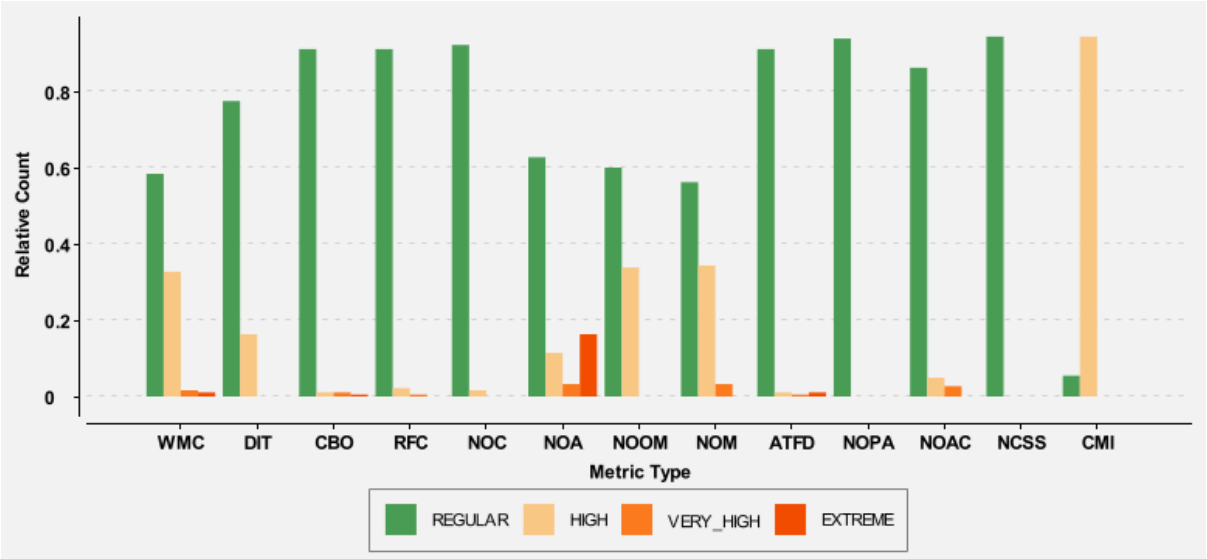
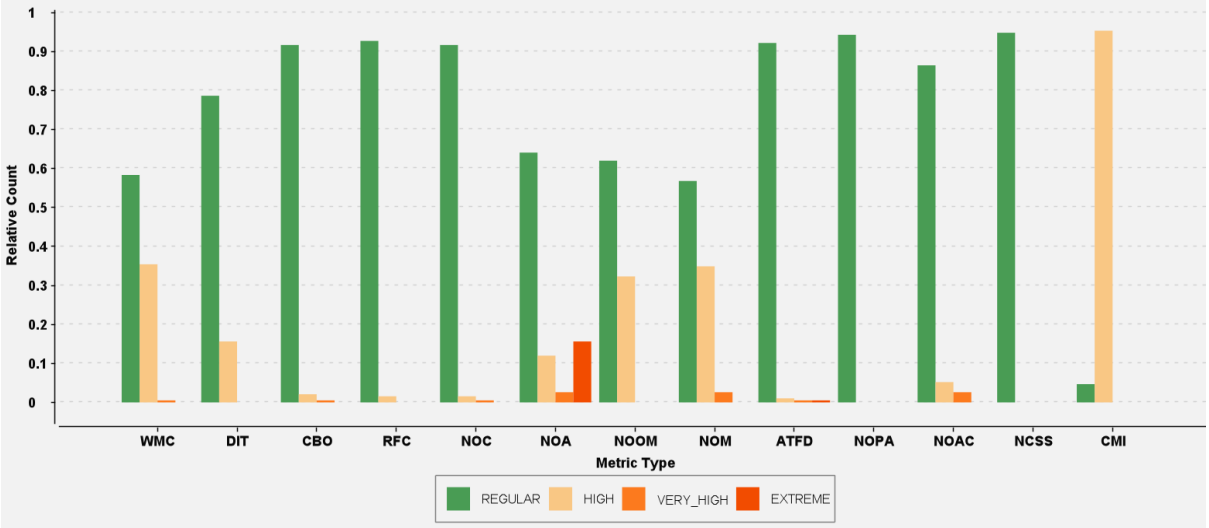
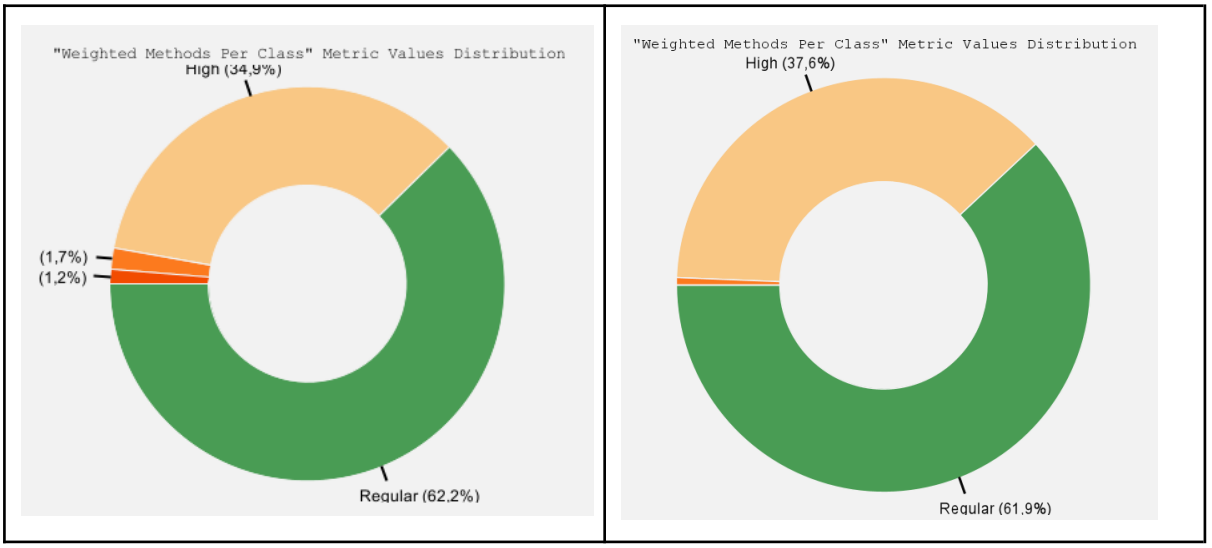
Assignment 2: 14a

Overview Metrics Before & After

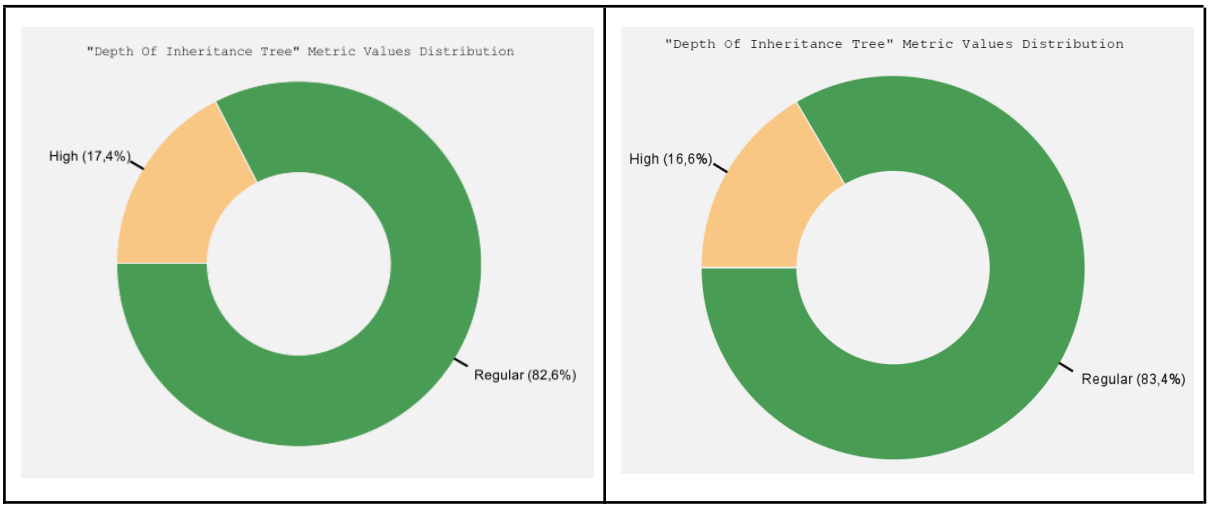




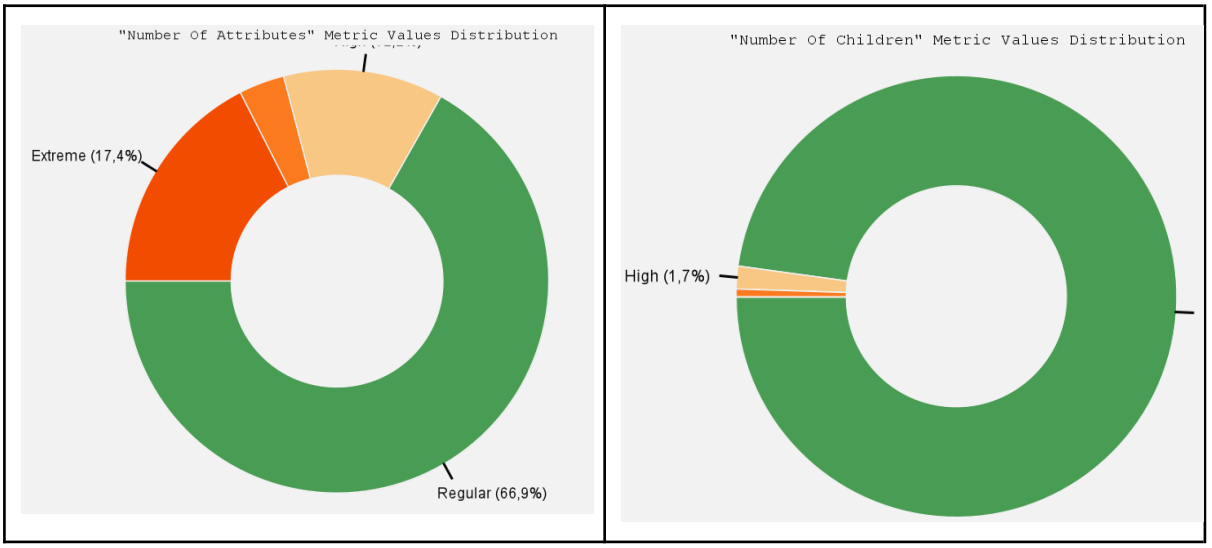
WMC: Weighted methods per class



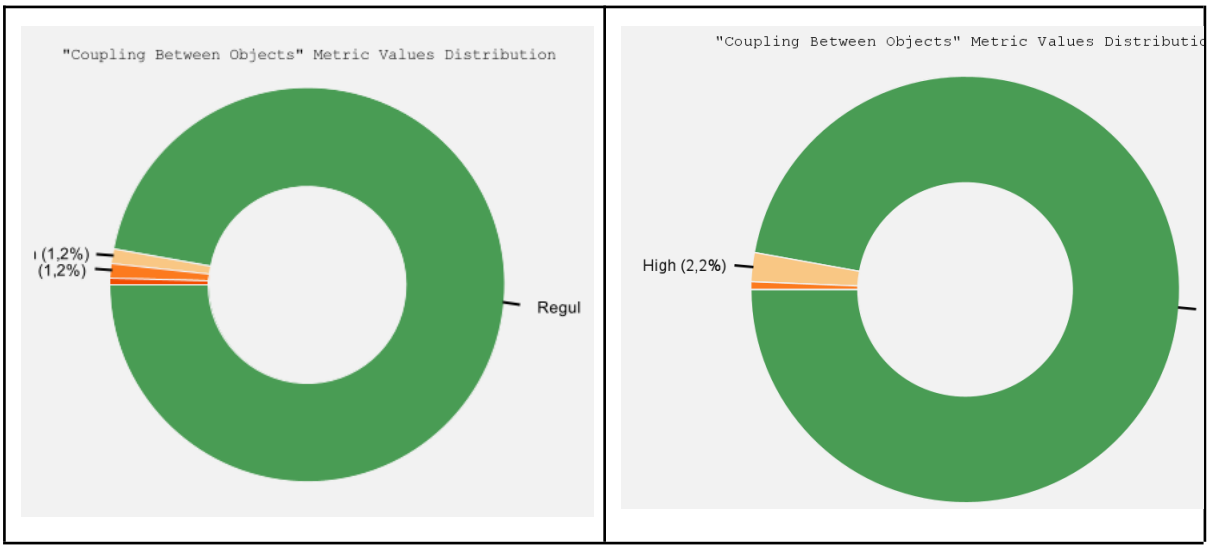
DIT: Depth of Inheritance Tree



NOC: Number of Children

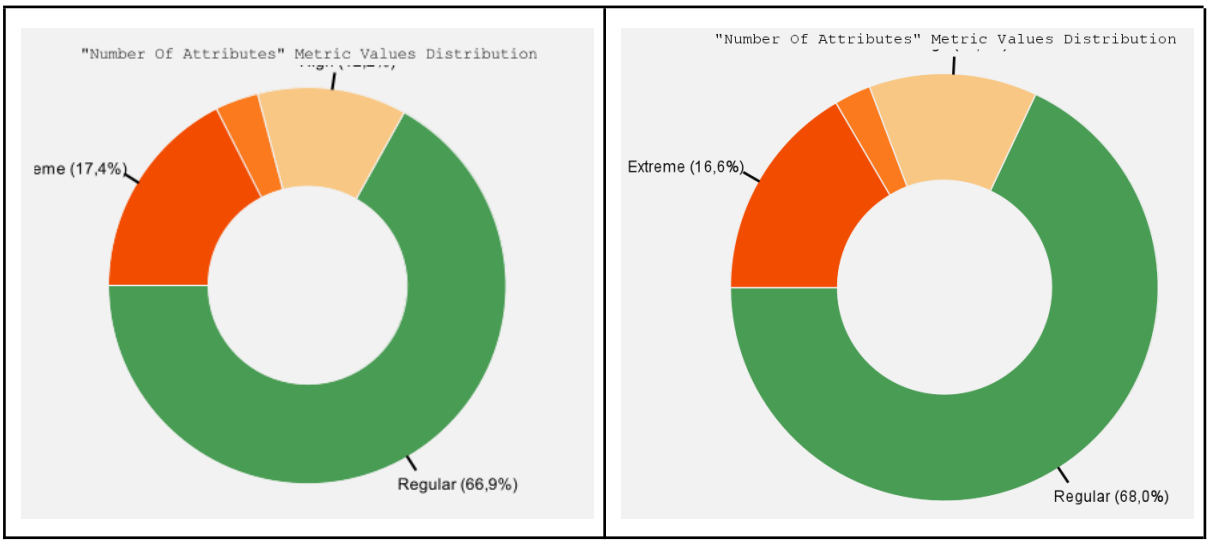


CBO: Coupling between object classes



RFC: Response for a Class

NOA: Number of Attributes



Pre Refactoring Analysis

The chosen tool for collecting the software metrics was MetricsTree. This was a natural choice because all team members used the same integrated development environment, thus simplifying the process of identifying problematic metrics. Furthermore, this specific tool provides the required Chidamber & Kemerer class-level metrics which generally provide useful insight for object oriented projects. Additionally, we also used a number of method-level metrics such as Cyclomatic Complexity (CC), Number of Parameters (NOP), Condition Nesting Depth (CND) and Lines of Code (LOC)

MetricsTree uses 4 categories to determine the severity of the problem, which are good, high, very high and extreme. We used the thresholds that MetricsTree used to label the problems to determine which methods and classes we wanted to tackle. We considered the metricsTree thresholds reliable and chose to maintain those automatically generated after cross-referencing with the papers written by Chidamber & Kemerer. Subsequently, we prioritised the refactoring of the methods and classes with very high and extreme problems.

Refactoring Documentation

1. Class: RequestSenderService

The request sender service is the worker behind our facade and encapsulates all the logic to do with authenticating and sending requests. This resulted in a class that became what could be considered a “god class”. When we decided to implement the facade design pattern we noticed that this was indeed a potential pitfall and therefore refactoring this class was a priority. The most glaring issues were:

- Extreme Weighted Method Count (WMC): 71
- Extreme Coupling Between Objects (CBO): 27
- Very High The Response for Class (RFC): 74

We applied refactoring by replacing delegation with inheritance, creating a separate class for processing requests to every respective microservice. A base request sender service is created that handles general requests that are not specific to any microservice with the same name as the original class.

Class	**WMC**	**CBO**	**RFC**
Original RequestSenderService	71	27	74
RequestSenderService	11	9	13
NodeRequestService	8	9	18
RequestsRequestService	18	9	25
ResourcePoolRequestService	37	22	57

We noticed that splitting the requests processing per microservice wasn’t enough to properly remove some of the bad code smells for the resource pool request service. Upon closer inspection, we noticed that this class seems to handle request processing and routing to the resource pool microservice but also handles a lot of the authentication and verification of the existence of a particular faculty.

At first glance, it would make sense to include all requests being made to a particular microservice in one class, however, from these software analytics metrics we can conclude that further refactoring is needed.

We then created a new class that handles the verification of the existence of a faculty and the authorization needed for interactions with the resource pool microservice. Furthermore, a new method was added to this verification service class to encapsulate all interactions with the faculty management service and the faculty verification service to reduce coupling between these classes as a whole. After we also noticed that there was a clear split inside the resource pool request sender class where a certain subset of methods related to scheduling which was delegated through the resource pool microservice and the other was related to requests interacting with the resource pool microservice itself. We split these into two separate methods for two primary reasons. Firstly, this large class implements too many functionalities and two distinct responsibilities and therefore violates the single-responsibility principle. Secondly, this large class is hard to maintain and control, particularly as more API endpoints are added.

This class extraction refactoring operation improved the metrics as follows:

Class	**WMC**	**CBO**	**RFC**
Original ResourcePoolRequestService	37	22	57
ResourcePoolRequestService	8	12	23
SchedulingRequestService	21	14	38
VerificationService	11	11	15

A. Method: authenticateGetResourcesRequest

After refactoring the RequestSenderService, this method ended up in the VerificationService. It authenticates a request to view the available resources for tomorrow. This method will return true in this list of cases.

- The user is a SysAdmin
- The user is a FacultyAccount, coupled to the requested faculty
- The user is an Employee, registered at the requested faculty

A lot of things can go wrong when making this request.

- The faculty can not exist
- The user can not exist
- The user can not be employed to the given faculty
- The user can be unauthorised

Due to this list being quite long, there was a very high cyclomatic complexity, as well as a lot of code lines.

- Very High Cyclomatic Complexity (CC): 6
- Very High Lines of Code (LOC): 34

We applied refactoring by extracting methods from this bigger method. All of the checks now happen in their respective methods.

Method	CC	LOC
Original authenticateGetResourcesRequest	6	34
authenticateGetResourcesRequest	4	25
checkFacultyExists	3	4
checkSysAdmin	3	1
checkFacultyAccount	2	6
checkEmployee	3	11

The checkFacultyExists method checks whether the given facultyId has a matching faculty. If that is not the case, it throws a FacultyException.

The checkSysAdmin checks whether the authorNetId belongs to a SysAdmin. If that is the case, the admin is allowed to see the available resources, so it returns true.

The checkFacultyAccount checks whether the authorNetId belongs to a FacultyAccount. If that is the case, the FacultyAccount is only allowed to see the available resources if they are the FacultyAccount of the given facultyId. The method returns true if that is the case.

The checkEmployee checks whether the authorNetId belongs to an Employee. If that is the case, the Employee is only allowed to see the available resources if they are employed at the faculty of the given facultyId. The method returns true if that is the case.

Since there are a lot of checks that need to be performed, we weren't able to make all of the numbers green. However, the Cyclomatic Complexity and the Lines Of Code are definitely reduced.

B. Method: createFaculty

After refactoring the RequestSenderService, this method ended up in the ResourcePoolRequestService. It sends a request to the resourcepool microservice to create a new faculty. This can only happen in the following cases.

- The user is a SysAdmin
- The user that is going to be the manager of the faculty is an Employee at the moment

A lot of things can go wrong when making this request.

- Something can go wrong with creating the faculty
- Something can go wrong with promoting the user to a FacultyAccount
- The user can not be a Sysadmin

Since there are a lot of levels things can go wrong at, there was a very high Condition Nesting Depth. The method is quite complicated as well, so there are a lot of lines.

- Very High Condition Nesting Depth (CND): 6

- Very High Lines of Code (LOC): 36

We applied refactoring by extracting methods from this bigger method.

Method	CND	LOC
Original createFaculty	6	34
createFaculty	2	22
sendCreateFacultyRequest	1	27

The sendCreateFacultyRequest is a method that handles all the logic on the sending of the request to the resource pool microservice. It returns a long, which is the facultyId of the recently created faculty. If something fails when creating the faculty, it throws a FacultyException.

After this change the Condition Nesting Depth was still high, so we extracted another method.

Method	CND	LOC
Original createFaculty	6	34
createFaculty	1	19
sendCreateFacultyRequest	1	27
checkEmployee	1	11

The checkEmployee method checks whether the given netId belongs to an employee. It returns true when that is the case.

2. Class: NodeManagementService

The node management service is responsible for registering and deleting nodes as well as performing the required verification for these operations by either querying the node repository or going through the chain of responsibility. This meant that there was a very high score for the coupling between objects (21) metric. Furthermore, the analytics also indicated an extreme number of parameters (6) and very high cyclomatic complexity (5) for the node registration method. As a result, refactoring needed to be performed to reduce these values to more sensible levels.

Upon inspection of the registerNode method, we noticed that the method was performing both the registration and the verification process. This is unlike the deleteNode method which uses a separate set of classes to perform the checks required to correctly remove nodes once they are live in the system. We quickly came to the conclusion that we needed

to do something similar to the registerNode method in order to help reduce the coupling between objects.

Move method refactoring was performed by creating a new class NodeVerifier with the sole purpose of verifying that node names, URLs, and tokens are unique in the repository and ensuring that the node provides a valid set of resources. With this in place, the coupling between objects was reduced to a more acceptable value of 16. By extracting all of the variables which needed verification into a single class, the number of parameters for the registerNode method could greatly be reduced by using this class as one parameter instead of all of the individual ones. With the addition of a few getters in the NodeVerifier class, the number of parameters was reduced to only 3.

The last order of business was to reduce cyclomatic complexity as much as possible. To solve this problem, we decided to separate the checking of names, urls, tokens and resources into four separate methods. This choice was made to increase the flexibility of the code base by separating the logic in such a way that allows new verification steps to be inserted without much hassle. This resulted in the cyclomatic complexity being reduced to 2 for each method.

The table below shows the Cyclomatic Complexity and the Number Of Parameters of the refactored methods. It can be seen that the values for these metrics have been reduced drastically. The only exception to this is the cyclomatic complexity of the checkResourceRequirements. This is because all of the individual parameters of a resource need to be checked in order to satisfy business logic so there is not much that can be done to reduce it even further. All of these changes helped reduce the class level metric of Coupling Between Objects from 21 to a much more acceptable value of 16.

Method	CC	NOP
checkNameExistance	2	1
checkUrlExistance	2	1
checkTokenExistance	2	1
checkResourceRequirements	5	1
registerNode	1	3

3. Class: NodeContributionRequestModel

A. Method: equals

The NodeContributionRequestModel is a class we use as model for the objects we transfer between two different microservices. We annotate this model with the @Data annotation

which automatically creates like getters, setters, equals and toString. The problem here was that the automatically generated equals method used all fields and had an ***extreme cyclomatic complexity of CC 13*** which also made the class have a very high WMC of 35. We fixed this by Overwriting the the automatically generated equals method with a custom one because the uniqueness of a node can be determined by the name alone, so our custom method only checks if the names are equal. This reduces the CC to 3 witch is very good. This also automatically reduces the WMC to 20 which solves the issue in the class as well.

4. Class: RegistrationService

The registration service is a class that deals with registering requests and updating their status (approved / rejected). Because of that the class was trying to do too much and had a **very high** method count of 17 and a **severe** Weighted Methods count of 54. We **extracted** 2 classes from it - the RequestHandler and the RequestChecker. The RequestHandler is responsible for searching in the requests repository. The RequestChecker, on the other hand, calculates if the status of a request has to be updated. By extracting those classes we achieved better separation of logic and lowered both metrics dramatically.

Class	Method Count	WMC
Original RegistrationService	**17**	**54**
new RegistrationService	10	23
RequestHandler	5	12
RequestChecker	4	18

A. Method: registerRequest

We chose to refactor 2 methods in this class. The first is registerRequest which is responsible for handling any request for resources when it gets submitted. That includes checking the validity of the resources within the request (throws an exception if resources requested are not valid), calculating a lot of variables that are important when deciding how to handle the request, deciding how to handle the request with those variables, and then finally taking the necessary actions to handle the request. Due to the long and complicated process just described, this method ended up having an ***extreme cyclomatic complexity of CC 13***, as well as an ***extreme LOC of 56***. In order to reduce both of these metrics, we extracted the latter part of the method into another method registerRequestOnceStatusDecided concerned with the handling of the request, once it has been decided how to handle it. This reduced the LOC to 30 which is pleasing and the cyclomatic complexity to 6 which is a much more acceptable value, considering that this method is concerned with the most complicated part of our business logic.

Method	CC	LOC
Original registerRequest	**13**	**56**
new registerRequest	6	30

B. Method: decideStatusOfRequest

The second method is decideStatusOfRequest which gets called by registerRequest and is concerned with *making the decision* of which of the 4 following categories a request falls into when being submitted:

- Automatically accepted and scheduled
- Automatically rejected
- Left for manual approval rejection by the faculty manager
- Left pending until the six-hour deadline (before the end of the day) when all the unused resources go to the free resource pool and the request can be reprocessed.

The method had an ***extreme cyclomatic complexity of 15*** since the calculations of which category a request should fall into were the core of the scheduling business logic and were pure propositional logic consisting of 3 or 4 boolean conditions. To reduce the cyclomatic complexity, we extracted those calculations into smaller, easier to maintain and understand methods: isRequestRejected, isRequestAutoApproved, and isRequestDelayedUntilSix. This reduced the cyclomatic complexity to 4, which is a drastic improvement from 15.

Method	CC
Original registerRequest	**15**
new registerRequest	4

C. Method: processRequestInPeriodOne

This method is concerned with processing pending requests after the 6 pm deadline. The method checks the available space on the free resource pool and calculates the new status (whether it is accepted or not).

The method had very high cyclomatic complexity of 5 and very high line count of 40. There was, however, a lot of code duplication and unnecessary branches. I removed the unnecessary branches and the duplicated code and that reduced to 2, which is excellent. Furthermore, I **extracted** 1 helper method, which calculates whether the resource pool has

space left and **reused** a second method. That reduced the line count to 14 (excluding JavaDoc).

Method	CC	LOC
Original processRequestInPeriodOne	**5**	**40**
new processRequestInPeriodOne	2	22
getStatus	1	16
hasEnoughResources (reused, was in the code before)	3	15

5. Class: PromotionAndEmploymentService

The promotion and employment service concerns itself with promoting regular users to a faculty or sysadmin account and employing and firing users from faculties. We noticed that it had a very high Number of Attributes (NOA), this is mostly because there are many other services and repositories that it relies on.

Step 1 in the refactoring process was removing attributes relating to legacy code that has since been moved from the class.

Secondly, we introduced a parameter object that would represent a data clump and replaced the references where necessary. We also made the parameter object class immutable and gave the attributes private final access. This way we can use the relevant attributes without having a very confusing and long parameter list. If we want to in the future extend our application and include other services, for example, to check the number of requests an employee has made, we could use this same UserServices parameter object data clump to simplify our application and increase maintainability. We reduced the NOA from 8 to 3.

6. Class: RpManagementService

A. Method: deleteNode

The RpManagementService does all the interaction with the Rescource Pools database, we noticed that this class had a method, deleteNode, with a very high cyclomatic complexity of 5. When looking at this method we noticed that we were not using methods that already exist in the system and optimise the process of subtracting resources. We used these methods and also extracted some of the validation steps like checking if the facultyId exists and if the system has enough resources to be removed. These changes lowered the CC to a very reasonable 2. We also applied the same changes to the similar method contribute node.

This wasn’t necessary but we wanted to stay consistent. The changes to this method lowered the CC from a 3 to a 1.