Group-Project-8-OP29-SEM57-Assignment-2

Timen Zandbergen
Merdan Durmuş
Luca Becheanu
Can Parlar
Alexandru Bobe
Matthijs de Goede

Task 1: Code Refactoring (30 pts)

1.1. Select a tool for computing code metrics in your project. Then, compute code metrics at the method and/or class level and identify which of them need to be improved by specifying and motivating the thresholds that allow you to identify them (15 pts).

In this assignment, we will be working with CodeMR to improve our code both on the method and class level. We start by running the analysis on the master branch. In the overall analysis, all classes have fair attributes, and no problematic classes are indicated by the program, so that's a great start. According to the lecture, some important class-level metrics for object-oriented code are covered by the Chidamber and Kemerer (CK) metrics:

- WMC = Weighted Methods for Class
- CBO = Coupling between objects
- RFC = Response For Class
- LCOM = Lack of Cohesion of Methods
- DIT = Depth of Inheritance Tree
- NOC = Number of Children
- CMLOC = Class Methods Lines Of Code
- Complexity
- Number of fields

Next to those, we'd like to consider two additional metrics that CodeMR has to offer:

- 1-CAM = Lack of Cohesion Among Methods
- 1-TCC = Lack of Cohesion between Tight Classes

With CodeMR we can produce coloured graphs indicating problem areas with relation to these quality attributes. The severe problems are indicated in red, and the less severe ones with orange and yellow. We will now give an overview of the state of these attributes in our entire codebase before and after refactoring under 'Metrics graphs before and after refactoring'.

As becomes clear from the graphs, the Weighted Methods for Class *(Figure 1)* is in good shape, which indicates that both the Cyclomatic Complexity of our methods and the

number of methods per class is alright. The Coupling between objects *(Figure 2)* is slightly high in the controllers. As they generally have a lot of tasks in the current implementation, we can try to decrease the coupling by splitting them up into multiple parts, each having fewer dependencies. The Response For Class attribute (*Figure 3*) is generally okay, except for the calendar service, where the controller seems to potentially invoke a large number of methods. Also, this can be tackled by splitting it up. The Lack of Cohesion of Methods (*Figure 4*), which measures how methods of a class are related to each other and give an estimate for whether the single responsibility principle holds seems to be a problem for a number of classes around the project, such as the Course entity on the calendar service and the Communicator on the calendar service, next to various classes on the Identity service. Maybe, some functionality is redundant or should be ported over to other classes to tackle this issue. The Depth of the Inheritance Tree (*Figure 5)* and the Number of Children (*Figure 6)* are alright, as there is no multilevel inheritance in our system. The complexity (*Figure 7*) seems alright, except for the calendar controller. The number of fields (*Figure 8)* and the number of lines in the class methods (*Figure 9)* also seem to be alright. Cohesion (*Figures 10 and 11)* seems to be a problem that we should definitely try to tackle. Although it seems hard in the rather specialized identity controller. For the controllers, splitting them up might help this. We should also check whether there are any getters and setters that can be generated with Lombok so that they don't decrease the cohesion.

1.2. Improve the quality of at least 5 methods (method-level code smells) and 5 classes (class-level code smells) identified in point (1) by applying proper code refactoring. Afterwards, recompute the metrics to ensure that the quality of that code has been improved successfully (15 pts).

Documentation Method-level Code Smells:

1. *Lack of Cohesion in Course entity due to constructor that makes no uses of some of the attributes:*

After reviewing the Course entity in the Calendar service, we came to the conclusion that the custom constructor used for testing decreased the cohesion of methods (LCOM) of the rather small entity class by not making use of 3 out of 4 class attributes. We initially increased the cohesion by adding these attributes to the constructor as well and changing the test cases accordingly. Later on, we noticed a more fundamental Class level problem with the Course class. It appeared to be an Object-Oriented abuser as it was storing a list of Attendances and we were only interested in the netIDs of those objects, whereas they also included an obsolete courseID. So each time we mapped the attendances to the corresponding netIDs. We refactored the entire class so that it now stores a list of Strings including the netIDs and this way we could remove the custom constructor and converter function and make the class a data class with just getters and setters created by Lombok. Because there aren't any methods remaining, the LCOM issue has been solved.

2. *Dispensable methods in the Calendar Communicator and Controller:*

When investigating the medium-high lack of cohesion of methods in the Communicator class on the Calendar service, we spotted a method that wasn't used anywhere in the code. The method setHttpClient was added to add support for another httpClient in the future, but as this became a won't have, we have now removed it. After removing it the cohesion of methods improved. When investigating the Calendar controller, we found another method that wasn't used; validate - as the validation is now done by a separate Validate class. The removal resulted in a similar improvement in the metrics.

3. *The cohesion of methods in the LectureScheduler:*

The LectureScheduler still lacks some cohesion of methods. However, the methods there are relatively small and dedicated already as functionality has been extracted into dedicated methods during the development process. We saw no way of improving the medium-high cohesion of methods further without changing the logic of our algorithm.

4. *Number of methods called in the Application class of identity:*

The method initUsers() had a high number of method calls (7), therefore we refactored the method such that some of these calls that were repetitive are now merged together. Some calls were not necessarily needed in the method and we decided to initialise them as variables before the method declaration.

5. *Lack of cohesion and complexity at restrictions services:*

While checking the CodeMR reports for the restrictions microservice, we observed that the controller and communication classes had cohesion problems. There were also some reports of low-medium complexity in the controller of the microservice. To tackle these problems, we divided the controller into three separate controllers; one dedicated for getters, one for setters and the last one for the capacity adjusters. However, dividing was not the answer to our problematic report. We also realized that there were many functions with a high number of method calls (7-10). In order to deal with these functions, we used the divided controllers and in some places created separate methods for specific functionalities. We also divided the methods in order to decrease the use of duplicated code and have as little communication with the repository as possible. For example, there was a method that takes the relatively-fixed (while the function runs) value of "percentage of allowed capacity for big rooms", inside its main for loop that goes over every room in the repository. We also made use of the shared repository of our system while using some constants. These improvements decreased the complexity and size of our microservice in general and helped the problem with cohesion.

Documentation Class-level Code Smells:

1. *Lack of cohesion for the LectureScheduler class:*

Throughout the development process of the application, we always paid attention to the quality of code. For example, we tried to refactor methods that were too long or too complex and this can be observed in the initial code metrics. Now, after running the *CodeMR tool,* some less obvious improvements were done to the code in order to eliminate the code's bad smells. One of our findings was the Dispensable code. We got rid of the getters and setters in the *LectureScheduler* class by using Lombok annotations, which greatly reduced the Lack Of Cohesion for this class. We succeeded in reducing the complexity of the class from “low-medium” to “low” and its lack of cohesion from “medium-high” to “low-medium”. We applied the same technique for the OnCampusCandidate class and the results improved for this class, too. The whole range of results and improvements can be seen in the attached diagrams.

2. *High coupler and blob code class CalendarController causing a lack of cohesion and high response for class:*

The Calendar controller was a rather big controller that dealt with 1) the scheduling of new lectures, 2) the retrieval of schedules for lecturers and students and 3) absence management. Instead of a single responsibility, the class now had 3 responsibilities, which explains the lack of overall cohesion. Since each of these tasks required different dependencies, the number of dependencies for this class was rather high, resulting in high coupling. By using class extraction and splitting the class into a Scheduler Controller, AbsenceController and two separate controllers to retrieve schedules for teachers and students, we managed to solve all of the three problems. We created separate tests to go with the new controllers.

3. *Lack of class cohesion in the Course-Management class:*

In the Course-Management class, we say that the ‘courseController’ and ‘lectureController’ had a high-medium lack of cohesion smell. These controllers are basically the backbone of the course management class, so refactoring them was important to us. The CourseController was related to all course-related problems, such as creating a course, deleting a course and retrievals. We have decided to refactor this controller into 2 controllers; The courseRetrievalController which takes care of all retrieval methods for course-related problems and a ‘courseController’ which creates and deletes a course. The same was applied to the lectureController, however, this controller was split into 3 controllers. The lectureRetrievalController which again was for retrievals of lecture related problems, a lectureController which takes care of deleting a lecture and a lecturePlannerController which plans the lectures for courses. By refactoring all of these controllers the lack of cohesion in these classes became low or low-medium.

4. *Lack of tight class cohesion in the Application class of identity:*

This metric measures the lack of cohesion between the public methods of a class. However, after taking a closer look to the class as a whole, we realised that some of these

methods don’t have to be public at all, and just removing their keyword solved the problem.

5. *Lack of cohesion in rooms and restrictions microservices:*

We also had problems with cohesion in both of these microservice in the class-level. The first thing we noticed was that these were caused by the entities in the services. We have realized that these entities had getters and setters we could have easily implemented via Lombok. We deleted these methods and used the Lombok builder instead. This improved our report in general. Then while checking the code thoroughly, we deleted some unused methods from communication and controller classes that were there because of possible future improvements. Then, we also made the helper functions package-private, because they were not supposed to be public. We divided our large controllers into smaller classes, where we had methods that use a common function in the same file. In rooms this meant that we have a class dedicated to CRUD operations and another for the listing of the rooms. In restrictions, this meant that we have a controller for getters, setters and capacity adjustions. This way, we decreased lack of cohesion and complexity.

6. *Bonus - Change preventing hardcoding of constants:*

In the battle against low cohesion, we also decided to include the Strings indicating the teacher and student role, as well as the “noAccessMessage” in the Constants file under shared entities so that they could be reused and code duplication could be avoided.

Problems that we couldn’t fix:

There were also some problems that couldn’t be fixed. The coupling of the newly introduced StudentRetrieval controller on the Calendar service seemed to be fixed locally, but after merging to development, it for some reason indicates medium-high without any line of code changed.

We had high coupling for the `UserController` in the Identity service, but unfortunately, there is not much we can do about this. The `UserController` acts as a wrapper around the rest of the service for the outside world and therefore requires the usage of multiple other classes from within the microservice. We were however able to reduce the lines of code used in the `validate` method. The same thing applies for the ‘SecurityConfig’ class, where, due to the fact that we are using the spring security framework, we have many overridden methods. That raises the Specialisation Index by a lot and we have no way of improving it.

We had medium Lack of Tight Class Cohesion in one of the restrictions controllers; the getter controller. This report metric could not be reduced more, while keeping the efficiency of the code and avoiding code duplication. All methods in there use a common getRestrictionVal method, via the common restrictionRepository, and dividing this controller or combining with others is not a valid solution.

Conclusion

All in all, we have made a fair number of refactorings in the code, mainly addressing the lack of cohesion among methods, classes and the high coupling. We have identified and resolved a number of code smells and were able to get rid of a lot of redundant code. Next to that, the functionality is now more evenly spread over some newly introduced classes. By changing the test cases accordingly, our test coverage even slightly increased!

Metrics Graphs before and after Refactoring

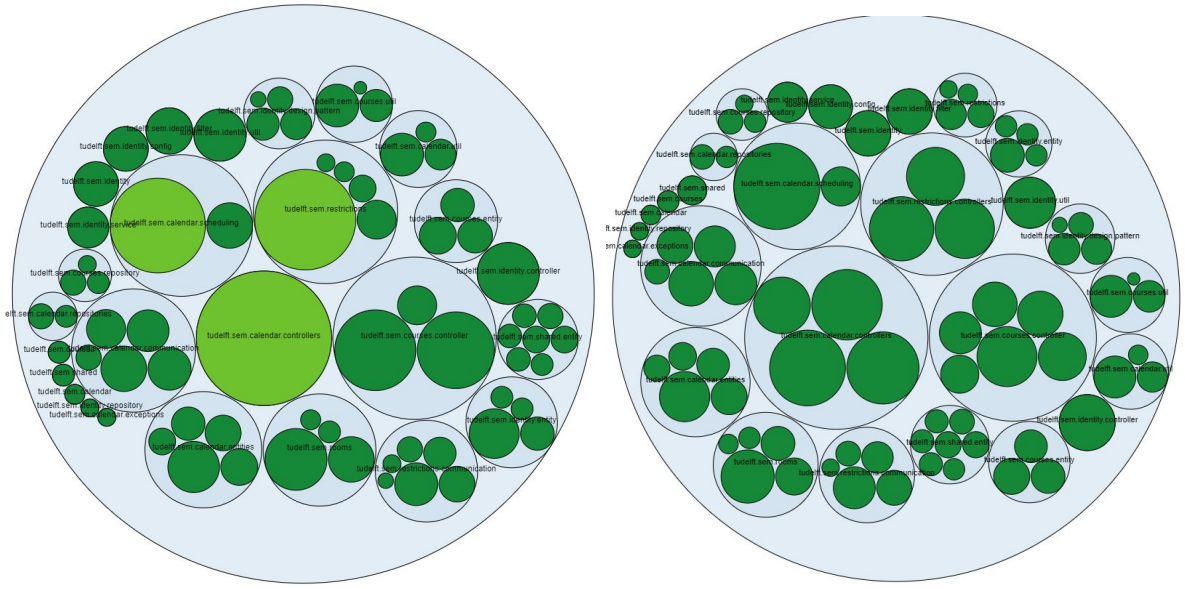

Figure 1: WMC

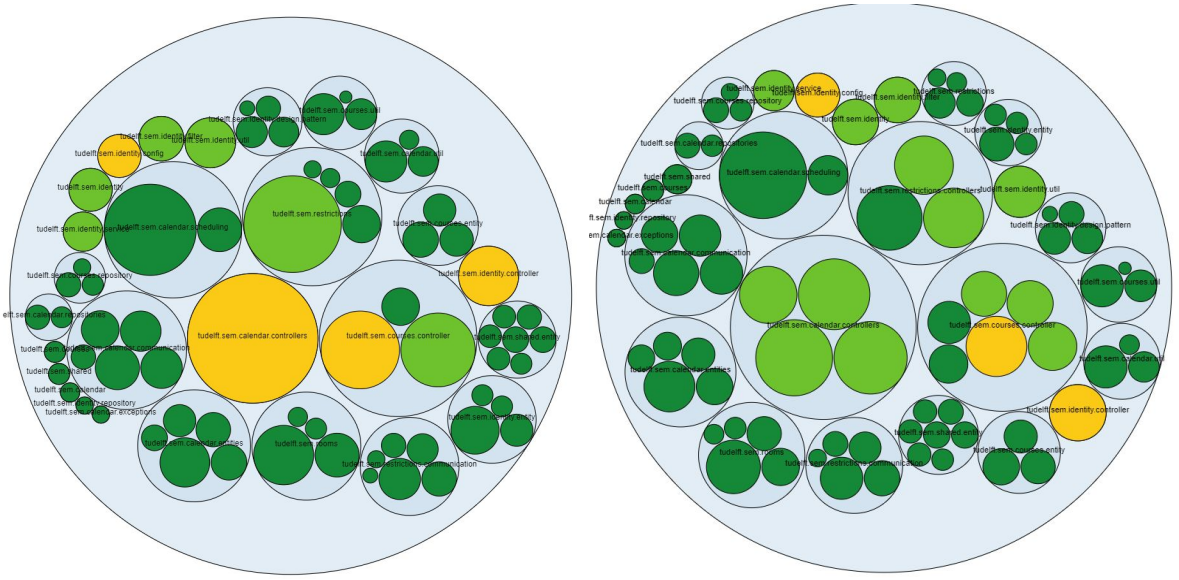

Figure 2: CBO

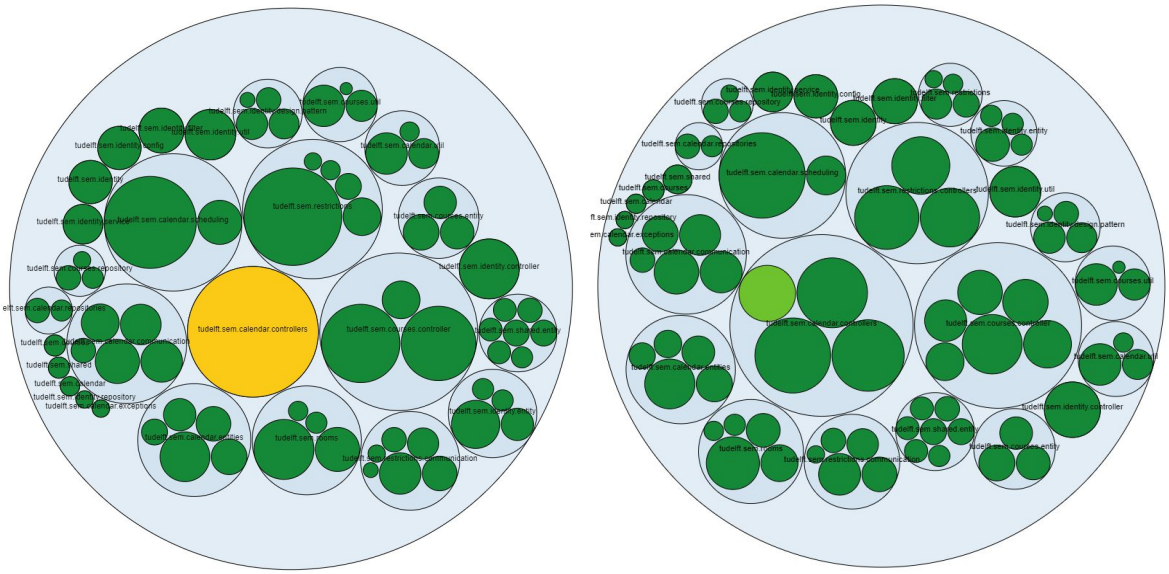

Figure 3: RFC

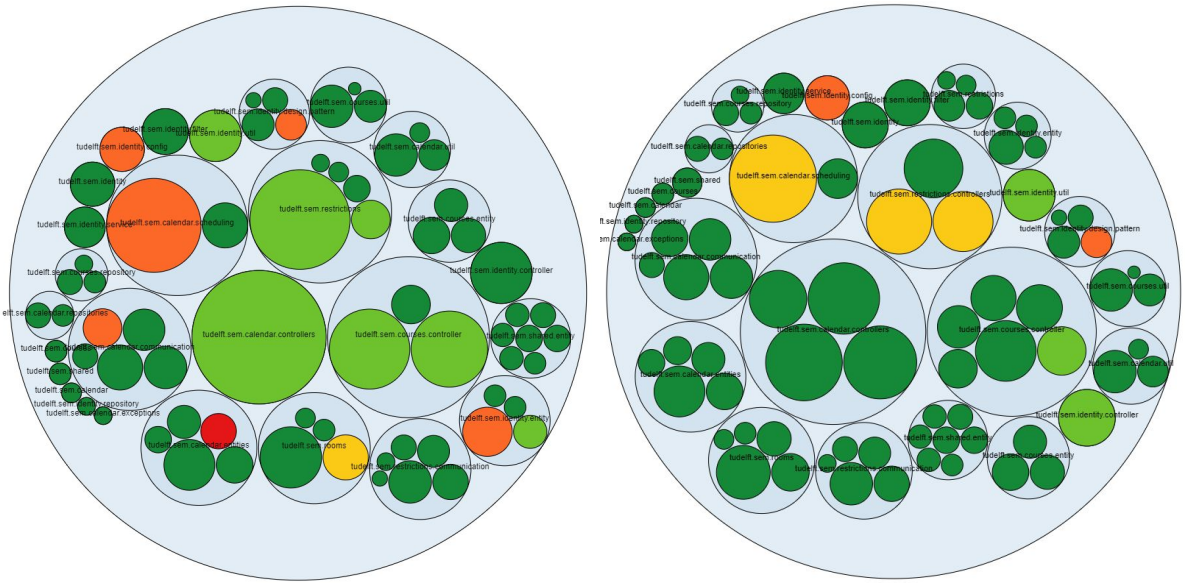

Figure 4: LCOM

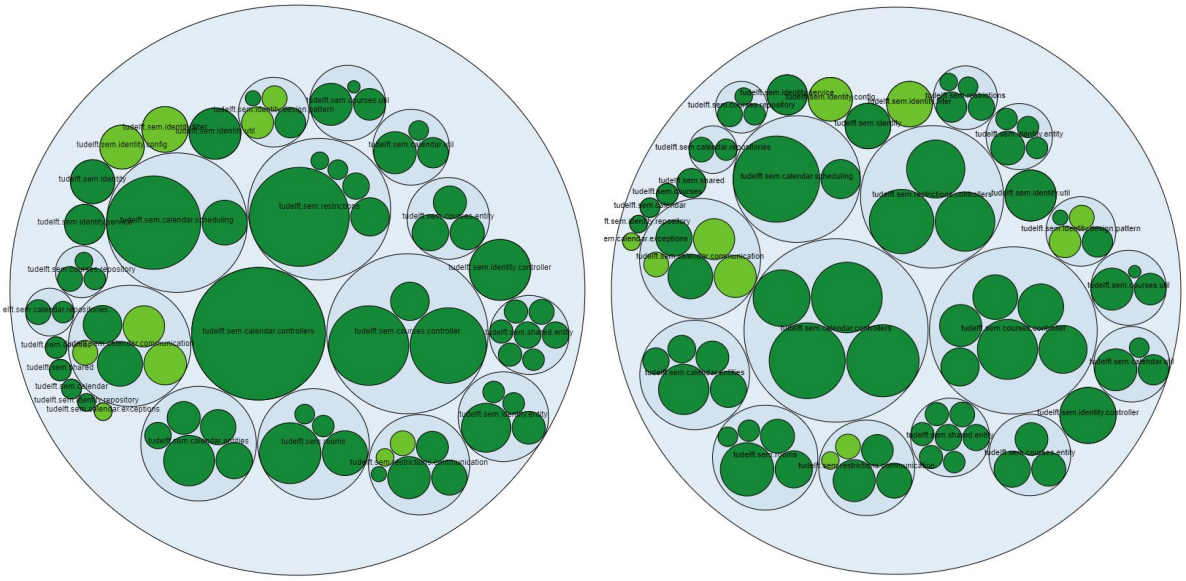

Figure 5: DIT

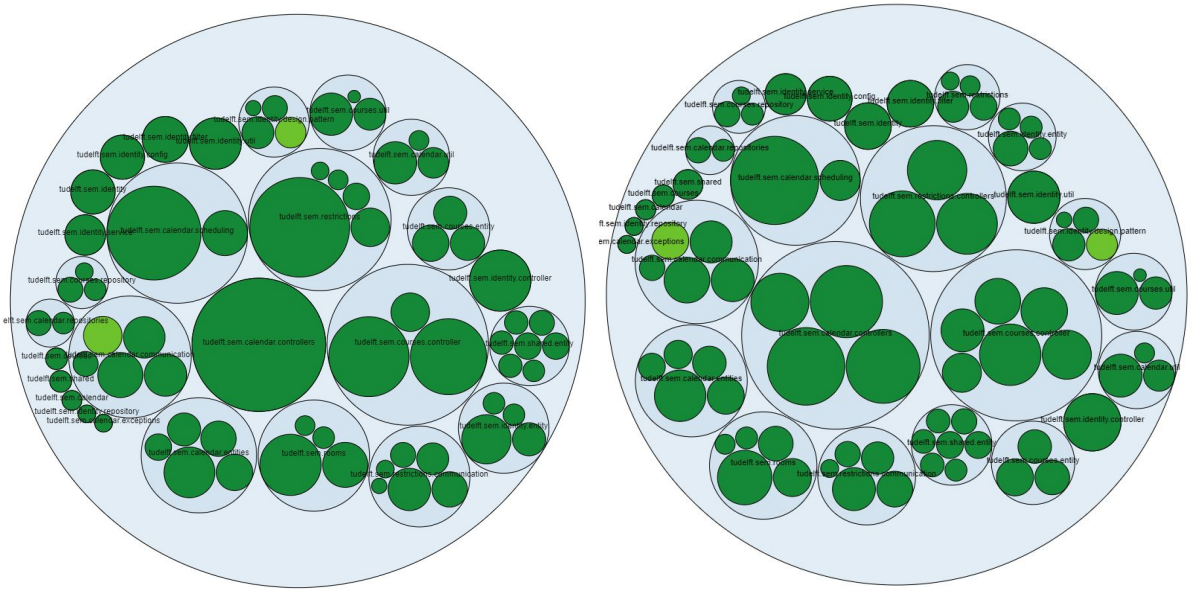

Figure 6: NOC

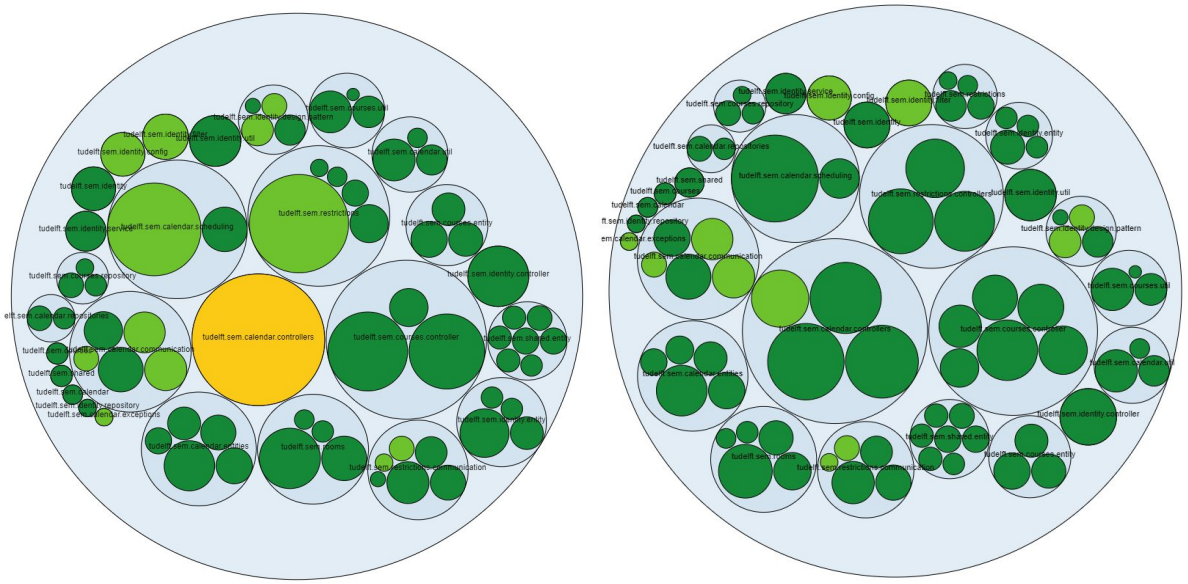

Figure 7: Complexity

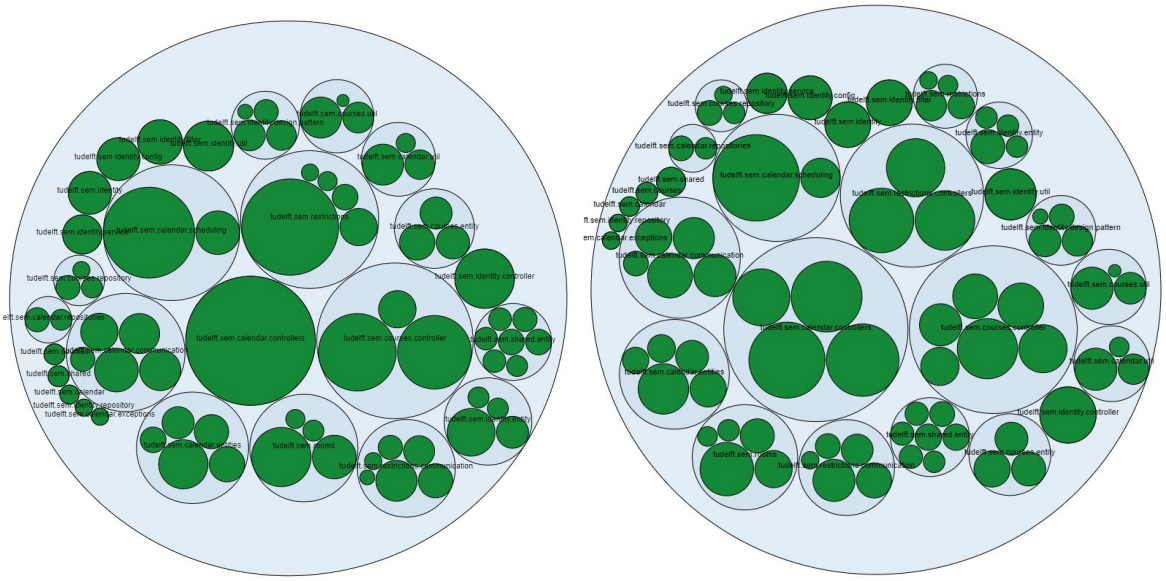

Figure 8: Number of fields

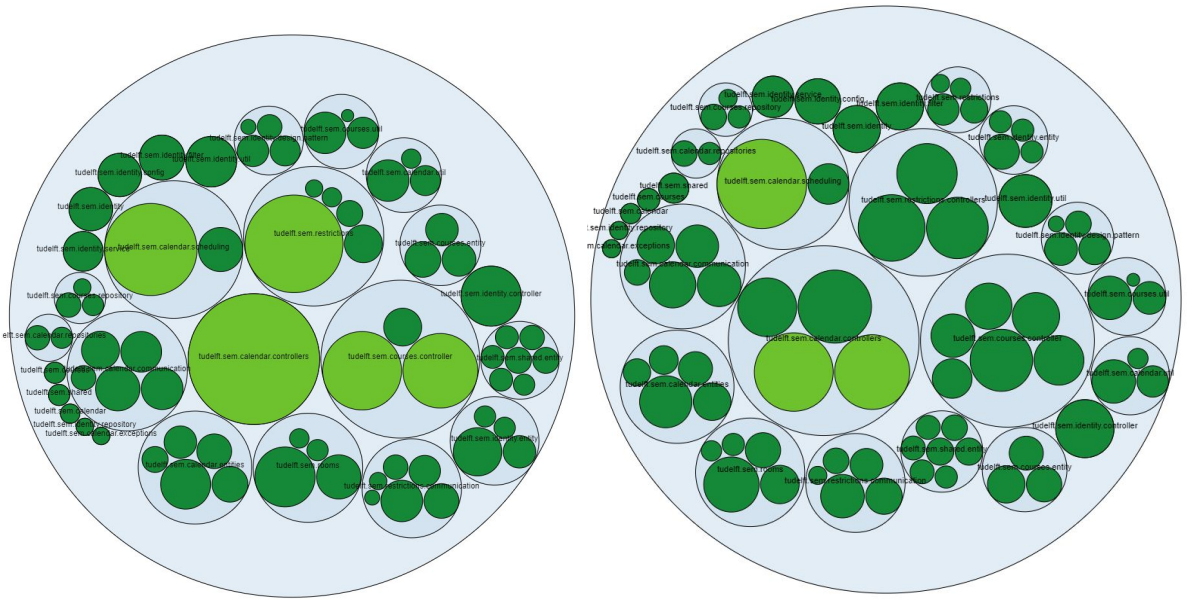

Figure 9: CMLOC

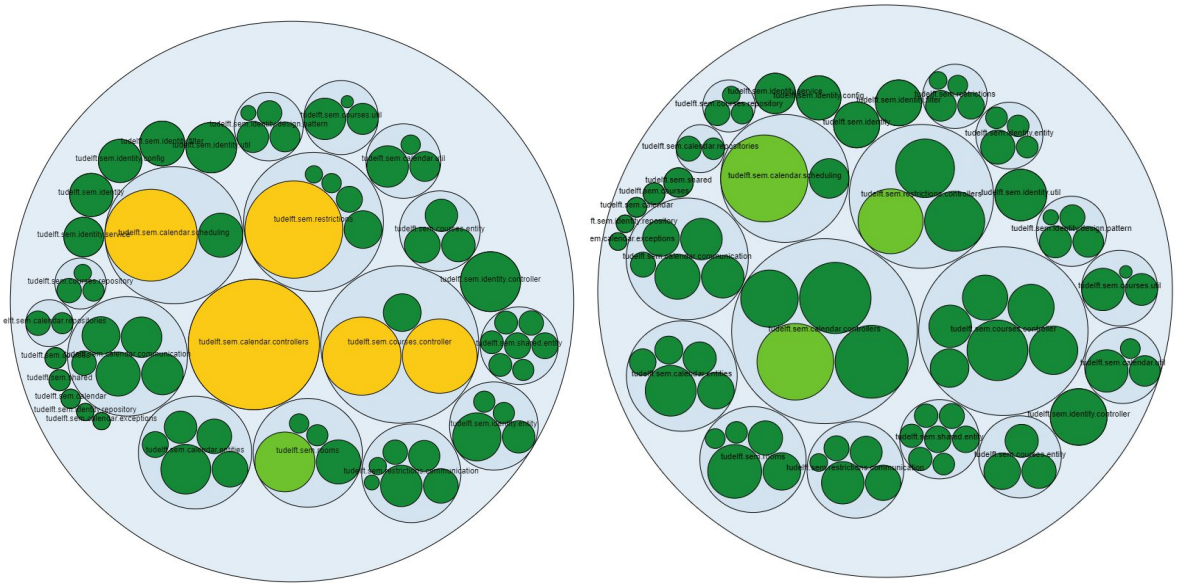

Figure 10: 1-CAM

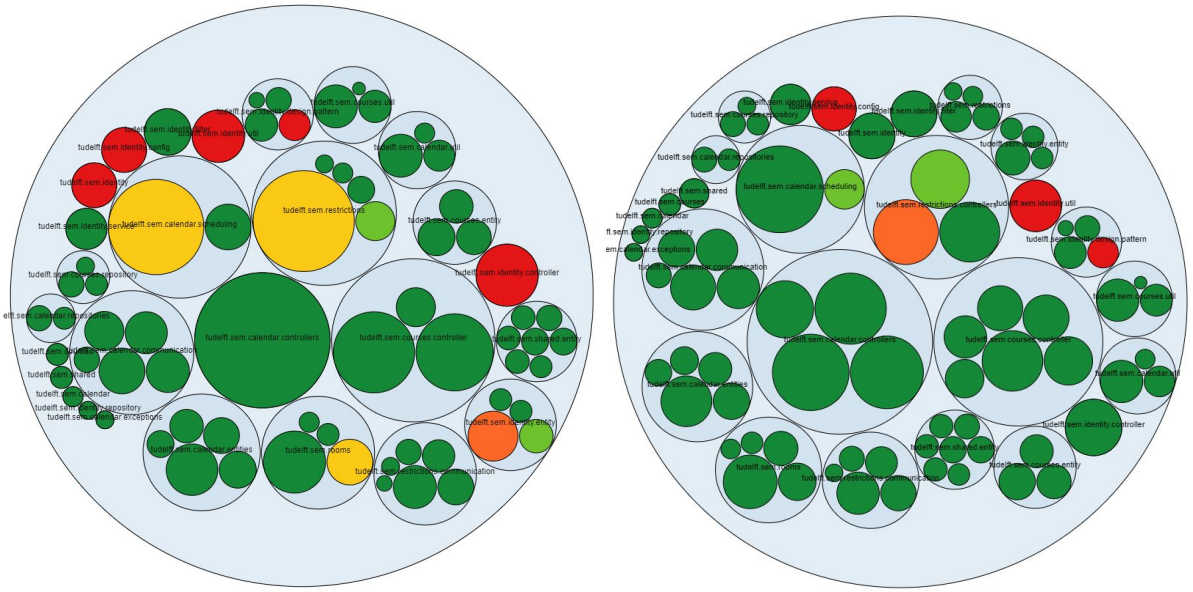

Figure 11: 1-TCC