SEM Group 50

# Assignment 2

# Part 1 - Decisions

We used CodeMR to compute the code metrics of our project. After we generated the reports, we looked through all of the metrics and noted down all the yellow classes and methods, so that we could look into refactoring these methods. When we went through all of this, we saw we didn't have that many yellow classes, so we also considered classes which had low-medium issues, so we had enough to refactor.

After we had noted these down we decided to split up the work and we would all work in teams to refactor these classes and methods. This is when we saw that some things simply couldn't be refactored. An example of this are the entity classes, which gave low-medium issues with class cohesion. However, these classes just consisted of some getters and setters, which after we looked into this, we couldn't really refactor.
Another example of such a class that we couldn't improve all the code metrics is the SecurtityConfig file. In this file the coupling had medium-high issues, however that is the nature of the class we realised. This class is supposed to use a lot of the objects from the Spring framework, and there isn't really a way to work around this unless we had to completely change up our security and make our own security. This in the end will however be less secure, so therefore we have decided to not refactor this code.

At the end we decided to refactor the following:
Classes:

- FoodController - had a medium-high complexity and medium-high coupling. We wanted to reduce those to at least low-medium (to low, if possible).
- FoodRoute - had a medium-high lack of tight class cohesion. We wanted to reduce that to low-medium (to low, if possible).
- AuthorizationFilter and Authentication Filter - both had low-medium coupling. We wanted to reduce them to low.
- All validators - they had low-medium complexity and we wanted to reduce that to low.
- AuthenticationControllerTest - its complexity was low-medium and the size of it was medium-high. We decided to try to reduce the complexity to low and the size at least to low-medium.

Methods:

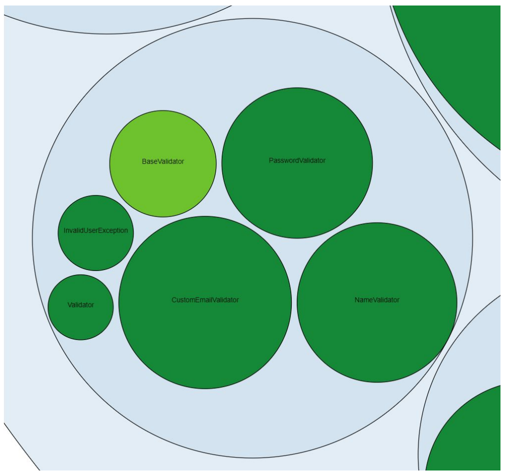
- AddFood - its coupling was very high. We wanted to reduce that at least to low-medium (we didn't want to set our threshold to low, because it uses chain of responsibility design patterns which increases the coupling of the method and cannot be removed).

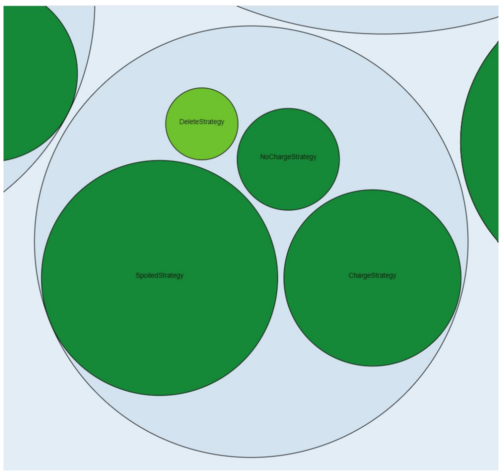
- changeName - it had a medium-high coupling. We decided to try and reduce that at least to low-medium.
- ConsumePortionsOfFoodByPeople - both its size and its coupling were medium-high. We wanted to reduce the coupling to at least low-medium and the size to low.
- DeleteFood - its coupling was medium-high. We wanted to reduce that to at least low-medium (we didn't want to set our threshold to low, because this method uses the strategy design pattern which raises the coupling).
- Handle (this applies to all validators) - the complexity was low-medium. Our goal was to reduce it to low.

Some other notable things we wanted to try to refactor the the number of children and the depth of inheritance tree metrics for the validators and the strategies (we were not certain, if those can be reduced).

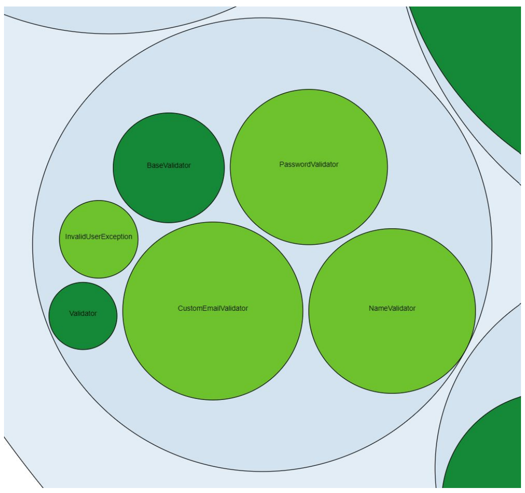
# Part 2 - Refactorings

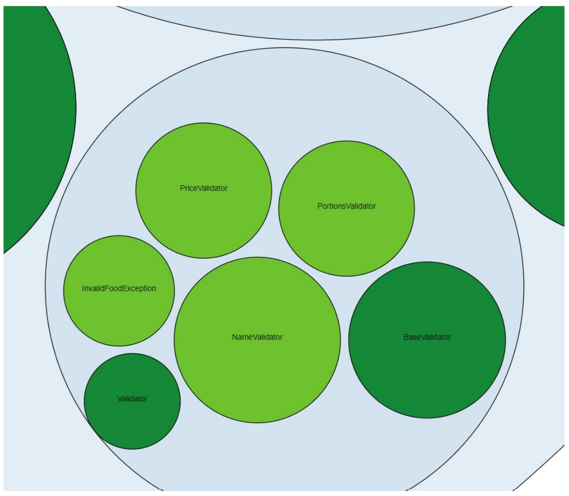
**Validators, Strategies - number of children metric codemr**





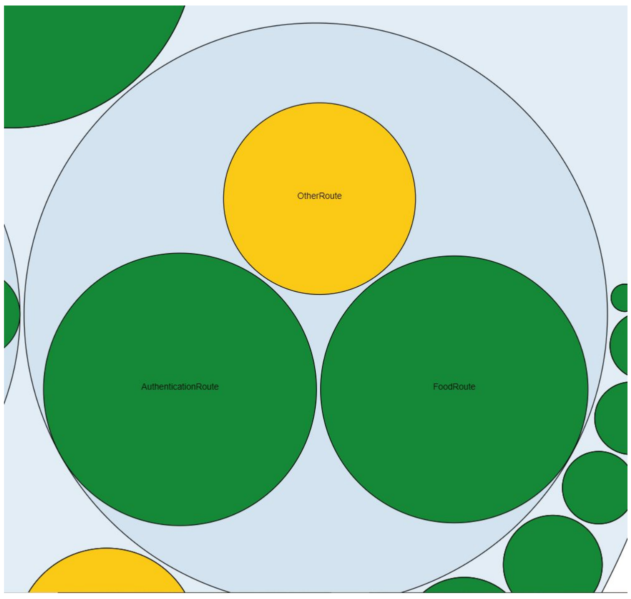
**Validators, Strategies - depth of inheritance tree**





The validators and the strategy classes showed low-medium levels for the Number of Children (BaseValidator, DeleteStrategy) and Depth of Inheritance. However, this is how the design pattern actually works, so the threshold for these metrics was set to low-medium due to the lack of changes we could have done without influencing the design patterns.
We have a medium-high level for the entities (User and Food) for the Lack of Tight Class Cohesion. This could easily be fixed by removing the unused methods of those classes, however, if we remove for instance the *setId(long id)* from the Food entity we will get pmdMain errors. That is because we can either have a transient, static attribute or we can have a normal attribute but it has to have a setter and getter set up. That is why we decided to leave the threshold for these classes to medium-high in order for our pipeline to pass.

**Other route - lack of cohesion of methods**



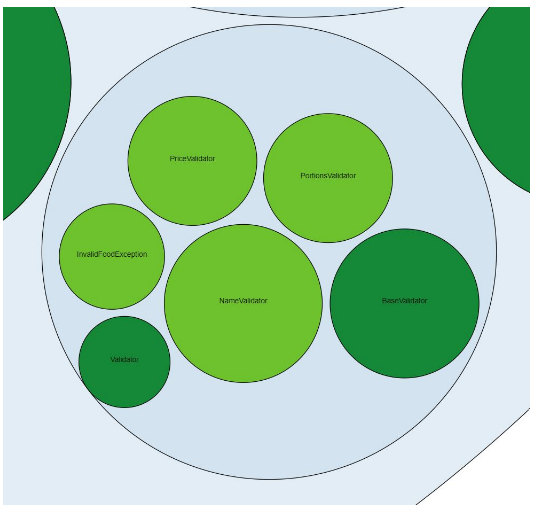
The reset method in the OtherRoute class in the gateway microservice initially had a medium-high level for the Lack of Cohesion of Methods due to the reset method which had some code duplication. It made an almost identical request twice, with only a small difference between the URL where this request had this go.

In order to refactor this, we created a separate method which sends this request. As parameters, this method has the URL the request needs to go to and the HttpServletRequest.

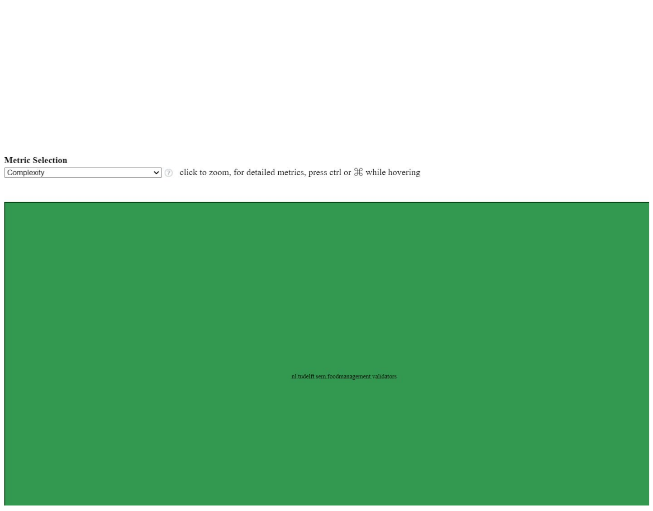
Even though this modification improved the size metric, we haven't managed to refactor this class in such a way that the metric decreases from medium-high. After looking at the method, we didn't really see how we can further refactor this class to improve the metrics.

### Validators - complexity

Before



After


**Metric Selection**

Complexity

click to zoom, for detailed metrics, press ctrl or ⌘ while hovering

nl.tudelft.sem.foodmanagement.validators

The validators have a medium-low complexity due to the if statements used in the handle method. We tried fixing and lowering the level to low by creating a separate method in which we would check the conditions of the if statements. By putting both if statements in the same method we observed that the complexity did not improve by a lot so we decided to move each if statement in a separate method. After the modification, the Lack of Tight Class Cohesion was very high, so we found that the methods we created for the checks were public and it not only made testing more complicated, but also played with the cohesion of classes. The solution we came up with was making them private so that only the class methods could actually use those checks.

Another refactoring we did was by introducing the design patterns. The chain of Responsibility pattern was included by creating separate validators, in the end refactoring the FoodController and AuthenticationController classes. This design pattern was used to improve the coupling of the classes and enhance the flexibility of the services implemented. The principle beneath the functionality of this design pattern is that there is only one class in charge of handling the chain, which makes it easier to use and to alter methods if necessary. The division over multiple classes makes the code easier to test and creates a clear structure. When testing this pattern, we only need to test the main method and not all the various functionalities of the validators. The size and all metrics which were related to the number of lines of code improved when this pattern was applied, because the methods clearly called the right functionality through the validators. The chain of validators has the purpose of processing the request by all consecutive validators and checking if the conditions of the validators are met.

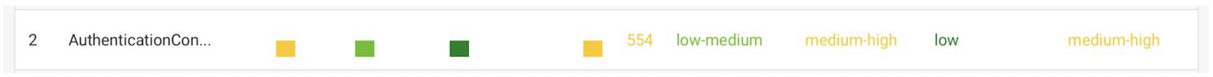
The Strategy design pattern we applied in our application refers to the deleting food functionality. When using this pattern, the code duplicity drastically decreases and creates not only a readable and understandable code, but also increases the flexibility. We chose to implement three different strategies: one that deletes the food that is spoiled by equally subtracting the credits from all users,  another that deletes the food that is not spoiled without changing any credits (maybe it was just a mistake adding it) and the last one that deletes the food that is not spoiled and subtracts the credits from the user. The optimal deletion strategy which will be used after processing the request is decided upon run time, making the functionality of our code a lot more flexible.

**AuthenticationControllerTest - size**

The issue with our testing code were the huge files with tests. An example of that was the AuthenticationControllerTest class that contained 710 lines of code and numerous test methods. It would be a nightmare to maintain that huge test cluster when we would for example change the way one method works, we would need to look over this huge file, wasting a lot of time. Hence, we decided to split the tests thematically into smaller classes. Now instead of one we have for files, each regarding specifically tests for registration, password manipulation, user management, and credit transactions. Hence, we lowered the size metric of the tests. Furthermore, we lowered the duplicated code in each test class by utilizing the BeforeEach annotation, and setting generally used mocks there.

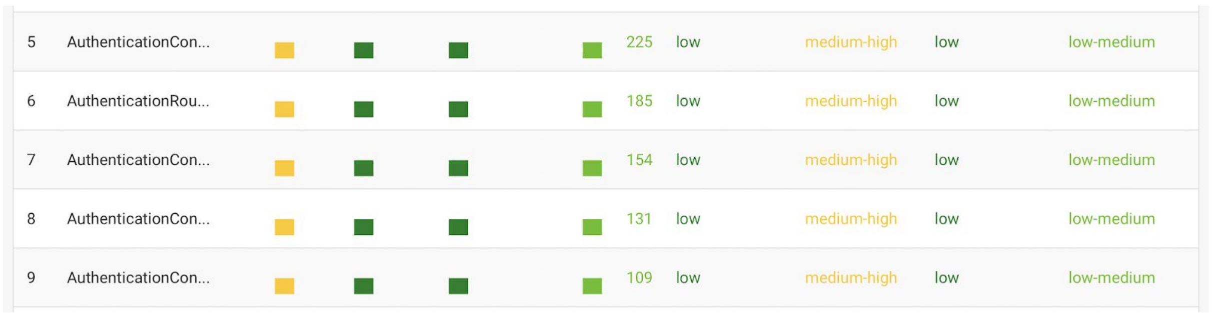
## Test screenshots

Before



| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 2 | AuthenticationCon... | 🟨 | 🟩 | 🟩 | 🟨 | 554 | low-medium | medium-high | low | medium-high |

After

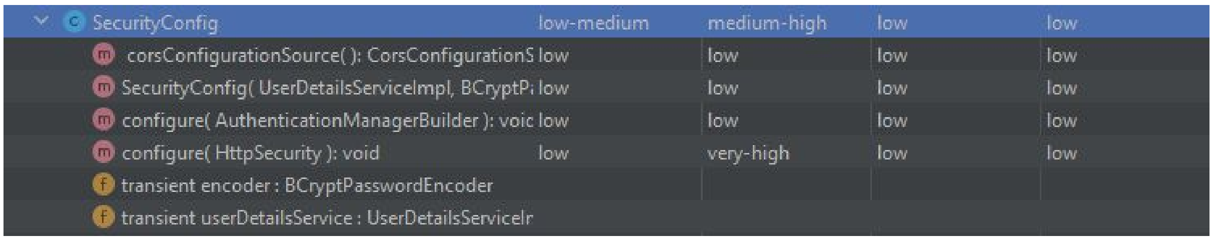


| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 5 | AuthenticationCon... | 🟨 | 🟩 | 🟩 | 🟩 | 225 | low | medium-high | low | low-medium |
| 6 | AuthenticationRou... | 🟨 | 🟩 | 🟩 | 🟩 | 185 | low | medium-high | low | low-medium |
| 7 | AuthenticationCon... | 🟨 | 🟩 | 🟩 | 🟩 | 154 | low | medium-high | low | low-medium |
| 8 | AuthenticationCon... | 🟨 | 🟩 | 🟩 | 🟩 | 131 | low | medium-high | low | low-medium |
| 9 | AuthenticationCon... | 🟨 | 🟩 | 🟩 | 🟩 | 109 | low | medium-high | low | low-medium |

CodeMR pointed out some issues with the complexity and coupling of our code. Although not critical, we got some medium-high metrics for this category with some methods so there was some room for improvement. One example of that was the SecurityConfig.configure method. It used constants from the SecurityConstants class which already contained numerous hardcoded strings that were used by other classes. The fact that it relied on said constants made the coupling in the configure method high. Though we were not able to lower the class coupling, we managed to lower the coupling of the configure method from very-high to medium-high as we simply replaced those constants with their string representatives, hence, the method relied on one less class. We couldn't refactor the SecurityConfig class on a class level, as it relied on too many other classes in order to do its job. Other issues with code complexity occurred in authentication validator classes (CustomEmailValidator, NameValidator, and PasswordValidator). In each, we had a long method that checked the correctness of the provided strings. This spaghetti code would be hard to maintain therefore we split each check into singular methods to make future maintenance easier. Furthermore, each method is a private one, so that they cannot be accessed by other classes, hence, providing encapsulation.

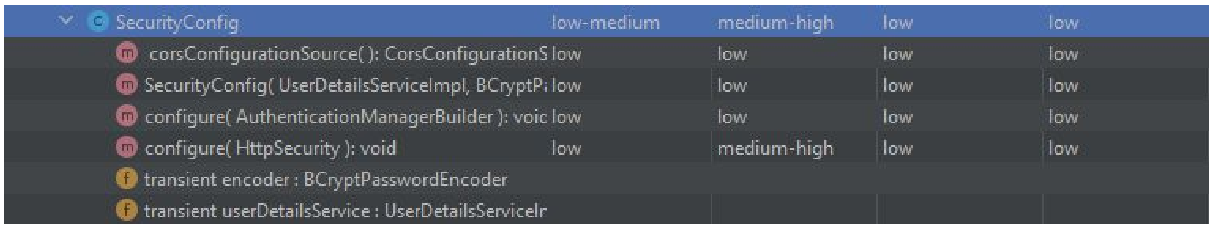
**SecurityConfig screenshots**

Before



| | | | | |
|---|---|---|---|---|
| SecurityConfig | low-medium | medium-high | low | low |
| corsConfigurationSource( ): CorsConfigurationS | low | low | low | low |
| SecurityConfig( UserDetailsServiceImpl, BCryptP | low | low | low | low |
| configure( AuthenticationManagerBuilder ): voic | low | low | low | low |
| configure( HttpSecurity ): void | low | very-high | low | low |
| transient encoder : BCryptPasswordEncoder | | | | |
| transient userDetailsService : UserDetailsServiceIr | | | | |

After

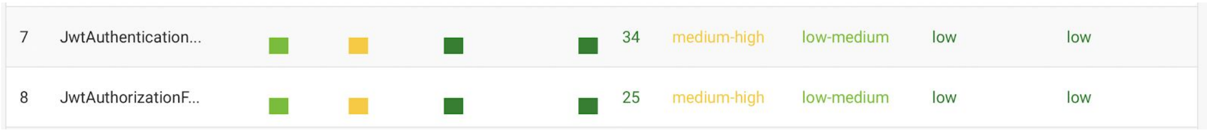


| | | | | |
|---|---|---|---|---|
| SecurityConfig | low-medium | medium-high | low | low |
| corsConfigurationSource( ): CorsConfigurationS | low | low | low | low |
| SecurityConfig( UserDetailsServiceImpl, BCryptP | low | low | low | low |
| configure( AuthenticationManagerBuilder ): voic | low | low | low | low |
| configure( HttpSecurity ): void | low | medium-high | low | low |
| transient encoder : BCryptPasswordEncoder | | | | |
| transient userDetailsService : UserDetailsServiceIr | | | | |

The next thing we refactored was the Authorization and Authentication filters. We tried lowering their complexity since it was medium-high, but it didn't fall down. Hence, we decided to lower their coupling on a class level. What we did was simply, take both filters and create util classes for them. Basically we made the methods that are being overridden, to call other util methods from other util classes, hence, lowering the coupling from low-medium to low.

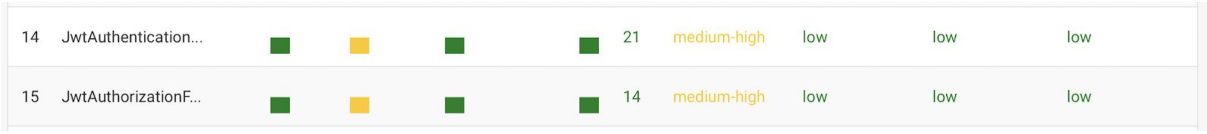
## Filters screenshots

Before



| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 7 | JwtAuthentication... | 🟩 | 🟨 | 🟩 | 🟩 | 34 | medium-high | low-medium | low | low |
| 8 | JwtAuthorizationF... | 🟩 | 🟨 | 🟩 | 🟩 | 25 | medium-high | low-medium | low | low |

After



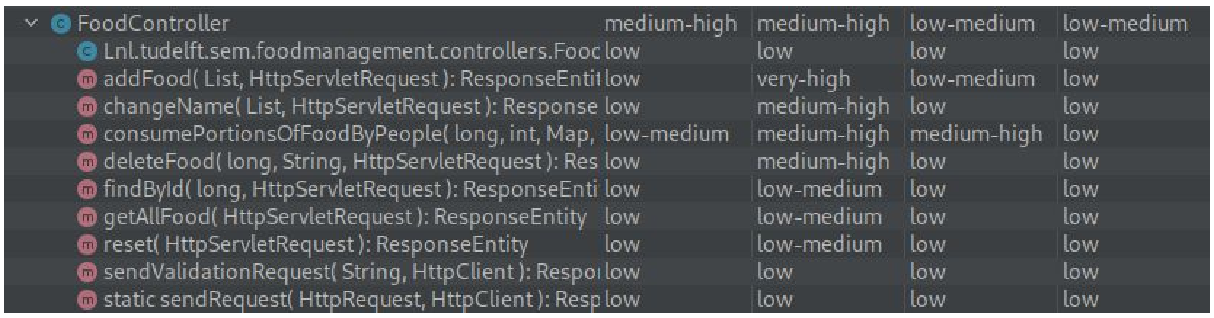
| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 14 | JwtAuthentication... | 🟩 | 🟨 | 🟩 | 🟩 | 21 | medium-high | low | low | low |
| 15 | JwtAuthorizationF... | 🟩 | 🟨 | 🟩 | 🟩 | 14 | medium-high | low | low | low |

In the first part of the project we have done some big refactoring regarding classes. To be more specific, we had to refactor the entire project to go from everything in one single folder and only having one microservice, to having multiple microservices. This meant that we had to first of all find the split and see which methods belonged there. This also came with seeing that there were methods that belonged to both.

There were some issues with code that used to work because it was all in one file and now didn't, so new methods had to be set up to make them work. To be more specific, the two microservices of course had to communicate with one another, so this code had to be written now.  In order to connect the two microservices, we have written a gateway API which will send the request to the correct microservice, but also can be made calls from one microservice to the other to make an update.

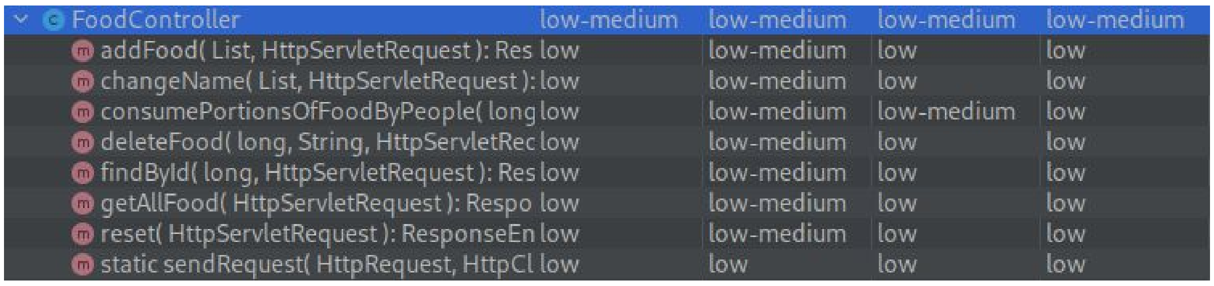
**FoodController screenshots**

Before



| | | | | |
|---|---|---|---|---|
| FoodController | medium-high | medium-high | low-medium | low-medium |
| Lnl.tudelft.sem.foodmanagement.controllers.Fooc | low | low | low | low |
| addFood( List, HttpServletRequest ): ResponseEnti | low | very-high | low-medium | low |
| changeName( List, HttpServletRequest ): Response | low | medium-high | low | low |
| consumePortionsOfFoodByPeople( long, int, Map, | low-medium | medium-high | medium-high | low |
| deleteFood( long, String, HttpServletRequest ): Res | low | medium-high | low | low |
| findById( long, HttpServletRequest ): ResponseEnti | low | low-medium | low | low |
| getAllFood( HttpServletRequest ): ResponseEntity | low | low-medium | low | low |
| reset( HttpServletRequest ): ResponseEntity | low | low-medium | low | low |
| sendValidationRequest( String, HttpClient ): Respo | low | low | low | low |
| static sendRequest( HttpRequest, HttpClient ): Resp | low | low | low | low |

After



| | | | | |
|---|---|---|---|---|
| FoodController | low-medium | low-medium | low-medium | low-medium |
| addFood( List, HttpServletRequest ): Res | low | low-medium | low | low |
| changeName( List, HttpServletRequest ): | low | low-medium | low | low |
| consumePortionsOfFoodByPeople( long | low | low-medium | low-medium | low |
| deleteFood( long, String, HttpServletRec | low | low-medium | low | low |
| findById( long, HttpServletRequest ): Res | low | low-medium | low | low |
| getAllFood( HttpServletRequest ): Respo | low | low-medium | low | low |
| reset( HttpServletRequest ): ResponseEn | low | low-medium | low | low |
| static sendRequest( HttpRequest, HttpCl | low | low | low | low |

Changes to the class:

We analyzed the FoodController class with codeMR. Initially, the complexity and coupling of the class were medium-high. Moreover, addFood method had very high coupling, changeName, consumePortionsOfFoodByPeople, deleteFood methods had medium-high coupling.  The complexity of the consumePortionsOfFoodByPeople was low-medium. We extracted some of the functionality into private methods because that method was quite long and honestly hard to understand from first glance. We managed to lower the complexity of the method, but in return increased the class complexity to medium-high. The next task was coupling. As we had implemented two design patterns last week, namely the Strategy pattern and Chain of Responsibility pattern. Both of them use a variety of classes and using those patterns inevitably led to an increase in the coupling of the class and methods. We decided that we can abstract some functionality in a class called Utils and make those methods public and static. That way we divide the complexity and coupling of the FoodController into two classes.

Methods changed:

1) addFood - we abstracted a method validateFood and moved it into the Utils class. This method is responsible to validate a given food entity based on some conditions and return true if the requirements were met. Additionally, we used another helper method called alterCreditsOfUsers, which based on given input changes correspondingly the balance of the users. This method is again a public static method in the Utils class. Coupling was reduced from very-high to low-medium because of less interaction with foreign classes.
2) changeName - we use the validate method that was previously mentioned to validate the new name.
3) consumePortionsOfFoodByPeople - This method consumes a given food based on a user list with their corresponding consumed portions. We abstracted a method called checkPortions.This ensures that the portions that were given do not exceed the portions that were available and the portions given are positive integers. Furthermore, we abstracted another method that retrieves all users from the database, this was needed to check whether the inputted users are valid. We made use of the aforementioned alterCreditsOfUsers to keep the correct balance of users.
4) deleteFood - We abstracted a method that instantiates a Deleter class based on a given strategy in String representation. This method is called instantiateDeleter and it is a public static method in Utils.