

# Assignment 2 - Software Engineering Methods

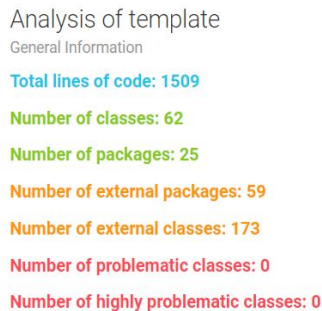
## Group 17A

Andrei Tociu, Pauline Hengst, Gijs van de Linde  
Iarina Tudor, Vincent van Vliet, Ravi Snellenberg

January 2022

# 1 Introduction

The purpose of this assignment is to analyze the code in order to improve the quality of our product regarding non-functional aspects. We chose to use CodeMR tool for computing the code metrics in our project. The general results were good, below is the picture of general analysis of template. We did not found high problematic classes.



Analysis of template  
General Information  
Total lines of code: 1509  
Number of classes: 62  
Number of packages: 25  
Number of external packages: 59  
Number of external classes: 173  
Number of problematic classes: 0  
Number of highly problematic classes: 0

We run different metrics on class and method level in order to find specific problems in our code. We identified some classes that were high couplers and bloaters, making the maintainability of the systems complicated. In the next sections, we detailed the results of computing the code metrics and what kind of refactoring we decided upon.

## 2 Class Refactoring

### 2.1 Class: `nl.tudelft.sem.Appliaction.services.ApplicationService`

#### 1. Result of metric tools :

For this class, we identify the large class code smell. For detecting this type of blob code, we used the metric **lack of cohesion of methods (LCOM)**. The result was 0.8756, labeled by CodeMR as 'high'. **Coupling** was also considered medium-high. We identify that `ApplicationService` class was implementing too many functionalities such as validating applications, requesting data from other microservices and implementing the logical behavior of the system. This led to lack of cohesion.

The motivation to decrease this threshold came from the fact that we believe that a class with too many responsibilities will be hard to debug, test and also make the system non-extensible. We want to maintain this class open for easily adding new functionalities and not too complex.

#### 2. Refactoring strategies :

We decided to firstly remove the unused attribute "portdata" from `applicationService` which improved LCOM to 0.8 (medium high). The next step was to use the extract class refactoring. We created a `validationService` class with now having a LCOM of 0. This refactoring also lowered the LCOM for application service to 0.5.

Moving methods that request data from other microservices from `applicationService` to a new class `RequestService` was not beneficial, as it increased `applicationService's` LCOM to 0.667. It also increased coupling since the methods left after splitting were dependant on each other. Therefore, we decided not to do this refactoring.

## 2.2 Class: nl.tudelft.sem.TAs.entities.TA

### 1. Result of metric tools :

We identified a low class cohesion using the metric **Lack of Cohesion of Methods (LCOM)** resulting in a score of 1.333 labeled by CodeMR as ‘very high’. This is the worst score we have got in all metrics, on all classes, but we decided it is not really relevant.

The metric scores high because this class has a lot of attributes (5), and only two methods : one of which uses only one attribute, and the other only three. Unlike other entities, TA’s attributes are being set using setters, which are generated using Lombok and thus are not not accounted in for the calculation of this metric.

### 2. Refactoring strategies :

Here are some refactoring ideas we had but decided not to put in action. Firstly creating a constructor with all attributes lowers LCOM to 0.667 (low-medium). This is not a valid refactoring because you’re not actually modifying the structure of the class, so not improving cohesion. Generating getters and setters instead of using Lombok lowers LCOM to 0.875 (medium-high). We decided using Lombok for testing purposes. Moving unused attributes to new class, lowers LCOM to 0.8 (medium-high). The new class has LCOM of 0. We decided not to do this, because TA class is an entity class, so extracting it would require modifications to the tables’ structures in the database.

## 2.3 Class: nl.tudelft.sem.TAs.controllers.TAController

### 1. Result of metric tools :

The code smell for this class is feature envy. We identified the problems by computing different metrics : **Coupling Between Object Classes (CBO)** with a score of 11 and **Lack of Cohesion of Methods (LCOM)** of 0.667 (medium-low). This class ended up highlighted “yellow” on chart for category “Coupling”. We believe this was due to the fact that TAController was really dependant on other classes, which reduces flexibility and re-usability of code. Of course, in order to have loose coupling, the cohesion will be high. We decided to have a balance between these two, and for this class, more independence is necessary.

### 2. Refactoring strategies :

We firstly created a new method in TA Service for adding a contract to a TA. Therefore, TA Service class took on the dependency on Contract Repository from the TA Controller. Secondly, we moved corresponding logic from TA Controller to TA Service. We had to refactor tests for TA Controller and TA Service. These steps were part of a class extraction, but we just realised some method would belong better to other existent classes rather than creating a new one. An important aspect to mention is that the method “addTimeSpent” from TA Controller was not moved into TA Service similarly to the previous one, because this would make the class TA Service too dependent on external classes and would thus “shift” the high coupling issue from one class to another. By refactoring only some of the methods, the dependency was better balanced between these 2 closely intertwined classes.

These changes resulted in lowering the CBO metric to 9 and LCOM to 0.444. Therefore, the TAController class ended up labeled as “green” on chart for category “Coupling”.

## 2.4 Class : `nl.tudelft.sem.Application.controllers.ApplicationController`

### 1. Result of metric tools :

The code smell for this class is feature envy. We identified this smell by looking at the **Coupling Between Object Classes (CBO)** , since this class was considered to have a tight-coupling. The score for the metric was 12. This class had 7 calls to the `ApplicationService` class and 0 to any of the other methods in the `ApplicationController` class. The service class had a CBO score of 19 (since it was calling the repository).

### 2. Refactoring strategies :

We used the strategy of extracting class. We moved the logic of problematic methods in the `ApplicationService` class and called them through the controllers. Then, we also moved the respective tests to the appropriate class for `ApplicationServiceTests` followed by a refactoring. This way, the controller class is now less dependant, loosing the coupling.

We also saw that this increased the CBO in `ApplicationService` to 21 (since there are more calls to the repositories), but that also lowered with the refactoring from 2.1. In the end, the changes we made reduced the net amount of dependencies and is thus overall a better state for the codebase.

## 2.5 `nl.tudelft.sem.User.security.SecurityConfig`

### 1. Result of metric tools :

We identified this class as a high coupler, having the smell of a feature envy. We computed the metrics **Lack of Tight Class Cohesion (LTCC)** which had a score of 1 considered very high and **Specialization Index (SI)** with a result of 0.8. This class is really important if we want to have a good security level of the application, thus we wanted to make it have a loose coupling to be easier to test, re-use and maintain.

### 2. Refactoring strategies :

We decided on extracting class. We firstly identified an unnecessary method. Therefore, we removed `userDetailsService()` method; This brought the specialization index down to 0.6, due to an `@Override` method being removed. We also updated `configureGlobal` to make use of self implemented `userDetailsService` implementation; This brought LTCC from 1.0 all the way down to 0.0. Now this class is more independent, thus easy to maintain.

## 2.6 nl.tudelft.sem.TAs.controllers.ContractController

### 1. Result of metric tools :

This class is another bloater with a code smells of large class and a bit of feature envy. We computed the class-level metrics **Size** with a score of low-medium and **Lack of Cohesion Among Methods (1-CAM or LCAM)** of 0.35. This are not alarming results, but since the contract functionality is something we would want to expand on by a lot in further iterations, we think it is good to make this refactoring as early as possible, to make sure we do not get stung by leaving it all in a single class. In the future the class should also become a problematic coupler.

### 2. Refactoring strategies :

We decided to use the extract class strategy, thus moving the modifying contract methods to a new controller class `ContractModificationController`. This is more extensible for the future, making testing easier and also lowers the coupling. Now the size of both classes are considered by CodeMR as low. The LCAM for the original class dropped to 0.15.

## 3 Method refactoring

### 3.1 Method : `nl.tudelft.sem.Application.controllers.ApplicationController:hireNStudents`

#### 1. Result of metric tools :

For this method we found the following code smell: complex method. We used the method-level metric called **Cyclomatic Complexity (CC)**. We reached a score of 7.

#### 2. Refactoring strategies :

First of all, we created a new private method named "hireStudents" that returns a Mono of a Boolean to refactor the code to. Second of all, we used extract method refactoring and extracted a part that was responsible for hiring the students to this new method and added the parameters (and thrown exceptions) to make it function as intended. We called this method from hireNStudents. In terms of tests, nothing needed to be changed.

After the refactoring, the cyclomatic complexity dropped to 3, while the new method has a CC of 5.

### 3.2 Method : `nl.tudelft.sem.Application.services.RecommendationService:sortOnStrategy`

#### 1. Result of metric tools :

For this method we found the following code smell: Object-Orientation Abuse, because of a Switch Statement. We used the method-level metric called **Cyclomatic Complexity (CC)**. We reached a score of 4 (1 + every strategy we add) since there is a switch statement. This score for now seems fine, but if we decide to add more strategies for recommending students, the complexity will increase and we don't want to have a too long method. We want to keep the method open to easily adding new features.

#### 2. Refactoring strategies :

We removed the switch statement by adding a new method to compare the input to the strategies name (adding the attribute "canonicalName"). In order to do this, the Strategy interface had to become an abstract class (for the purpose of adding the new method getStrategy with implementation provided). This does not interfere with the strategy design pattern as we had implemented it, as the interface was only being used to enforce the implementation of the method 'recommend()', which an abstract class can also do. The cyclomatic complexity stays the same (4) but now, with each future added strategy class, it will not increase.

### 3.3 Method : `nl.tudelft.sem.Application.controllers.ApplicationController:acceptApplication`

#### 1. Result of metric tools :

We diagnosed this method with "long method" code smell by looking at two method-level metrics. Firstly, we identified a **Cyclomatic Complexity (CC)** of 8 which is considered high. Secondly, we looked at the number of **Lines of Code (LOC)** which was equal to 37 (a big number compared to the average).

#### 2. Refactoring strategies :

The refactoring we did is extract method. We grouped functionality into two new classes : “isSelectionPeriodOpen” and “acceptApplication” in ApplicationService class. We also extracted parts of the method logic to 2 methods in ApplicationService which were more responsible for the tasks. After this steps, we created tests for the newly created methods and updated the old ones.

The refactoring resulted in better CC and LOC for the original method and also the new ones. We ended with a Cyclomatic Complexity of 5 for the original method , 3 for “isSelectionPeriodOpen” and 2 for “acceptApplication”. In terms of Lines of Code we lowered the number for the original one to 18. The two new methods had 9 and 12 LOC. This way the code is easier to debug and test branch coverage.

### 3.4 nl.tudelft.sem.User.security.JWTTokenService: newToken

#### 1. Result of metric tools :

We also identified this method as a ”long method”, specifically looking at the **Lines of Code (LOC)** metric. The newToken method had 24 lines of code, which was above average. This method was again part of the security feature, which we think is better to be as neat as possible. Our desire is to upgrade security, which would be really hard right now when the classes and methods responsible of this are unnecessarily complex, long and intricate. It will be also really hard to test, since we could not reach a high coverage on the USER microservice yet.

#### 2. Refactoring strategies :

We removed separate instantiation of DateTime object and also unused lines of code. We also had a separate unnecessary variable expiresAt which we decided to delete.

### 3.5 nl.tudelft.sem.TAs.WorkingHoursController: declareHours

#### 1. Result of metric tools :

We found another long method in our code. This time we looked at both **Lines of Code (LOC)** and **Cyclomatic Complexity (CC)**. The results were 31 for the first metric and 4 for the others. The feature of TA declaring hours is really important for students to receive correct payment and still needs to be polished. We believe that it has too many lines of code and it is hard to reach a good branch coverage testing. We will have to extend this method and right now it is challenging, thus we need refactoring.

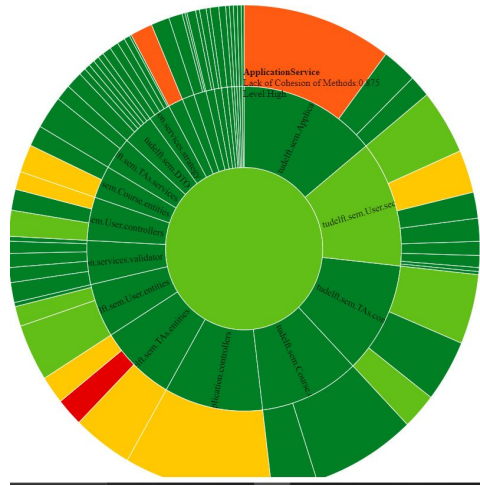
#### 2. Refactoring strategies :

We decided to use the extract method strategy. We moved some logic to a new method ”checkContract” in a more appropriate class (WorkingHoursService - this is a new class) Since we did not have a lot of WorkingHours functionality yet, having everything in the controller seemed sufficient, but when expanding code having logic inside a separate service is way preferred. The original method has now 24 lines of code and a CC of 2. The newly created method has a CC of 3 and only 11 lines of code.

## 4 Figures

Here we added 2 pictures of the metric results for the biggest problems we had.

This picture is from CodeMR when computing the metric lack of cohesion of methods. The light red Class is the `ApplicationService`, while the bright red is the TA entity we decided not to refactor.



In this picture we can see the results of the **Lack of Tight Class Cohesion (LTCC)** metric. The bigger the issues are in the Application microservice. The bright red stands for ApplicationService, RecommendationService and SecurityConfig.

