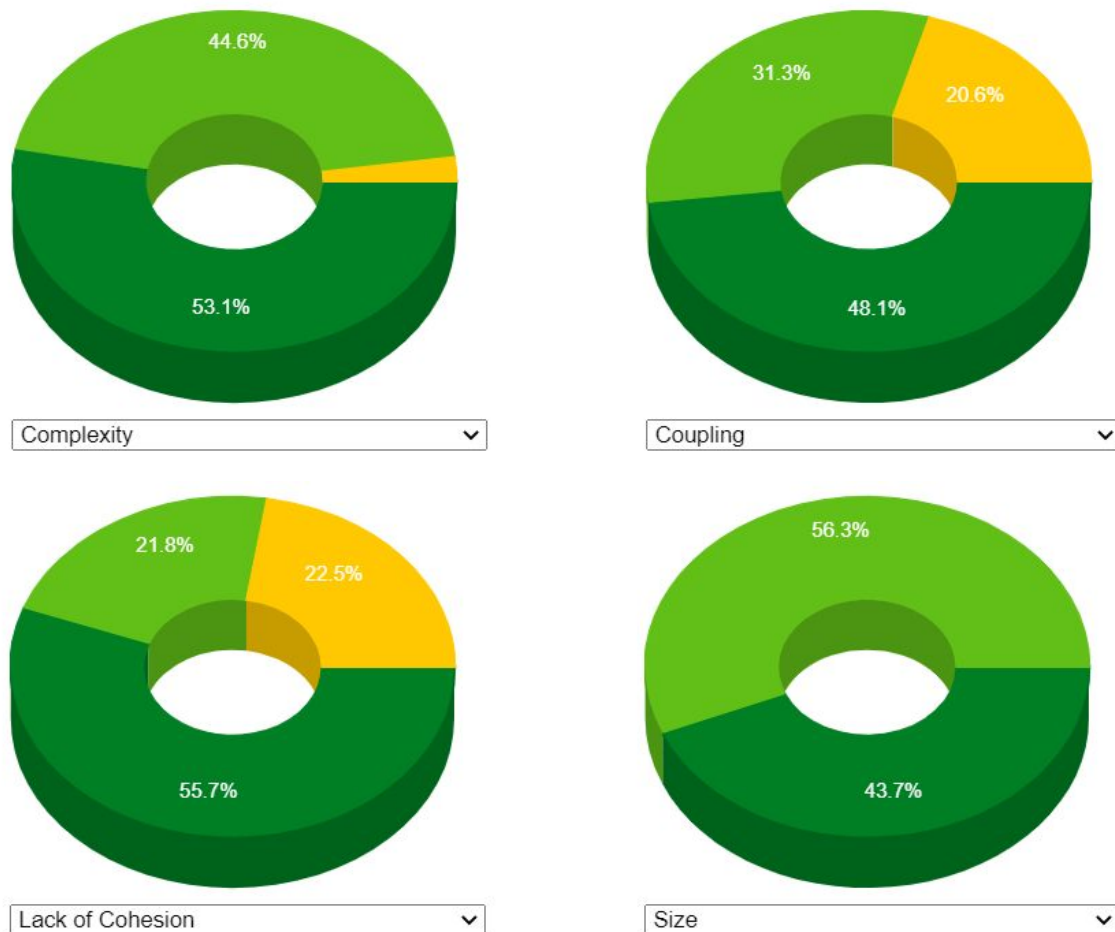


Figure 1: Results of measuring the performance of our project using Complexity, Coupling, Lack of Cohesion and Size as performance metrics.



However, as we too know well by now, there are always things to improve. Of course, we could have improved the “low-medium” results to point towards a better result, but we have chosen to make it of primary importance obtaining better results for the fields which returned a “medium-high” (colored yellow) result when measured with certain other performance metrics.

In order to improve our implementation, we have decided on making the most to improve the result of the C3 metric, which is the maximum value of the related software metrics: Cohesion, Complexity and Coupling. As it can be seen, regarding the overall project, we will need to work on a total of 11 classes, if we wish to achieve good results (“low” or “low-medium”) for the three main computed metrics.

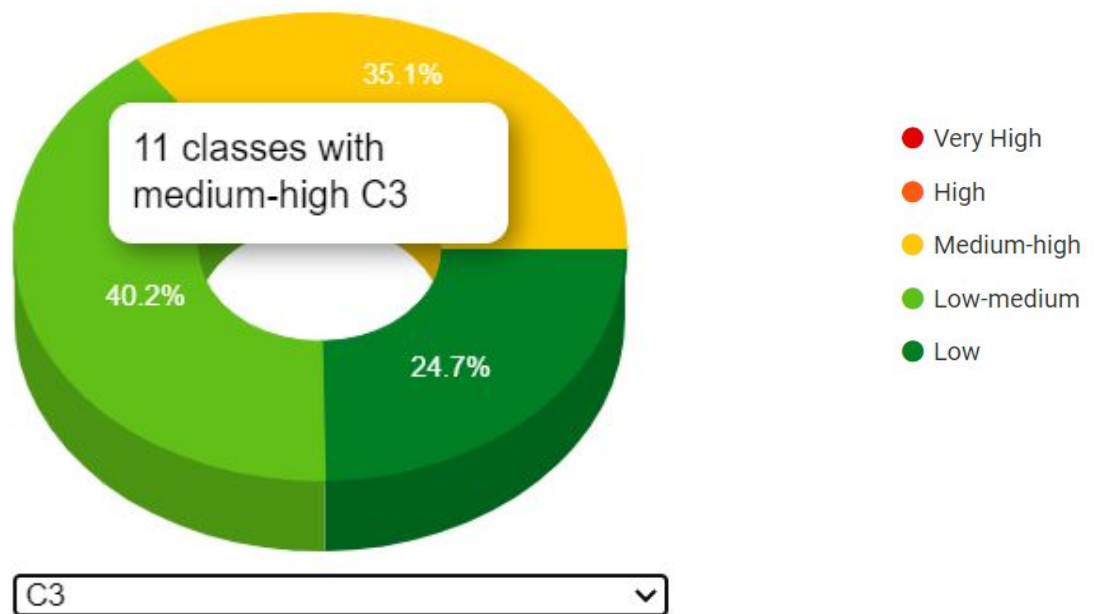


Figure 2: Result of measuring the performance of our project using C3 as a performance metric.

We have decided to go over all of the four main metrics, previously presented in Figure 1, mainly: Complexity, Coupling, Lack of Cohesion and Size, and only take into consideration the “medium-high” results that will need further improvement in our project.

We will now take a more in depth look at each performance metric and discuss the classes and methods to be worked on, in order to achieve our desired results.

a) Lack of Cohesion

Firstly, we have decided to look into the problem of improving Cohesion, or as we might further refer to it and how it is referred to in the CodeMR tool, the Lack of Cohesion. Cohesion measures how well methods within a class are related to each other. Low lack of cohesion is desired because this trait also implies robustness, reliability, reusability, and understandability. In Figure 3, it can be observed that 6 of the total number of classes have a “medium-high” lack of cohesion.

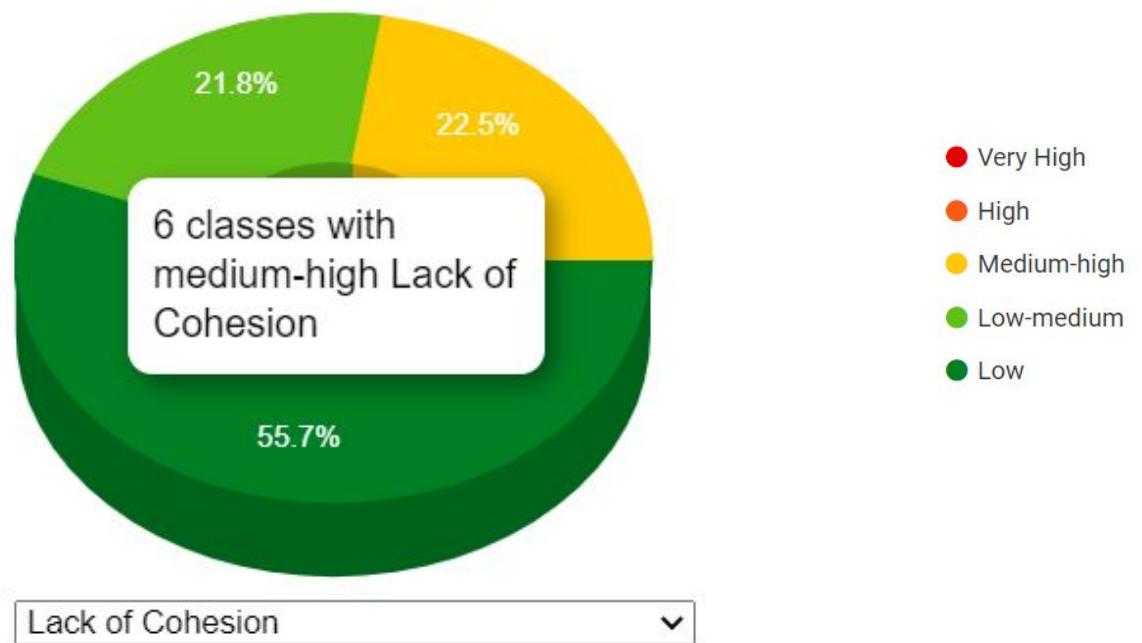


Figure 3: Result of measuring the performance of our project using Lack of Cohesion as a performance metric.

Another important metric associated with Cohesion is Lack of Cohesion among methods, LCAM, which measures the lack of cohesion based on parameter types of methods. Figure 4 shows the thresholds used in LCAM to give a verdict in terms of the performance when measuring Lack of cohesion.

✓	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)

Figure 4: The relation between Lack of Cohesion and LCAM

The 6 classes which have a “medium-high” lack of cohesion can be divided into two sets. The first set is composed of four classes representing entities, in which the “medium-high” Lack of Cohesion is given by the great number of setters and getters that are unused and do not interact with each other, resulting into values close higher than 0.7 for the LCAM metric, as shown in Figure 5. There might be an easy solution to improve this, whereas these classes contain a large amount of redundant methods that are not currently being used. Therefore, we have decided to remove those methods from our implementation in order to improve cohesion.

Name	Lack of Cohesion	LCAM
Product	medium-high	0.764
Product(): void	low	
Product(String, float, int, String): void	low	
addTransaction(Transactions): void	low	
getExpired(): int	low	
getPortionsLeft(): int	low	
getPrice(): float	low	
getProductId(): long	low	
getProductName(): String	low	
getTotalPortions(): int	low	
getTransactionsList(): List	low	
getUsername(): String	low	
removeTransaction(Transactions): void	low	
setExpired(int): void	low	
setPortionsLeft(int): void	low	
setPrice(float): void	low	
setProductId(long): void	low	
setProductName(String): void	low	
setTotalPortions(int): void	low	
setTransactionsList(List): void	low	
setUsername(String): void	low	
expired : int		
portionsLeft : int		
price : float		
productId : long		
productName : String		
totalPortions : int		
transactionsList : List		
username : String		

Name	Lack of Cohesion	LCAM
Request	medium-high	0.708
Request(): void	low	
Request(RequestId, House, User, boolean): void	low	
equals(Object): boolean	low	
getHouse(): House	low	
getId(): RequestId	low	
getUser(): User	low	
hashCode(): int	low	
isApproved(): boolean	low	
setApproved(boolean): void	low	
setHouse(House): void	low	
setId(RequestId): void	low	
setUser(User): void	low	
approved : boolean		
house : House		
id : RequestId		
static final serialVersionUID : long		
user : User		

Name	Lack of Cohesion	LCAM
Transactions	medium-high	0.709
Transactions(): void	low	
getPortionsConsumed(): int	low	
getProductFk(): Product	low	
getProductId(): long	low	
getTransactionId(): long	low	
getUsername(): String	low	
setPortionsConsumed(int): void	low	
setProduct(Product): void	low	
setProductFk(Product): void	low	
setTransactionId(long): void	low	
setUsername(String): void	low	
portionsConsumed: int		
productFk: Product		
transactionId: long		
username: String		

Name	Lack of Cohesion	LCAM
User	medium-high	0.711
User(): void	low	
User(String): void	low	
User(String, House, float, String, Set): void	low	
equals(Object): boolean	low	
getEmail(): String	low	
getHouse(): House	low	
getRequests(): Set	low	
getTotalCredits(): float	low	
getUsername(): String	low	
hashCode(): int	low	
setEmail(String): void	low	
setHouse(House): void	low	
setRequests(Set): void	low	
setTotalCredits(float): void	low	
setUsername(String): void	low	
email: String		
house: House		
requests: Set		
static final serialVersionUID: long		
totalCredits: float		
username: String		

Figure 5: Results of calculating Lack of Cohesion and LCAM on our project.

Although we have decided not to modify the latter four classes, there are still two classes for which the Lack of Cohesion is “medium-high”.

One of them is the Authentication class from the Authentication microservice. It’s LCAM score exceeds 0.7 and therefore it is classified as “medium-high”, as can be seen in Figure 6.

Name	Lack of Cohesion	LCAM
Authentication	medium-high	0.758
configure(AuthenticationManagerBuilder): void	low	
configure(HttpSecurity): void	low	
getDataSource(): DataSource	low	
getJwtConf(): JwtConf	low	
getUserDetailsService(): UserDetailsService	low	
jdbcUserDetailsManager(): JdbcUserDetail	low	
jwtConfig(): JwtConf	low	
passwordEncoder(): PasswordEncoder	low	
setDataSource(DataSource): void	low	
setJwtConf(JwtConf): void	low	
setUserDetailsService(UserDetailsService)	low	
dataSource: DataSource		
jwtConf: JwtConf		
userDetailsService: UserDetailsService		

Figure 6: Results after measuring the performance of Authentication class.

We will try to refactor this class in order to improve the performance of it with regards to the Lack of Cohesion metric, by improving the LCAM and bringing it to a value smaller than 0.7.

The second class which has a similar problem is the ProductController class of the Transactions microservice. Figure 7 shows that the LCAM value of this class is 0.702. This class only exceeds the threshold by 0.02, therefore improving the distribution of a few methods could easily fix our problem.

Name	Lack of Cohesion	LCAM
ProductController	medium-high	0.702
addNewProduct(Product): ResponseEntit	low	
deleteExpired(long): ResponseEntity	low	
deleteProduct(String, long): ResponseEnti	low	
getAllProducts(String): List	low	
getJwtConf(): JwtConf	low	
getProductRepository(): ProductRepositor	low	
getProductsByHouse(int): List	low	
getUserProducts(String): ResponseEntity	low	
setExpired(String, long): ResponseEntity	low	
setJwtConf(JwtConf): void	low	
setProductRepository(ProductRepository)	low	
updateProduct(String, Product): Response	low	
jwtConf : JwtConf		
productRepository : ProductRepository		
sortProductsStrategy : SortProductsStrateg		

Figure 7: LCAM value for ProductController class

b) Size

As far as the size is concerned, which is measured by the number of lines or methods in the code, it proved to be in a really good state as all the results after applying this metric reported by using the before mentioned tool, indicated our project to be “low” or “low-medium” as it can be seen in Figure 8, using dark green to indicate “low” and light green to indicate “low-medium”.

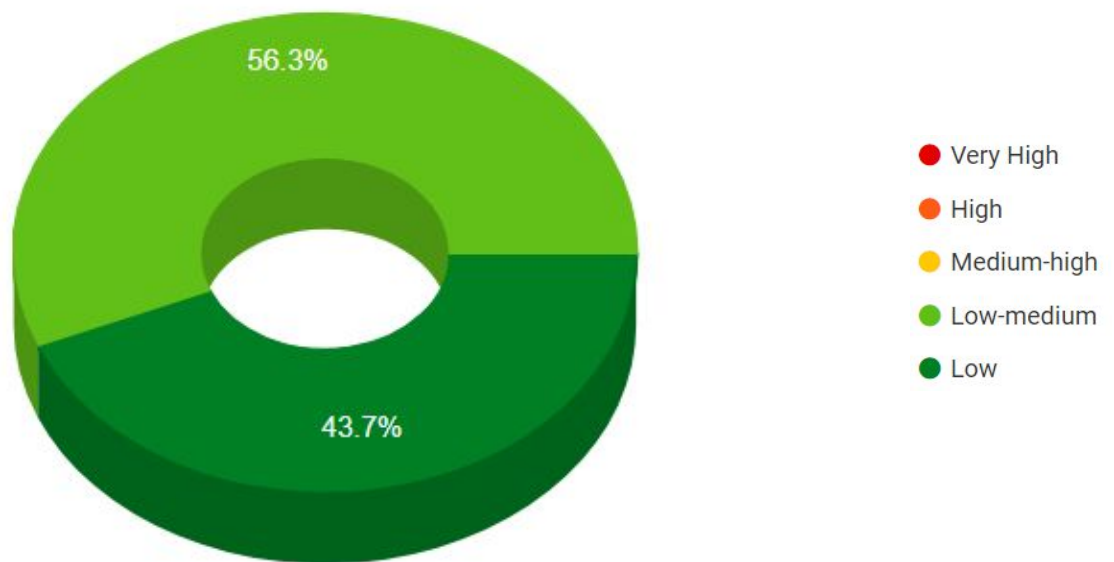


Figure 8: Result of measuring the performance of our project using Size as a performance metric.

Given these results, we have decided to focus primarily on obtaining better results when measuring with the metrics that give a worse result, therefore we will not use any refactoring methods to try and change these results.

c) Complexity

Regarding the complexity, which implies how difficult it is to understand and describes the interactions between a number of entities, we had very good results.

The complexity thresholds are calculated based on three scores: WMC (Weighted Method Count - the weighted sum of all classes' methods), RFC (Response For a Class - the number of the methods that can be potentially invoked in response to a public message received by an object of a particular class), and DIT (Depth of Inheritance Tree - the position of the class in the inheritance tree).

Figure 9 represents the checks made on these scores and then the category of the complexity is stated.

✓	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)

Figure 9: Thresholds for the Complexity results, using the WMC, RFC, and DIT scores.

As it can be seen from Figure 10, only one class had a complexity of “medium-high”, and more than half of the classes have “low” complexity.

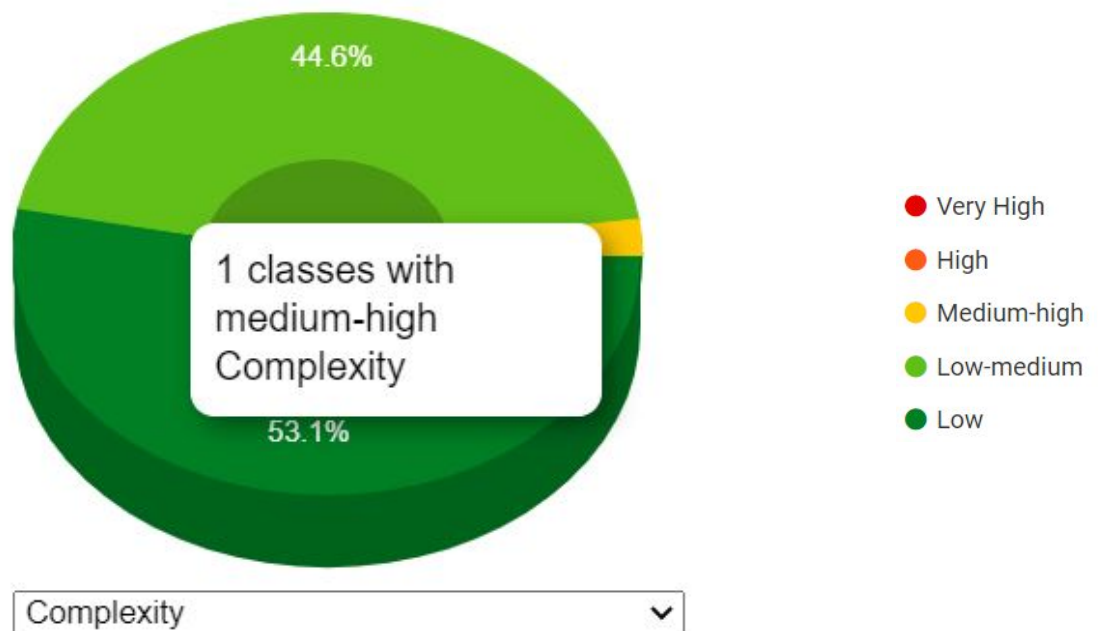


Figure 10: Result of measuring the project performance using Complexity as performance metric.

The only class that has a “medium-high” complexity is the “JwtFilter” class, in the “config” package of the Authentication microservice.

Name	Complexity	Coupling	Size	Lack of Cohesion	CBO
▼ JwtFilter	medium-high	low-medium	low	low-medium	10
JwtFilter(AuthenticationManager, JwtConf	low	low	low	low	3
attemptAuthentication(HttpServletRequest	low	low-medium	low	low	5
getAuthManager(): AuthenticationManag	low	low	low	low	1
getJwtConfig(): JwtConf	low	low	low	low	1
successfulAuthentication(HttpServletRequest	low	low-medium	low	low	5
final authManager : AuthenticationManag					
final jwtConfig : JwtConf					

Now, we will take a look at the scores used in calculating the complexity, then compare them with our resulting scores from Figure 11 and deduce what can be improved.


Name	Complexity	WMC	RFC	DIT
▼  JwtFilter	medium-high	6	33	4

Figure 11: Results of WMC, RFC, DIT scores used in calculating the Complexity category.

From Figure 10 and 11, we can say that our WMC and RFC are not high at all, which would fit into the “low” complexity category. However, we have a DIT score of 4, and this is where it went wrong. In order to get rid of the performance complexity of “medium-high”, we will need a DIT score of maximum 3.

We need to have in mind that this score represents the position of the class in the inheritance tree. It is zero for root and any non-inherited classes. For the case of multiple inheritance, the metric shows the maximum length.

Unfortunately, we can not achieve a DIT score of less than 3. All the inheritance is needed in order for the JwtFilter class to properly work, and we can not skip any level. Due to this reason, we will not be able to improve the Complexity score of this class, but we were able to fix the Coupling and Lack of Cohesion problems in this class and its methods.

d) Coupling

Lastly, the coupling of the classes in our project, representing the attribute or method references, calls, or inheritance between any two classes, had fairly good results. However, they are quite a number of classes that could be improved, regarding how tightly coupled they are.

When we will discuss more in depth about the classes and methods involved, we will refer to the CBO (Coupling between Object Classes) score, as well, which represents the number of classes that a class is coupled to. It is calculated by counting other classes whose attributes or methods are used by a class, plus those

that use the attributes or methods of the given class. More coupling means that the code becomes more difficult to maintain, because changes in other classes can also cause changes in that class.

Figure 12 represents the checks made on these scores and then the category of the coupling is stated. These thresholds are referred to the classes.

Enable	Level	Quality Attribute	Condition
<input checked="" type="checkbox"/>	low	Coupling	(CBO <= 5)
<input checked="" type="checkbox"/>	low-medium	Coupling	(CBO > 5 AND CBO <= 10)
<input checked="" type="checkbox"/>	medium-high	Coupling	(CBO > 10 AND CBO <= 20)
<input checked="" type="checkbox"/>	high	Coupling	(CBO > 20 AND CBO <= 30)
<input checked="" type="checkbox"/>	very-high	Coupling	(CBO > 30)

Figure 12: Thresholds for the Coupling results, using the CBO score.

In the case of methods, however, we have found different thresholds. Based on the results we have analyzed, methods with CBO lower than 3 (inclusive) will refer to the “low” coupling category, anything between 4-6 will be for “low-medium”, and anything over 7 (inclusive), will go into categories of “medium-high” and above.

For these reasons, what we wish to say is basically that any CBO score below 10 for classes or 7 for methods is a desirable number for the score. Anything higher than that, could result in a “medium-high” or worse performance.

As it can be seen from Figure 13, there are in total 6 classes that could be further improved, in order to make the classes more loosely coupled between each other.

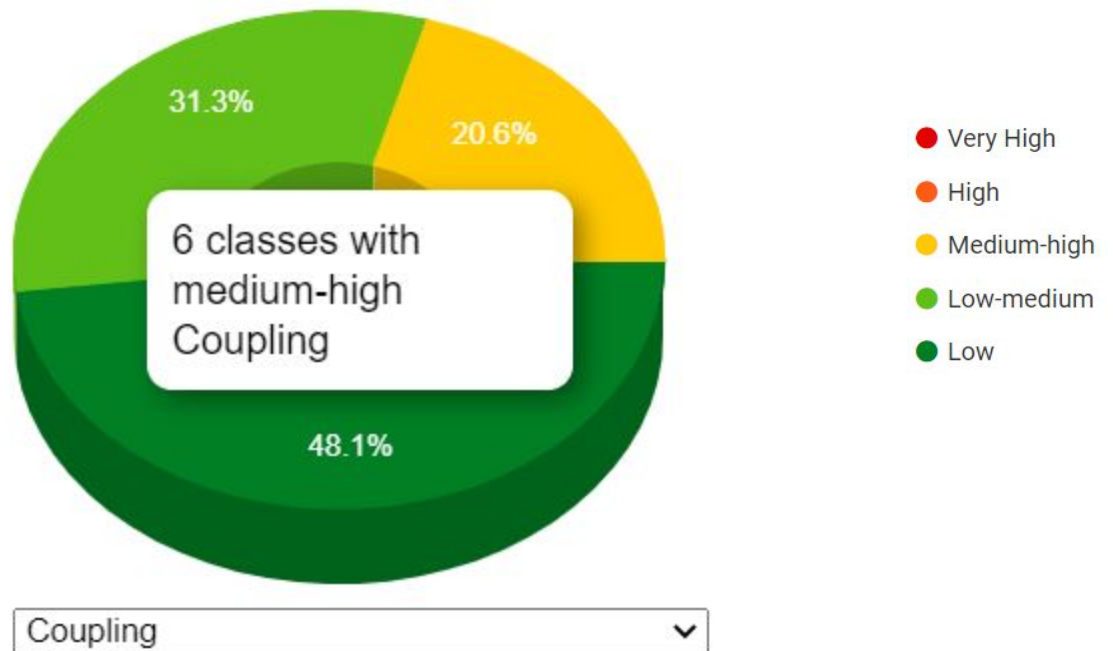


Figure 13: Result of measuring the project performance using Coupling as a performance metric.

Regarding the 6 classes that need work on, they are the following:

- “Authentication” class, “config” package, Authentication microservice

Name	Complexity	Coupling	Size	Lack of Cohesion	CBO
Authentication	low-medium	medium-high	low-medium	medium-high	16
configure(AuthenticationManagerBuilder	low	low	low	low	2
configure(HttpSecurity): void	low	medium-high	low	low	10
getDataSource(): DataSource	low	low	low	low	0
getJwtConf(): JwtConf	low	low	low	low	1
getUserDetailsService(): UserDetailsServic	low	low	low	low	1
jdbcUserDetailsManager(): JdbcUserDetai	low	low	low	low	1
jwtConfig(): JwtConf	low	low	low	low	1
passwordEncoder(): PasswordEncoder	low	low	low	low	2
setDataSource(DataSource): void	low	low	low	low	0
setJwtConf(JwtConf): void	low	low	low	low	1
setUserDetailsService(UserDetailsService ;	low	low	low	low	1
dataSource : DataSource					
jwtConf : JwtConf					
userDetailsService : UserDetailsService					

Looking at the “Coupling” column, we see that in this class (with a CBO score of 16), all methods but one, more specifically the *configure(HttpSecurity)* method (with a CBO score of 10), have low coupling. From this, it could be said that if we

would refactor the *configure* method, then we could bring down the performance to a “low-medium” or “low”, in the best case.

Unfortunately, we will not be able to improve the coupling of this method or class, as every internal method call that was used in creating the *configure* method is actually needed and they are crucial for our configuration of Spring Security to properly work. Moreover, every outside call of this class’s methods is needed in order for our Authentication microservice to do its job. Due to this reason, we will not be able to make any improvement to this method and class. Even though the chain of method calls in the *configure* method of the Authentication class can be split into several methods and classes, we believe that this will hurt the readability of the code, and will not help in any way. The Coupling metric in this case is not a valuable score, because the main problem here is the configuration method, and it would not make sense to split such a method across multiple classes.

- “UserController” class, “controllers” package, Authentication microservice

Name	Complexity	Coupling	Size	Lack of Cohesion	CBO
UserController	low	medium-high	low-medium	low	15
register(UserRegister, UriComponentsBuil	low	very-high	low-medium	low	15
static nop(String): void	low	low	low	low	0
transient discoveryClient : EurekaClient					
transient jdbcUserDetailsManager : JdbcU					
transient passwordEncoder : PasswordEnc					

Looking at the “Coupling” column, we see that in this class, the *register* method has “very-high” coupling, with a CBO score of 15, with the *UserController* class having the same score.

- “AuthenticationConfigurer” class, Gateway microservice

Name	Complexity	Coupling	Size	Lack of Cohesion	CBO
nl.tudelft.sem.gateway	low	low	low	low	
AuthenticationConfigurer	low-medium	medium-high	low	low	10
configure(HttpSecurity): void	low	medium-high	low	low	10
getJwtConf(): JwtConf	low	low	low	low	1
jwtConfig(): JwtConf	low	low	low	low	1
setJwtConf(JwtConf): void	low	low	low	low	1
jwtConf : JwtConf					

This class and *configure* method are very similar to what we had in the Authentication microservice (reference: page 10), but this time, the configure method is in the Gateway microservice. The method calls are just as important for the same reasons stated beforehand - they are crucial for our configuration of the *Gateway* and for it to properly work.

Unfortunately, due to all these reasons, we will not be able to make any improvement regarding the coupling to this method and class.

- “RequestController” class, “controllers” package, Requests microservice

Name	Complexity	Coupling	Size	Lack of Cohesion	CBO
RequestController	low	medium-high	low-medium	low	12
addRequest(Request): ResponseEntity	low	low-medium	low	low	4
deleteRequest(RequestId): ResponseEntit	low	low-medium	low	low	4
getAllRequests(): List	low	low	low	low	1
getRequestByld(RequestId): ResponseEnt	low	low	low	low	3
membersAcceptingRequest(String, int, Str	low	medium-high	low	low	9
updateRequest(Request): ResponseEntity	low	low-medium	low	low	4
transient houseRepository : HouseRepositc					
transient requestRepository : RequestRepc					
userRepository : UserRepository					

Looking at the “Coupling” column, we see that this class (CBO score of 12), and the *membersAcceptingRequest* method have “medium-high” coupling (with a CBO score of 9). From this, it could be said that if we would refactor the *membersAcceptingRequest* method, then we could bring down the performance to a “low-medium” or “low”, in the best case.

We could reduce the coupling of the *membersAcceptingRequest* method and the *RequestController* class by creating a separate service class. For this particular

method, all three repositories available were used (user, house and request repository), and that might be one of the reasons why the coupling between classes was a bit high. Furthermore, the coupling of the class will also be reduced as the attributes that were used or the method calls will then be counted for the CBO score of the separate class.

- “House Controller” class, “controllers” package, Requests Microservice
- “ProductController” class, “controllers” package, Transactions microservice

Name	Complexity	Coupling	Size	Lack of Cohesion	CBO
ProductController	low-medium	medium-high	low-medium	medium-high	15
addNewProduct(Product): ResponseEnti	low	low-medium	low	low	6
deleteExpired(long): ResponseEntity	low	low-medium	low	low	5
deleteProduct(String, long): ResponseEnt	low	low-medium	low	low	6
getAllProducts(String): List	low	medium-high	low	low	7
getJwtConf(): JwtConf	low	low	low	low	1
getRepository(): ProductRepositor	low	low	low	low	1
getProductsByHouse(int): List	low	low	low	low	2
getUserProducts(String): ResponseEntity	low	low-medium	low	low	4
setExpired(String, long): ResponseEntity	low	low-medium	low	low	5
setJwtConf(JwtConf): void	low	low	low	low	1
setRepository(ProductRepository): low	low	low	low	low	1
updateProduct(String, Product): Respons	low	low-medium	low	low	4
jwtConf : JwtConf					
productRepository : ProductRepository					
sortProductsStrategy : SortProductsStrateg					

Looking at the “Coupling” column, we see that this class (CBO score of 15), and the *getAllProducts* method have “medium-high” coupling (with a CBO score of 7). From this, it could be said that if we would refactor the *getAllProducts* method, then we could bring down the performance to a “low-medium” or “low”, in the best case.

We can reduce the coupling of the *getAllProducts* method by extracting method for the “strategy of returning products” to new method

- “TransactionValidator” class, “handlers” package, Transactions microservice

Name	Complexity	Coupling	Size	Lack of Cohesion	CBO
> TokensValidator	low-medium	low-medium	low	low	6
▼ TransactionValidator	low-medium	medium-high	low	low	11
TransactionValidator(EurekaClient): void	low	low	low	low	1
checkNext(Transactions, ProductRepositor	low	medium-high	low	low	7
handle(Transactions, ProductRepository, T	low	medium-high	low	low	9
transient discoveryClient : EurekaClient					
transient username : String					

Looking at the “Coupling” column, we see that this class (CBO score of 11), the *checkNext*, and *handle* methods have “medium-high” coupling (with a CBO score of 7, and 9 respectively). From this, it could be said that if we would refactor the two methods, then we could bring down the performance to a “low-medium” or “low”, in the best case.

We can reduce the coupling of the *checkNext* and *handle* methods by splitting them and introducing new “helper” classes that will implement some of the functionalities.

Conclusion

Based on the observations done until now, we have chosen 6 classes and 6 methods to work on, so we could improve the quality of our project.

2) Code refactoring

The RequestController class and its membersAcceptingRequest() method (1 class and 1 method refactored):

As already mentioned, the membersAcceptingRequest() method was the main cause of the medium-high coupling in the RequestController class. To improve the coupling of that class to low-medium, we had to reduce the CBO metric, which was 12, to 10 or less.

We noticed that this method was using 3 repositories, namely:

RequestRepository, HouseRepository and UserRepository. Throughout the class, the last two were only used in this exact method. We refactored the code by introducing an AcceptUserHelper class containing an acceptUser() method that accepts another user as long as the conditions are met to which we extracted this functionality from membersAcceptingRequest(). This resulted in the CBO of the class decreasing from 12 to 9. At the same time CBO of the membersAcceptingMethod() was reduced from 9 to 6, thus making the coupling decrease from medium-high to low-medium as well. Unfortunately, as a trade-off, the LCOM metric raised from 0.67 to 0.673 but the Lack of Cohesion stayed low as the threshold is 0.7.

These are the new metrics:

Name	Complexity	Coupling	Size	Lack of Cohesion	CBO
RequestController	low	low-medium	low-medium	low	9
addRequest(Request): ResponseEntity	low	low-medium	low	low	4
deleteRequest(RequestId): ResponseEntity	low	low-medium	low	low	4
getAllRequests(): List	low	low	low	low	1
getRequestById(RequestId): ResponseEnti	low	low	low	low	3
membersAcceptingRequest(String, int, Stri	low	low-medium	low	low	6
updateRequest(Request): ResponseEntity	low	low-medium	low	low	4
transient houseRepository : HouseReposito					
transient requestRepository : RequestRepos					
userRepository : UserRepository					
AcceptUserHelper	low	low-medium	low	low	7
AcceptUserHelper(UserRepository, HouseF	low	low	low	low	2
acceptUser(String, int, String): ResponseEr	low	low-medium	low	low	6

Extract method from deleteHouse() method (1 method refactored):

We have noticed that the coupling of the deleteHouse() method could be improved as it was medium-high with CBO being 7.

Name	Complexity	Coupling	Size	Lack of Cohesion	CBO
HouseController	low-medium	low-medium	low-medium	low-medium	9
HouseController(HouseRepository, UserRe low		low	low	low	2
addNewHouse(House, String): ResponseE low		low	low	low	3
deleteHouse(int): ResponseEntity	low	medium-high	low	low	7

We extracted a part of this method that goes over each user and sets his house to null to a new method called nullifyHousesForUsers().

Name	Complexity	Coupling	Size	Lack of Cohesion	CBO
HouseController	low-medium	low-medium	low-medium	low-medium	9
HouseController(HouseRepository, UserRe low		low	low	low	2
addNewHouse(House, String): ResponseE low		low	low	low	3
deleteHouse(int): ResponseEntity	low	low-medium	low	low	4
getAllHouses(): List	low	low	low	low	1
getAllUsersFromHouse(int): ResponseEnti low		low	low	low	2
getHouseByHouseNumber(int): Response low		low	low	low	2
getHouseByUsername(String): ResponseEi low		low-medium	low	low	4
getUserNamesByHouse(int): ResponseEnti low		low-medium	low	low	5
nullifyHousesForUsers(int): void	low	low	low	low	3

This way, the CBO decreased to 4 and subsequently coupling has become low-medium. Also the complexity was slightly improved by decreasing the number of lines of code from 13 to 10 and the McCabe Cyclomatic Complexity from 3 to 2.

The UserController class and its register() method (1 class and 1 method refactored):

There were many problems with the UserController's register() method. CodeMR initially had given the class's score as "medium-high" for coupling and "low-medium" for its size.

Its register method scored even worse - "very-high" for coupling and "low-medium" again for size. The method was also really long and complex as it did several things, but for some reason CodeMR did not give it a bad

score for its complexity. Nevertheless, it was really important that we fix this class and method.

We began the refactoring by splitting the register method into two methods - `jdbcCreateUser()` and `register()`. We also extracted the communication with the requests microservice into a separate class - `MicroserviceCommunicator`, following the pattern we had used with the other communicator classes. Even though the method was a lot more readable now, the CodeMr scores did not change much, so we needed to keep refactoring. After all there was still a lot of functionality that could be split further.

The biggest problem was the Coupling of the methods and classes, because CodeMR required a CBO (Coupling Between Object Classes) score of ≤ 5 . Since the Coupling scores of both classes (`MicroserviceCommunicator` and `UserController`) were still “medium-high”, we decided to introduce two new Helper classes that extract some of the functionality like building and sending http requests, and serializing `UserRequest` objects to JSON, so that they can be sent to the requests microservice. This reduced the `MicroserviceCommunicator` class’s Coupling score, however, the `UserController` and the helper classes needed more work.

In the end, we split the register method into 6 other classes and a total of 15 other smaller methods. Each of them had the desired “low” score on each of the 4 code metrics we decided to improve - Complexity, Coupling, Size, Lack of Cohesion.

Because of this refactoring, the readability of the method really improved. Since most of the newly created methods have clear names and javadocs, the functionality of the register method is more clear.

Another benefit of the refactoring is that most of the code can be reused now. For example, now there are methods for serializing `User` objects, sending HTTP requests, checking HTTP response statuses and more that can be reused in the future if our codebase grows.

It is also easier to modify the code, without breaking the whole register method. We will be able to introduce new functionality, or change our

existing functionality without the need to change too many methods, because the Coupling of all newly introduced methods is low.

For more information about all the new classes we had to add, you can see the merge request for the authentication refactoring - [LINK HERE](#). This was a really complicated method and class to fix, so we will not document every necessary small change.

These are the new and improved metrics of the UserController class and register method.

Name	Complexity	Coupling	Size	Lack of Cohesion	CBO
UserController	low	low	low	low	4
register(UserRegister): ResponseEntity	low	low	low	low	3
static nop(String): void	low	low	low	low	0
transient discoveryClient : EurekaClient					
transient jdbcHelper : JdbcHelper					

And all classes that were created in order to split the functionality:

Name	Complexity	Coupling	Size	Lack of Cohesion	CBO
JdbcHelper	low	low	low	low	5
checkUserExists(String): Optional	low	low	low	low	3
jdbcCreateUser(UserRegister): void	low	low	low	low	3
transient jdbcUserDetailsManager : Jc					
transient userBuilderHelper : UserBuil					
UserBuilderHelper	low	low	low	low	4
buildUser(String, String): UserDetail	low	low	low	low	3
encodePassword(String): String	low	low	low	low	1
transient passwordEncoder : Passwor					

▼	HttpRequestHelper	low	low	low	low	5
	static buildHttpRequest(EurekaClient)	low	low	low	low	2
	static checkResponseCode(HttpResp	low	low	low	low	3
	static sendHttpRequest(EurekaClient	low	low	low	low	1
▼	UserHelper	low	low	low	low	5
	UserHelper(EurekaClient): void	low	low	low	low	2
	addUser(String, String): ResponseEn	low	low	low	low	2
	nop(String): void	low	low	low	low	0
	postNewUser(String, String): Respor	low	low	low	low	3
	serializeUserRequest(String, String): low	low	low	low	low	1
	transient userRequestHelper : UserRei					
▼	UserRequestHelper	low	low	low	low	5
	addUser(String, String): ResponseEr	low	low	low	low	2
	serializeUserRequest(String, String): low	low	low	low	low	2
	UserRequestHelper(EurekaClient): vc	low	low	low	low	2
	transient mapper : ObjectMapper					
	transient microserviceCommunicator					
▼	MicroserviceCommunicator	low	low	low	low	5
	MicroserviceCommunicator(EurekaCl	low	low	low	low	1
	addNewUser(String, String): Respon	low	low	low	low	3
	checkResponse(HttpResponse, String	low	low	low	low	2
	sendRequest(EurekaClient, String): f	low	low	low	low	2
	static nop(HttpResponse): void	low	low	low	low	0
	transient discoveryClient : EurekaClie					

Authentication class (1 class refactored):

While refactoring the UserController we noticed that the Authentication class has some unnecessary getters and setters for spring autowired variables. Since these variables are autowired, it is not necessary to use the getters and setters, so we decided to remove them.

This reduced the size score of the class from low-medium to low and its Lack of Cohesion score from medium-high to low.

Here are the new and improved metrics for the Authentication class:

Name	Complexity	Coupling	Size	Lack of Cohesion	CBO
▼ Authentication	low-medium	medium-high	low	low	15
configure(AuthenticationManagerBuilder)	low	low	low	low	2
configure(HttpSecurity): void	low	medium-high	low	low	10
jdbcUserDetailsManager(): JdbcUserDetail	low	low	low	low	1
jwtConfig(): JwtConf	low	low	low	low	1
passwordEncoder(): PasswordEncoder	low	low	low	low	2
transient dataSource : DataSource					
transient jwtConf : JwtConf					

As we described above, the complexity and coupling of this class cannot be fixed.

JwtFilter and its attemptAuthentication() and successfulAuthentication() methods (1 class and 2 methods):

The attemptAuthentication() and successfulAuthentication() methods had a Coupling score of “low-medium”, and because of that, the whole class also had a Coupling score of “low-medium”.

CodeMR also gave it a “medium-high” Complexity score, because it has a DIT (Depth of Inheritance Tree) score equal to 4. This however cannot be fixed, because this is a Filter configuration method, and as such it needs to extend UsernamePasswordAuthenticationFilter in order to authenticate users and generate JWT tokens. It is not up to us to fix this, because it is simply required by Spring Security.

In order to fix the Coupling problems of the methods, we created three new authentication helper classes. The first one is responsible for checking whether the username and password combination is correct. The second one is used to generate a JWT token on successful authentication, and the last one is used to simplify the calls to the other two. This makes the functionality of the JwtFilter methods more clear, because each new method is smaller, and is named according to its only functionality, making them more readable. The low coupling also makes the functionality easier to maintain, because methods and classes are responsible for smaller, simpler tasks.

These are the new scores of the JwtFilter class and its methods:

Name	Complexity	Coupling	Size	Lack of Cohesion	CBO
▼  JwtFilter	medium-high	low	low	low	5
 JwtFilter(AuthenticationManager, JwtConf	low	low	low	low	3
 attemptAuthentication(HttpServletRequest	low	low	low	low	3
 successfulAuthentication(HttpServletRequest	low	low	low	low	3
 final transient authManager : Authenticati					
 final transient jwtConfig : JwtConf					

And for the newly implemented helper classes:

Name	Complexity	Coupling	Size	Lack of Cohesion	CBO
AuthenticationAttemptHelper	low	low	low	low	3
static obtainCredentials(InputStream): Us	low	low	low	low	3
AuthenticationHelper	low	low	low	low	4
static generateToken(Authentication, Dat	low	low	low	low	2
static obtainCredentials(InputStream): Us	low	low	low	low	2
AuthenticationSuccessHelper	low	low	low	low	4
static createJwtBuilder(Authentication): J	low	low	low	low	3
static generateJwtToken(Authentication, T	low	low	low	low	3

Extract method from getAllProducts(): (1 method refactored)

This method was long in terms of lines of code (code smell - “blob code”), moreover it used many variables of the class, thus it could be described as “code coupler”. We decided to extract some of the functionality of the method - “creating the strategy for retrieval of products, based on the name of the strategy provided”, to another method.

We did so by creating a new method: createStrategy(String strategy) (named after the intention of the method). Then we copied the code from the source method to the newly created method. Next, we added a call in the source method to the newly created method.

After doing the refactoring the code is more readable (dealing with “blob code”) (functionality splitted into two methods with names that describe their behaviour) and the coupling of the getAllProducts() is reduced from “medium-high” to low, making the code easier to maintain.

The newly created method is a private method, as it is not a part of our API (it is used only internally), thus we can say that we followed one of the core OOP principles - encapsulation.

Here we can observed the decreased coupling of the getAllProducts() method from medium high to low:

Name	Complexity	Coupling	Size	Lack of Cohesion	CBO
template					
nl.tudelft.sem.transactions.controllers	low				
ProductController	low-medium	medium-high	low-medium	low-medium	15
addNewProduct(Product): Res	low	low-medium	low	low	6
createStrategy(String): SortProc	low	low-medium	low	low	6
deleteExpired(long): Response	low	low-medium	low	low	5
deleteProduct(String, long): Re	low	low-medium	low	low	6
getAllProducts(String): List	low	low	low	low	2
getRepository(): Product	low	low	low	low	1
getProductsByHouse(int): List	low	low	low	low	2
getUserProducts(String): Respo	low	low-medium	low	low	4
setExpired(String, long): Respo	low	low-medium	low	low	5
setRepository(ProductR	low	low	low	low	1
updateProduct(String, Product)	low	low-medium	low	low	4
productRepository : ProductRep					
transactionConf : JUnitConf					

Remove unused methods and unnecessary comments (many classes refactored; counted as 1 class refactored)

There was a “dead code” smell in our code. This was due to many methods that we thought beforehand that would be useful, but they turned out to be not useful in our code (not called at all). Moreover, some of the methods turned out to be obsolete due to constant restructuring of our code.

Dead code made the classes harder to understand by new developers (they would have to read the code that is not used at all in our system). This makes the code less maintainable.

By deleting unused methods we improved the LCAM metric for the affected classes.

Splitting of controllers so that each controller is responsible for one and only one corresponding JPA entity (many classes refactored; counted as 1 class refactored)

Note: this code refactoring was performed during the first part of the project

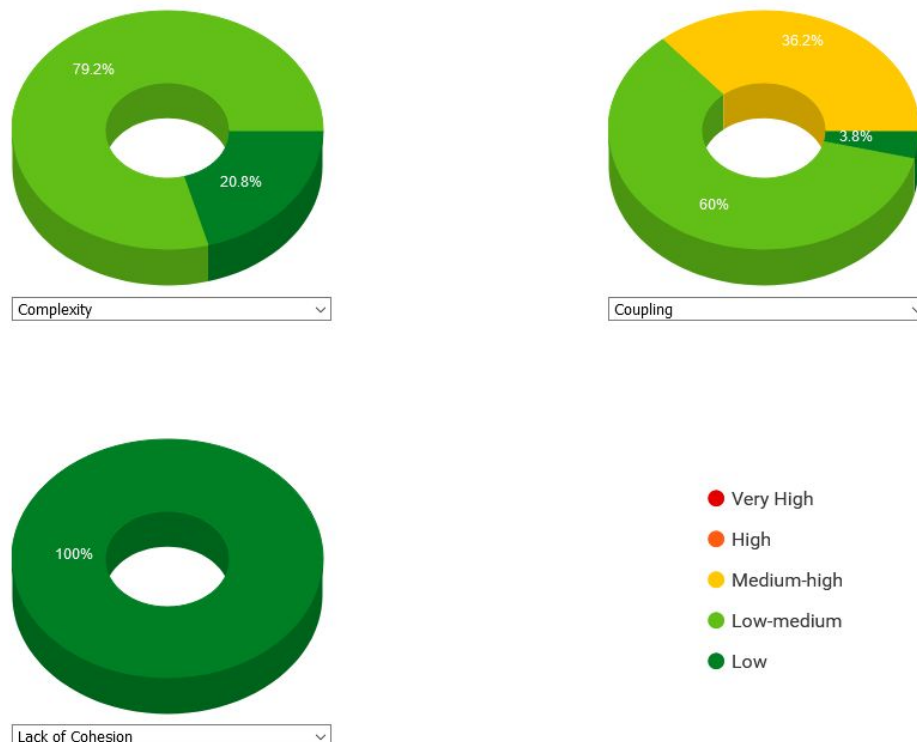
We splitted the functionality of the controllers, so that there is one controller corresponding to one JPA entity. This introduces clear separation of responsibilities between classes. Without this refactoring there would be one “god class” for each microservice which would serve as an API. Which would make the code hard to maintain (for example: merge conflicts, understanding of the whole class needed to change the code, hard unit testing due to mocks that require a lot of methods to be mocked etc.)

This change vastly improved the LOC and lack of cohesion metrics.

Refactoring the handlers package - Transactions microservice:

The package containing the validators for creating transactions, implemented for the first assignment, did not perform very well when we generated the CodeMR statistics. The results we obtained encouraged us to refactor the whole handlers package.

Distribution of Quality Attributes
Complexity, Coupling, Cohesion, and Size



Name	Complexity	Coupling	Size	Lack of Cohesion	CBO	RFC	SRFC	DIT	NOC	WMC
template										
nl.tudelft.sem.transactions.handlers	low	low-medium	low-medium	low						28
> BaseValidator	low	low-medium	low	low	7	14	10	1	4	6
> HouseValidator	low-medium	low-medium	low	low	7	17	13	2	0	6
> ProductValidator	low-medium	low-medium	low	low	7	15	12	2	0	4
> TokensValidator	low-medium	low-medium	low	low	6	17	7	2	0	2
> TransactionValidator	low-medium	medium-high	low	low	11	30	24	2	0	8
> Validator	low	low	low	low	4	2	0	1	1	2

Name	LOC	CMLOC	NOF	NOSF	NOM	NOSM	NORM	LCOM	LCAM	LTCC	ATFD	SI
template												
nl.tudelft.sem.transactions.handlers	130											
> BaseValidator	22	20	1	0	4	0	0	0.0	0.5	0.0	0	0.0
> HouseValidator	29	28	0	0	1	0	0	0.0	0.0	0.0	0	0.0
> ProductValidator	16	15	0	0	1	0	0	0.0	0.0	0.0	0	0.0
> TokensValidator	11	10	0	0	1	0	0	0.0	0.0	0.0	0	0.0
> TransactionValidator	47	44	2	0	3	0	1	0.5	0.333	1.0	0	0.667
> Validator	5	2	0	0	2	0	0	0.0	0.4	0.0	0	0.0

The two images above show the scores from CodeMR before the refactoring. These results represent our classes in the handlers package.

1) Refactoring the TransactionValidator class:

As you can see we had a problem mainly with the coupling, where we had more than 36% of the classes in the medium-high sector. From the table above it is clear that the TransactionValidator class has the worst scores, as it has very high coupling. This is why we decided to create a new class - TransactionsCommunicator. Its main idea is to be a “helper” class for the TransactionValidator by implementing some of the functionalities needed for the establishment of communication with the Request microservice. After introducing this class we managed to reduce the coupling to low-medium for the TransactionValidator class.

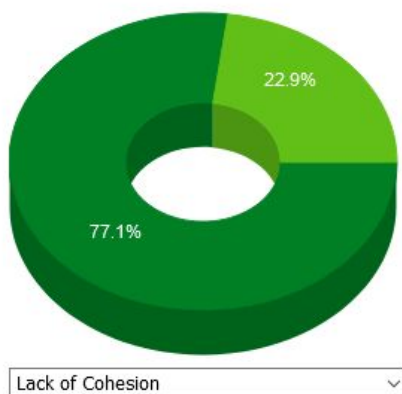
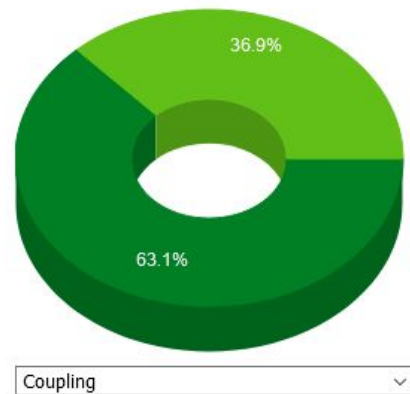
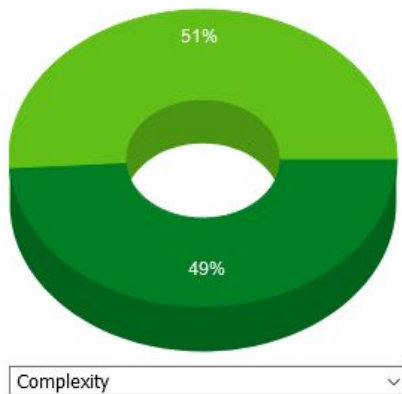
2) Refactoring the Base, House, Product, Tokens validators classes:

Apart from the TransactionValidator class, we can still see that the results for the coupling of the other classes are not very good. These scores inspired us to refactor the structure of the whole package. Before the refactoring we used to pass three separate variables throughout all validators. This is not considered as good practice and it increased the coupling in all of the classes. This is why we decided to create a new ValidatorHelper class that stores the variables we pass to all validators, as well as the most used

methods in the whole package. After implementing this class we managed to achieve low coupling in all of the other validator classes (except the TransactionValidator). On the two images below you can see the results for the handlers package after the refactoring. We managed to achieve a lot better results from CodeMR after refactoring the classes in the package and we now do not have any class with medium-high results for any metric.

Distribution of Quality Attributes

Complexity, Coupling, Cohesion, and Size



- Very High
- High
- Medium-high
- Low-medium
- Low

Name	Complexity	Coupling	Size	Lack of Cohesion	CBO	RFC	SRFC	DIT	NOC	WMC		
template												
nl.tudelft.sem.transactions.handlers	low	low-medium	low-medium	low						51		
> BaseValidator	low	low	low	low	5	17	7	1	4	5		
> HouseValidator	low-medium	low	low	low	5	20	11	2	0	8		
> ProductValidator	low-medium	low	low	low	4	20	9	2	0	7		
> TokensValidator	low-medium	low	low	low	3	21	5	2	0	3		
> TransactionCommunicator	low	low-medium	low	low	7	19	15	1	0	6		
> TransactionValidator	low-medium	low-medium	low	low	10	42	19	2	0	11		
> Validator	low	low	low	low	2	2	0	1	1	2		
> ValidatorHelper	low	low	low	low-medium	5	18	13	1	0	9		
Name	LOC	CMLOC	NOF	NOSF	NOM	NOSM	NORM	LCOM	LCAM	LTCC	ATFD	SI
template												
nl.tudelft.sem.transactions.handlers	157											
> BaseValidator	14	12	1	0	4	0	0	0.0	0.5	1.0	0	0.0
> HouseValidator	21	20	0	0	4	0	0	0.0	0.125	0.0	1	0.0
> ProductValidator	17	16	0	0	4	0	0	0.0	0.25	0.0	0	0.0
> TokensValidator	8	7	0	0	2	0	0	0.0	0.25	0.0	0	0.0
> TransactionCommunicator	24	22	1	0	4	0	0	0.0	0.6	0.0	1	0.0
> TransactionValidator	34	32	1	0	8	0	1	0.0	0.531	0.952	1	0.25
> Validator	3	2	0	0	2	0	0	0.0	0.333	0.0	0	0.0
> ValidatorHelper	36	32	3	0	8	0	0	0.667	0.7	0.893	0	0.0

Refactoring the methods in handlers package - Transactions microservice:

There are also some problems with the methods in the validator classes. On the image below you can see the methods before refactoring them:

Name	Complexity	Coupling	Size	Lack of Cohesion	CBO	RFC	SRFC	DIT	NOC	WMC	LOC	CMLOC	NOF	NOSF	NOM	NOSM	NORM	LCOM	LCAM	LTCC	ATFD	SI
template																						
nl.tudelft.sem.transactions.handlers	low	low-medium	low-medium	low						28	130											
> BaseValidator	low	low-medium	low	low	7	14	10	1	4	6	22	20	1	0	4	0	0	0.0	0.5	0.0	0	0.0
calculateCredits(Transactions, low	low	low-medium	low	low	3																	
calculatePortionsLeft(Transact low	low	low	low	low	2																	
checkNext(Transactions, Prod low	low-medium	low	low	low	5																	
setNext(Validator): Validator low	low	low	low	low	0																	
transient next : Validator																						
> HouseValidator	low-medium	low-medium	low	low	7	17	13	2	0	6	29	28	0	0	1	0	0	0.0	0.0	0.0	0	0.0
handle(Transactions, Product low	medium-high	low	low	low	7																	
> ProductValidator	low-medium	low-medium	low	low	7	15	12	2	0	4	16	15	0	0	1	0	0	0.0	0.0	0.0	0	0.0
handle(Transactions, Product low	medium-high	low	low	low	7																	
> TokensValidator	low-medium	low-medium	low	low	6	17	7	2	0	2	11	10	0	0	1	0	0	0.0	0.0	0.0	0	0.0
handle(Transactions, Product low	low-medium	low	low	low	6																	
> TransactionValidator	low-medium	medium-high	low	low	11	30	24	2	0	8	47	44	2	0	3	0	1	0.5	0.333	1.0	0	0.667
TransactionValidator(EurekaCl low	low	low	low	low	1																	
checkNext(Transactions, Prod low	medium-high	low	low	low	7																	
handle(Transactions, Product low	medium-high	low	low	low	9																	
transient discoveryClient : Eure																						
transient username : String																						
> Validator	low	low	low	low	4	2	0	1	1	2	5	2	0	0	2	0	0	0.0	0.4	0.0	0	0.0
handle(Transactions, Product low	low-medium	low	low	low	4																	
setNext(Validator): Validator low	low	low	low	low	0																	

From the image we can see that the coupling is medium-high for four methods - the handle methods in HouseValidator, ProductValidator and TransactionValidator and the checkNext method in the TransactionValidator.

1) Refactoring the handle methods in HouseValidator and ProductValidator:

To refactor these methods, we decided to split them, by implementing different functionalities in different methods. Furthermore, we used some of the methods from the ValidatorHelper class to further reduce the coupling. After the refactoring we managed to reduce the coupling for these methods from medium-high to low. Moreover, now all of the metrics for the methods in HouseValidator and ProductValidator are in the “low” category.

2) Refactoring the handle and checkNext methods in TransactionValidator:

Here we again refactor by splitting the methods. As the handle method in the TransactionValidator was with the highest coupling in the whole package, it needed more refactoring than the others. There we used a lot of the methods in the TransactionCommunicator. This reduced a lot the coupling and with the additional methods we implemented, we managed to achieve low-medium results for the coupling metric. For the checkNext method, after the refactoring we reduced the coupling from medium-high to low. Furthermore, for all other methods in the TransactionValidator class we got “low” in all metrics.

You can see the results after refactoring the methods on the image below:

Name	Complexity	Coupling	Size	Lack of Cohesion	CBO	RFC	SRFC	DIT	NOC	WMC	LOC	CMLOC	NOF	NOSF	NOM	NOSM	NORM	LCOM	LCAM	LTCC	ATFD	SI
template																						
nl.sudelft.sem.transactions.handlers	low	low-medium	low-medium	low						51	157											
BaseValidator	low	low	low	low	5	17	7	1	4	5	14	12	1	0	4	0	0	0.0	0.5	1.0	0	0.0
calculatePortionsLeft(Validator)	low	low	low	low	2																	
checkNext(ValidatorHelper): Flow	low	low	low	low	3																	
getHouseNumber(String): int	low	low	low	low	1																	
setNext(Validator): Validator	low	low	low	low	0																	
transient next: Validator																						
HouseValidator	low-medium	low	low	low	5	20	11	2	0	8	21	20	0	0	4	0	0	0.0	0.125	0.0	1	0.0
badRequest(): ResponseEntity	low	low	low	low	2																	
getProductUsername(Validator)	low	low	low	low	2																	
getTransactionUsername(Validator)	low	low	low	low	2																	
handle(ValidatorHelper): Response	low	low	low	low	2																	
ProductValidator	low-medium	low	low	low	4	20	9	2	0	7	17	16	0	0	4	0	0	0.0	0.25	0.0	0	0.0
badRequest(): ResponseEntity	low	low	low	low	2																	
badRequestDoesNotExist(): Response	low	low	low	low	2																	
calculatePortions(ValidatorHelper)	low	low	low	low	1																	
handle(ValidatorHelper): Response	low	low	low	low	3																	
TokenValidator	low-medium	low	low	low	3	21	5	2	0	3	8	7	0	0	2	0	0	0.0	0.25	0.0	0	0.0
badRequest(): ResponseEntity	low	low	low	low	2																	
handle(ValidatorHelper): Response	low	low	low	low	2																	
TransactionCommunicator	low	low-medium	low	low	7	19	15	1	0	6	24	22	1	0	4	0	0	0.0	0.6	0.0	1	0.0
TransactionValidator	low-medium	low-medium	low	low	10	42	19	2	0	11	34	32	1	0	8	0	1	0.0	0.531	0.952	1	0.25
TransactionValidator(EurekaClient)	low	low	low	low	1																	
badRequest(): ResponseEntity	low	low	low	low	2																	
checkNext(ValidatorHelper): Flow	low	low	low	low	3																	
getCredits(ValidatorHelper): Flow	low	low	low	low	2																	
getProduct(ValidatorHelper): Response	low	low	low	low	2																	
getUsername(ValidatorHelper): Response	low	low	low	low	2																	
goodRequest(String): Response	low	low	low	low	2																	
handle(ValidatorHelper): Response	low	low-medium	low	low	4																	
final transient discoveryClient:																						
Validator	low	low	low	low	2	2	0	1	1	2	3	2	0	0	2	0	0	0.0	0.333	0.0	0	0.0
handle(ValidatorHelper): Response	low	low	low	low	2																	
setNext(Validator): Validator	low	low	low	low	0																	
ValidatorHelper	low	low	low	low-medium	5	18	13	1	0	9	36	32	3	0	8	0	0	0.667	0.7	0.893	0	0.0

Last words

We had problems with finding code to refactor, since we applied from the very beginning clean code practices.

Moreover, the use of microservices architecture decreases class level lack of cohesion, as the functionality is clearly splitted between microservices. (The possibility of very large, incohesive classes decreases)