

1. Assignment 2

Task

Exercise 1	1
Method Level	1
Class Level	3
Exercise 2	6
Method Level	6
Class Level	13

Acceptable Ranges & Motivation

1. *McCabe Cyclomatic Complexity* - [0, 4] - We have chosen this acceptable range based on observation, and we decided that a cyclomatic complexity of 4 would be the maximum accepted, both for readability reasons and for testing complexity reasons.
2. *Number of Parameters* - [0, 4] - This acceptable range was derived from a group discussion, where we considered possible readability and maintainability consequences of being above such values. In the end, we settled on 4 as the maximum number of parameters a method should have.
3. *Number of Methods (NOM)* - [0, 14] - This range was derived based on class and test class length and maintainability. For 14 methods, assuming an average length of 10 lines, the number of lines will be around 140, and the test suite with possibly more than 150 lines, if comprehensive. These are the values we consider to be the maximum acceptable.
4. *Weighted Methods Per Class (WMC)* - [0, 25] - This range was derived with the same process of the previous one.
5. *Number of attributes (NOA)* - [0, 9] - We chose this range in order to ensure that the class has a clear, recognizable and relatively small purpose.

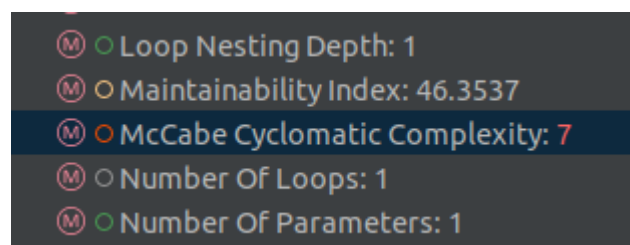
6. *Coupling Between Objects (CBO)* - [0,13] - We chose this range to ensure that one class and its method is not used too many times by others.
7. *Response For Class (RFC)* - [0, 52] - This range was recommended by the tool we have used and we agreed that it is appropriate.
8. *Lack of Cohesion of Methods (LCOM)* - [0, 10] - We chose this range to ensure that our classes stay cohesive and have a clear purpose
9. *Depth of Inheritance Tree (DIT)* - [0, 3] - We chose this range since decreasing it for classes such as *NotEnoughResourcesException*, or *JwtRequestFilter* was unfeasible, as inheritance is necessary for these classes to properly function, and the classes that they inherit from are outside our influence.
10. *Number of Children (NOC)* - [0, 3] - We chose this range in order to limit the consequences that come with having a lot of children classes. With many children classes, the likelihood of misused abstraction of the parent class will increase.

To calculate the metrics we have used IntelliJ plugin *MetricsTree*.

Exercise 1

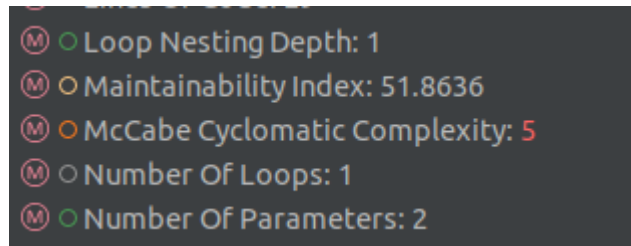
Method Level

1. By running class metrics in *user-microservice/src/main/java/nl.tudelft.sem.template.user/controllers/MainUserController.java* we found that the *addNewUser* method had values for *McCabe Cyclomatic Complexity* that were out of the acceptable range.

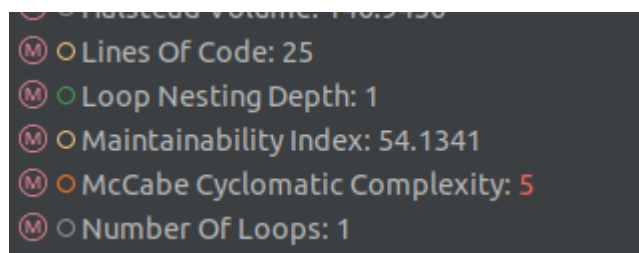


2. By running class metrics in *faculty-microservice/src/main/java/sem.faculty/handler/FacultyHandlerService.java* we found that the *acceptRequests* method had values for

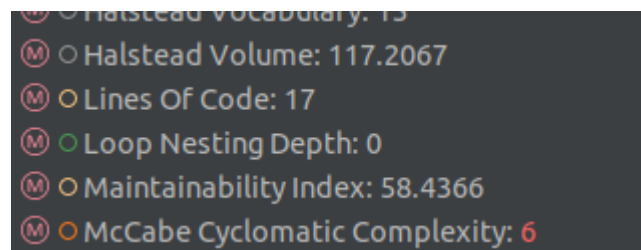
McCabe Cyclomatic Complexity that were out of the range we consider acceptable.



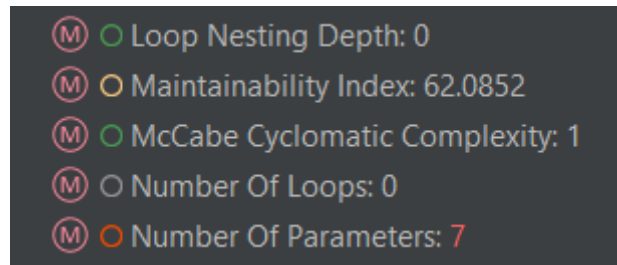
3. By running class metrics in *resource-manager-microservice/src/main/java/nl.tudelft.sem.resource.manager/domain/services/DateSchedulingService.java* we found that the *getDateForRequest* method had values for *McCabe Cyclomatic Complexity* that were out of the range we consider acceptable.



4. By running class metrics in *commons/src/main/java/sem.commons/Resource.java* we found that the *checkResourceValidity* method had values for *McCabe Cyclomatic Complexity* that were out of the range we consider acceptable.



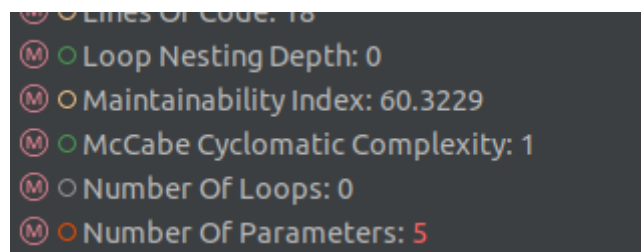
5. By running class metrics in *faculty-microservice/src/main/java/sem/faculty/domain/Request.java* we found that the constructor went over the number of parameters we considered acceptable. Therefore, we have decided this method should be refactored.



A screenshot of a code quality tool's output for the file `ResourceAvailabilityService.java`. It lists five metrics, each preceded by a circular icon with the letter 'M'. The metrics are: Loop Nesting Depth: 0 (green circle), Maintainability Index: 62.0852 (yellow circle), McCabe Cyclomatic Complexity: 1 (green circle), Number Of Loops: 0 (grey circle), and Number Of Parameters: 7 (orange circle).

Metric	Value
Loop Nesting Depth	0
Maintainability Index	62.0852
M McCabe Cyclomatic Complexity	1
Number Of Loops	0
Number Of Parameters	7

6. By running class metrics in `resource-manager-microservice/src/main/java/nl.tudelft.sem.resource.manager/domain/services/ResourceAvailabilityService.java` we found that the main constructor had a number of parameters that was above what we consider acceptable. Therefore, we have decided this method should be refactored.

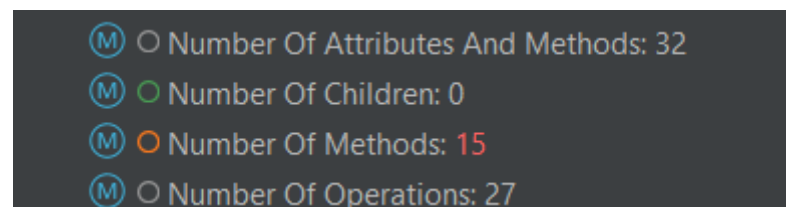


A screenshot of a code quality tool's output for the file `ClusterNode.java`. It lists five metrics, each preceded by a circular icon with the letter 'M'. The metrics are: Lines Of Code: 18 (grey circle), Loop Nesting Depth: 0 (green circle), Maintainability Index: 60.3229 (yellow circle), McCabe Cyclomatic Complexity: 1 (green circle), Number Of Loops: 0 (grey circle), and Number Of Parameters: 5 (orange circle).

Metric	Value
Lines Of Code	18
Loop Nesting Depth	0
Maintainability Index	60.3229
M McCabe Cyclomatic Complexity	1
Number Of Loops	0
Number Of Parameters	5

Class Level

1. By running the class metrics for `resource-manager-microservice/src/main/java/nl/tudelft/sem/resource/manager/domain/node/ClusterNode.java` we found that the total number of methods, including those generated using lombok, totalled to 15. As this goes above our accepted range, we have decided it is due for refactor.

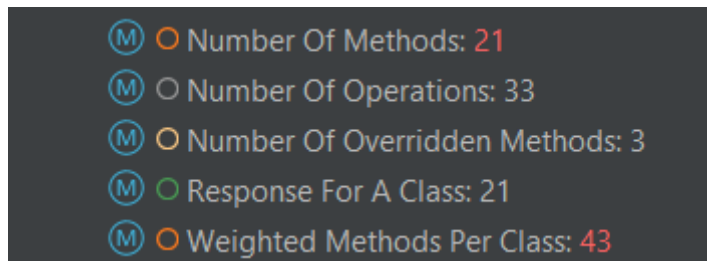


A screenshot of a code quality tool's output for the class `ClusterNode`. It lists four metrics, each preceded by a circular icon with the letter 'M'. The metrics are: Number Of Attributes And Methods: 32 (blue circle), Number Of Children: 0 (green circle), Number Of Methods: 15 (orange circle), and Number Of Operations: 27 (blue circle).

Metric	Value
Number Of Attributes And Methods	32
Number Of Children	0
Number Of Methods	15
Number Of Operations	27

2. By running class level metrics for `commons/src/main/java/sem/commons/RequestDTO.java` we can detect that the values for the number of methods, both weighted and

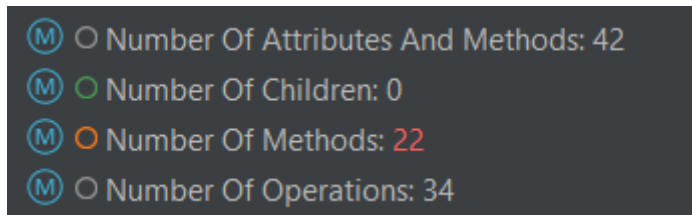
unweighted, go above our accepted range. For this precise case, if we inspect the source class, we can see that the overwhelming number of methods (every method besides the constructor) here are getters and setters due to the large amount of parameters. Below you can see that both metrics here in question have values above our accepted range.



A screenshot of a terminal window showing class metrics for `Request.java`. The metrics are listed with a blue circle containing an 'M' icon, a green circle, and the metric name followed by its value. The values are: Number Of Methods: 21, Number Of Operations: 33, Number Of Overridden Methods: 3, Response For A Class: 21, and Weighted Methods Per Class: 43.

Metric	Value
Number Of Methods	21
Number Of Operations	33
Number Of Overridden Methods	3
Response For A Class	21
Weighted Methods Per Class	43

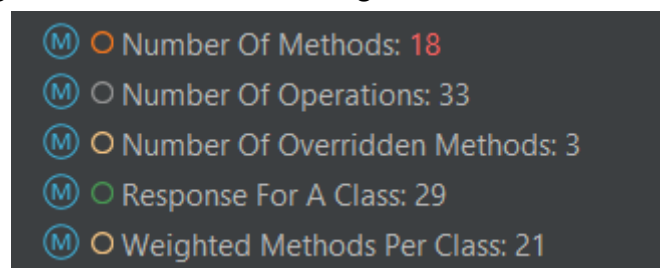
- By running the class metrics for `faculty-microservice/src/main/java/sem/faculty/domain/Request.java` we found that this class went above our acceptable range for the *Number of Methods* metric. Thus, we decided it needs refactoring.



A screenshot of a terminal window showing class metrics for `Request.java`. The metrics are listed with a blue circle containing an 'M' icon, a green circle, and the metric name followed by its value. The values are: Number Of Attributes And Methods: 42, Number Of Children: 0, Number Of Methods: 22, and Number Of Operations: 34.

Metric	Value
Number Of Attributes And Methods	42
Number Of Children	0
Number Of Methods	22
Number Of Operations	34

- By running the class metrics for `user-microservice/src/main/java/nl/tudelft/sem/template/user/domain/user/AppUser.java`, we found that the Number Of Methods exceeded our accepted ranges. Therefore, we thought it was due for refactor

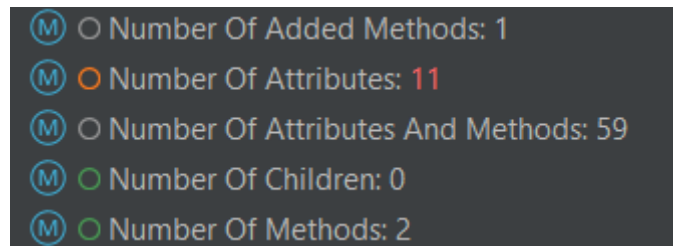


A screenshot of a terminal window showing class metrics for `AppUser.java`. The metrics are listed with a blue circle containing an 'M' icon, a green circle, and the metric name followed by its value. The values are: Number Of Methods: 18, Number Of Operations: 33, Number Of Overridden Methods: 3, Response For A Class: 29, and Weighted Methods Per Class: 21.

Metric	Value
Number Of Methods	18
Number Of Operations	33
Number Of Overridden Methods	3
Response For A Class	29
Weighted Methods Per Class	21

- By running the class metrics for `gateway/src/main/java/nl/tudelft/sem/template/gateway/authentication/J`

wtRequestFilter.java, we found that the number of attributes is above our maximum admitted limit.



6. By running the class metrics for *gateway/src/main/java/nl/tudelft/sem/template/gateway/config/ReplyingTemplateConfiguration.java*, we found two metrics that exceeded the limits set by us. Firstly, the Number of Methods is equal to 21, which goes above our acceptable range.

○ NOM	Li-Henry Metrics Set	Number Of Methods	21
○ WMC	Chidamber-Kemerer Metrics Set	Weighted Methods Per Class	21
○ LCOM	Chidamber-Kemerer Metrics Set	Lack Of Cohesion Of Methods	21
○ CBO	Chidamber-Kemerer Metrics Set	Coupling Between Objects	17

Exercise 2

Method Level

1. The `acceptRequests` method in the class `FacultyHandlerService` has been refactored to have a McCabe Cyclomatic Complexity of 2 instead of 7. In order to achieve this, certain decision points were removed from the original method, and added to newly created ones with the purpose of acting as helper methods. The following screenshots show the new source method and the two methods with extracted decision points. This was previously all one method.

```

public StatusDTO addNewUser(UserDTO userDTO) {
    StatusDTO result = checkStatusDTOValidity(userDTO);
    if (result != null) {
        return result;
    }

    NetId netId = new NetId(userDTO.getNetId());
    Password password = new Password(userDTO.getPassword());
    Role role = Role.valueOf(userDTO.getRole());

    List<String> facultiesStrings = userDTO.getFaculties();
    List<nl.tudelft.sem.template.user.domain.user.FacultyName> faculties = new ArrayList<>();

    result = fillFaculties(facultiesStrings, faculties);
    if (result != null) {
        return result;
    }

    try {
        registrationService.registerUser(netId, password, role, faculties);
    } catch (NetIdAlreadyInUseException e) {
        return new StatusDTO("User with netID: " + e.getMessage() + " already exists");
    }

    return new StatusDTO("OK");
}

```

(M) ○ Maintainability Index: 49,8381
 (M) ○ McCabe Cyclomatic Complexity: 4
 (M) ○ Number Of Loops: 0
 (M) ○ Number Of Parameters: 1

```

public StatusDTO checkStatusDTOValidity(UserDTO userDTO) {
    if (userDTO.getNetId().equals("")) {
        return new StatusDTO("NetId cannot be empty!");
    }
    if (userDTO.getPassword().equals("")) {
        return new StatusDTO("Password cannot be empty!");
    }
    try {
        Role unused = Role.valueOf(userDTO.getRole());
    } catch (IllegalArgumentException e) {
        return new StatusDTO("Provided role does not exist!");
    }
    return null;
}

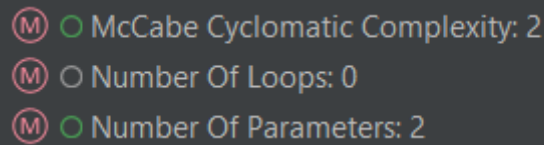
public StatusDTO fillFaculties(List<String> facultiesStrings, List faculties) {
    for (String name : facultiesStrings) {
        try {
            faculties.add(FacultyName.valueOf(name));
        } catch (IllegalArgumentException e) {
            return new StatusDTO("Wrong faculty name");
        }
    }
    return null;
}

```

Unfortunately there still had to be an if-statement to check if the other

methods found any problems. But these two if-statements replace 5 decision points, reducing the complexity in total.

2. The MainUserController method `addNewUser` has been refactored to have a McCabe Cyclomatic Complexity of 2 instead of 7.



Ⓜ ○ McCabe Cyclomatic Complexity: 2
Ⓜ ○ Number Of Loops: 0
Ⓜ ○ Number Of Parameters: 2

To achieve this, a few decision points, like *try* and *if* statements, were moved to new methods with a specific functionality. The following screenshots show the new source method and the two methods created from it with extracted decision points. These three methods are all a single method prior to this.

```
public StatusDTO acceptRequests(String facultyName, List<Long> acceptedRequests) {  
    FacultyName facName;  
    try {  
        facName = FacultyName.valueOf(facultyName);  
    } catch (IllegalArgumentException e) {  
        return new StatusDTO("Wrong faculty name");  
    }  
    List<Long> badRequests = acceptRequestsHelper(facName, acceptedRequests);  
    return acceptRequestsOutput(badRequests);  
}
```

```
public List<Long> acceptRequestsHelper(FacultyName facName, List<Long> acceptedRequests){  
    List<Long> badRequests = new ArrayList<>();  
    for (Long id : acceptedRequests) {  
        Request request = requestRepository.findById(id);  
        if (request == null) {  
            badRequests.add(id);  
            continue;  
        }  
        facultyHandler.handleAcceptedRequests(facName, request);  
    }  
    return badRequests;  
}  
  
public StatusDTO acceptRequestsOutput(List<Long> badRequests){  
    if (badRequests.isEmpty()) {  
        return new StatusDTO("OK");  
    }  
    return new StatusDTO("Could not find the following requests: " + badRequests.toString());  
}
```

Meanwhile the two new methods created having the following cyclomatic complexities respectfully:

(M) ○ McCabe Cyclomatic Complexity: 3	(M) ○ McCabe Cyclomatic Complexity: 2
(M) ○ Number Of Loops: 1	(M) ○ Number Of Loops: 0
(M) ○ Number Of Parameters: 2	(M) ○ Number Of Parameters: 1

3. The increased cyclomatic complexity of the *getDateForRequest* method was caused by checking if there were enough free resources to fit the request:

```

LocalDate getDateForRequest(
    Resource resources,
    LocalDate date,
    Reserver facultyName) {
    LocalDate today = currentDateProvider.getCurrentDate();

    while (date.isAfter(today)) {
        Resource freeResources = resourceAvailabilityService.freeResourcesByDateAndReserver(date, facultyName);

        if (freeResources.getCpuResources() >= resources.getCpuResources() &&
            freeResources.getGpuResources() >= resources.getGpuResources() &&
            freeResources.getMemResources() >= resources.getMemResources()) {

            return date;
        }
        date = date.minusDays( daysToSubtract: 1);
    }

    return null;
}

```

This was remedied by moving this comparison into a separate, private method in the same class:

```

LocalDate getDateForRequest(
    Resource resources,
    LocalDate date,
    Reserver facultyName) {
    LocalDate today = currentDateProvider.getCurrentDate();

    while (date.isAfter(today)) {
        Resource freeResources = resourceAvailabilityService.freeResourcesByDateAndReserver(date, facultyName);

        if (enoughResourcesAvailable(freeResources, resources)) {

            return date;
        }
        date = date.minusDays( daysToSubtract: 1);
    }

    return null;
}

1 usage  👤 Olgreanu Codrin
private boolean enoughResourcesAvailable(Resource freeResources, Resource neededResources) {
    return freeResources.getCpuResources() >= neededResources.getCpuResources() &&
        freeResources.getGpuResources() >= neededResources.getGpuResources() &&
        freeResources.getMemResources() >= neededResources.getMemResources();
}

```

This results in a cyclomatic complexity of just 3 for both methods:

(M) ○ Maintainability Index: 56.4979	(M) ○ Maintainability Index: 72.3103
(M) ○ McCabe Cyclomatic Complexity: 3	(M) ○ McCabe Cyclomatic Complexity: 3
(M) ○ Number Of Loops: 1	(M) ○ Number Of Loops: 0

Since all that was done was to extract existing functionality into a separate *private* method, there was no need to add existing tests. After

adding documentation to the new method, and ensuring that tests pass and formatting is up to standard, this refactoring is concluded.

4. In order to refactor the *checkResourceValidity* method above mentioned we will have to extract some functionality from this method into two new methods. In its current state, *checkResourceValidity* does 2 checks on the amount of resources, and each check consists of checking 3 subsequent constraints, which is the reason for the value of 6 for the *McCabe Cyclomatic Complexity*. This is the current state of *checkResourceValidity*.

```
/**
 * Checks the constraint for the resource class.
 * Called in the constructor before making the object.
 * @param cpu - cpu of resource
 * @param gpu - gpu of resource
 * @param memory - memory of resource
 * @throws NotValidResourcesException - when either negative values are found or other constraints are broken
 */
2 usages  ↳ Jasper van Beusekom
public void checkResourceValidity(int cpu, int gpu, int memory) throws NotValidResourcesException {
    if (cpu < 0 || gpu < 0 || memory < 0) {
        throw new NotValidResourcesException("Resource cannot have negative values");
    }

    //The business logic for issue #23
    if (cpu < memory || cpu < gpu) {
        throw new NotValidResourcesException("The cpu resources should be equal to at least max(memory, gpu)");
    }
}
```

What we will do here is to transfer each of the subsequent checks into new methods, called *checkResourcesNonNegative* and *checkResourcesRelationalConstraints*. Here you can see the state after refactoring (not commented yet in order to have a smaller image)

```

public void checkResourceValidity(int cpu, int gpu, int memory) throws NotValidResourcesException {
    if (checkResourcesNonNegative(cpu, gpu, memory)) {
        throw new NotValidResourcesException("Resource cannot have negative values");
    }

    //The business logic for issue #20
    if (checkResourcesRelationalConstraint(cpu, gpu, memory)) {
        throw new NotValidResourcesException("The cpu resources should be equal to at least max(memory, gpu)");
    }
}

1 usage new *
public boolean checkResourcesNonNegative(int cpu, int gpu, int memory) {
    return (cpu < 0 || gpu < 0 || memory < 0);
}

1 usage new *
public boolean checkResourcesRelationalConstraint(int cpu, int gpu, int memory) {
    return (cpu < memory || cpu < gpu);
}

```

This way we can reduce the value of *McCabe Cyclomatic Complexity* for the method in question here, `checkResourceValidity`, to 3, which is under our thresholds (which are motivated above).

```

(M) ○ Halstead Volume: 83.7618
(M) ○ Lines Of Code: 17
(M) ○ Loop Nesting Depth: 0
(M) ○ Maintainability Index: 59.5739
(M) ○ McCabe Cyclomatic Complexity: 3

```

These are the main refactoring changes, as it fixes the previously found code smell without creating new ones in the process (the new methods have all metrics under our thresholds). After checking the test suite for this method, we found that it has extensive unit tests for all combinations in the flow of the creation of the *Resource* object, as these methods are called through the constructor. Therefore, in the new methods, which simply relocate functionality, there isn't the need to add more tests. After such refactoring is done, documentation is adjusted, and formatting is according to our standards, the refactoring is concluded.

5. The major issue in this constructor method is the number of parameters, which can be seen in the following image.

```

/**
 * Constructor method.
 */
Razvan Nistor +1
public Request(String name, String netId, String description,
               LocalDate preferredDate, RequestStatus status, FacultyName facultyName, Resource resource) {
    this.name = name;
    this.netId = netId;
    this.description = description;
    this.preferredDate = preferredDate;
    this.status = status;
    this.facultyName = facultyName;
    this.resource = resource;
}

```

This creates problems with the readability, maintainability and usability of the class. It is hard to create a new object of this class, because it is hard to remember the correct order of all these parameters. In order to refactor this method, we have introduced a new Parameter Object class called RequestDetails that holds the name, description, date and status of the new request.

```

@Getter
public final class RequestDetails {
    1 usage
    private final String name;
    1 usage
    private final String description;
    1 usage
    private final LocalDate preferredDate;
    1 usage
    private final RequestStatus status;

    /**
     * Constructor method.
     */
    Razvan Nistor
    public RequestDetails(String name, String description, LocalDate preferredDate, RequestStatus status) {
        this.name = name;
        this.description = description;
        this.preferredDate = preferredDate;
        this.status = status;
    }
}

```

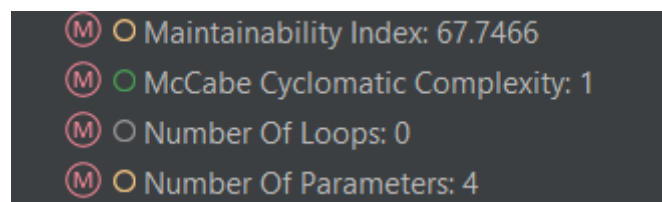
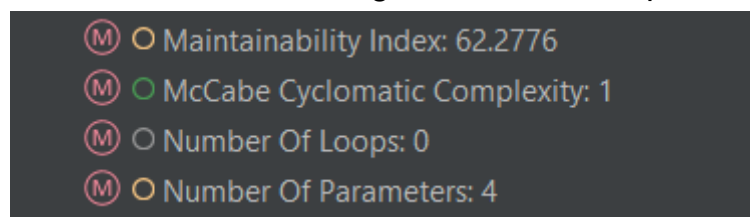
This has reduced the number of parameters from 7 to 4 (which is the threshold we set, as described above). This is the new constructor which makes use of this new Parameter object class, which is much easier to use and understand.

```

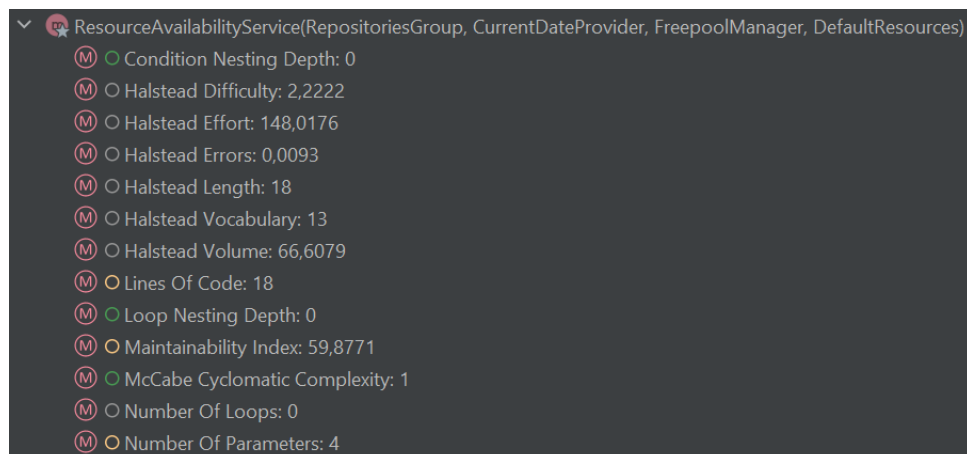
/**
 * Constructor method.
 */
public Request(RequestDetails requestDetails, String netId, FacultyName facultyName, Resource resource) {
    this.name = requestDetails.getName();
    this.description = requestDetails.getDescription();
    this.preferredDate = requestDetails.getPreferredDate();
    this.status = requestDetails.getStatus();
    this.netId = netId;
    this.facultyName = facultyName;
    this.resource = resource;
}

```

Below, there are 2 images of the *MetricsTree* tool run on the constructors from both classes. The first image is the constructor of the Request class, and the second image is from the RequestDetails class.



- To lower the number of parameters of the constructor of the *ResourceAvailabilityService* class we have introduced a new class *RepositoriesGroup* which groups all repositories used by *Resource Manager* microservice. Thanks to this refactoring, we have brought down the *Number of Parameters* metric to fulfill our threshold of 4.



Class Level

1. The issue with the *ClusterNode* class was caused by careless overuse of *Lombok* annotations. This caused numerous methods to be implicitly generated. Upon closer inspection, it was apparent that only the *@Getter* and *@NoArgsConstructor* annotations were needed. *@ToString* was not used, and out of the *setters* there was only one use case in the project:

```
@Test
void test_equals() {
    node2.setToken(new Token( tokenValue: "token1"));
    assertThat(node1).isEqualTo(node2);
}
```

As this test could be easily rewritten not to use the method *setToken*, the *@Setter* annotation could also be scrapped:

```
@Test
void test_equals() {
    ClusterNode node3 = new ClusterNode(
        new OwnerName("name3"),
        new URL( urlValue: "url3"),
        new Token( tokenValue: "token1"),
        Resource.with( resources: 300)
    );

    assertThat(node1).isEqualTo(node3);
}
```

After this refactoring, the number of methods and WMC metrics dropped within our accepted range:

```
(M) ○ Number Of Methods: 9
(M) ○ Number Of Operations: 21
(M) ○ Number Of Overridden Methods: 2
(M) ○ Response For A Class: 11
(M) ○ Weighted Methods Per Class: 11
```

Since dependencies on the removed methods were either nonexistent, or changed accordingly, this refactoring required no further testing or documentation to be added.

2. The problem we have to solve with this refactoring can be traced back from the exorbitant amount of methods to the excessive number of parameters (lombok annotations add the *getters*, *setters*, and more to the class). Below you can see what the problem is:

```
@Data
@AllArgsConstructor
@NoArgsConstructor
public class RequestDTO {

    no usages
    private long requestId;
    1 usage
    private String name;
    1 usage
    private String netId;
    1 usage
    private FacultyName faculty;
    1 usage
    private String description;
    1 usage
    private LocalDate preferredDate;
    1 usage
    private Resource resource;
```

As can be seen, each parameter will generate 2 methods, leading to 12 just for *getters* and *setters*. The rest pertain only to the class, but still amount to 21. Our refactoring strategy here will consist in extracting some parameters into their own “logical units”. The final structure will be the following:

```
public class RequestDTO {

    no usages
    private long requestId;
    1 usage
    private RequestFacultyInformation requestFacultyInformation;
    1 usage
    private RequestResourceManagerInformation requestResourceManagerInformation;
```

```

public class RequestResourceManagerInformation {

    1 usage
    private String name;
    1 usage
    private String description;
    1 usage
    private Resource resource;

```

```

public class RequestFacultyInformation {

    1 usage
    private LocalDate preferredDate;
    1 usage
    private FacultyName faculty;
    1 usage
    private String netId;

```

With this new aggregation of related data, we not only make the class and the data it's carrying clearer, we also make it more maintainable! Below you can see the metrics after the refactoring for the main class and the ones created in the refactoring:

RequestDTO

○ NOOM	Lorenz-K...	Number Of Overridden Methods	3
○ NOAM	Lorenz-K...	Number Of Added Methods	7
○ SIZE2	Li-Henry...	Number Of Attributes And Methods	28
○ NOM	Li-Henry...	Number Of Methods	13

RequestResourceManagerInformation

○ NOOM	Lorenz-K...	Number Of Overridden Methods	3
○ NOAM	Lorenz-K...	Number Of Added Methods	7
○ SIZE2	Li-Henry...	Number Of Attributes And Methods	27
○ NOM	Li-Henry...	Number Of Methods	12

RequestFacultyInformation

○ NOOM	Lorenz-K...	Number Of Overridden Methods	3
○ NOAM	Lorenz-K...	Number Of Added Methods	7
○ SIZE2	Li-Henry...	Number Of Attributes And Methods	27
○ NOM	Li-Henry...	Number Of Methods	12

All metrics are indeed contained in our acceptable ranges. With this refactoring we also had to adjust the getters and setters of *requestDTO*

in order to make it compatible with the code that was using it in its previous form. We have also adjusted tests and documentations to cover this new code. With that, this refactoring is concluded.

3. The problem here closely resembled that of the requestDTO before, so the solution was quite similar. The many attribute fields were replaced by four in total: requestId, Status, RequestResourceManagerInformation and RequestFacultyInformation. For information about the last two attributes, see the previous point.

As seen below the number of methods is now below the 14 we decided should be the maximum per class.

	Metric	Metrics Set	Description ▲	Value	Regular Ra...
✓	NOC	Endowment...	Number Of Children	0	[0..2)
○	NOM	Li-Henry M...	Number Of Methods	12	[0..7)
○	NOO	Lorenz-Kid...	Number Of Operations	25	
○	NOOM	Lorenz-Kid...	Number Of Overridden Methods	3	[0..3)

The class looks a lot cleaner after the refactor:

```
@Entity
@Table(name = "requests")
@NoArgsConstructor
@Getter
public class Request {

    @Id
    @Setter
    @GeneratedValue(strategy = GenerationType.SEQUENCE)
    @Column(name = "requestId", nullable = false)
    private long requestId;

    @Setter
    @Enumerated(EnumType.STRING)
    @Column(name = "status", nullable = false)
    private RequestStatus status;

    @Embedded
    protected RequestFacultyInformation requestFacultyInformation;

    @Embedded
    protected RequestResourceManagerInformation requestResourceManagerInformation;
```

The decision to include a setter for requestId was because it is not

added in the constructor, since it is generated by the annotation. The Status attribute has a setter because the Status of a request can change during its lifetime.

4. Upon closer inspection, the issue with the AppUser class was caused by the careless overuse of different Lombok annotations, causing multiple methods to be generated. It soon became apparent that only the @Getter and @NoArgsConstructor annotations were actually needed. All method setters, except setId(), were actually not being used. So, it would make sense to remove the extra lombok annotation, and this method separately. Then, tests using the other setters could be removed. An instance of such a test can be seen below:

```
@Test
void setRole() {
    assertThat(user.getRole()).isEqualTo(testRole);
    Role newRole = Role.FACULTY_REVIEWER;
    user.setRole(newRole);
    assertThat(user.getRole()).isEqualTo(newRole);
}
```

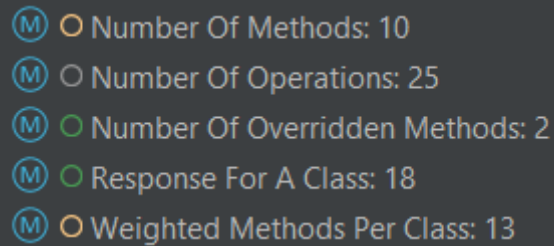
Another problem caught contributing to the excess methods issue was the ToString method. This had been divided into two different methods:

```
public String facultyString() {
    String faculties = this.faculty.toString();
    faculties = faculties.replace( target: "[", replacement: "");
    faculties = faculties.replace( target: "]", replacement: "");
    return faculties;
}

@Override
public String toString() {
    return "AppUser{" +
        "NetId = " + netId.toString() +
        ", password = " + password.toString() +
        ", role = " + role.name() +
        ", faculties = " + facultyString() +
        "}";
}
```

This could very well be reduced to one toString method. In fact there were a few other methods that were also not used. These are changePassword(), addFaculty(), and hashCode(). Therefore, these

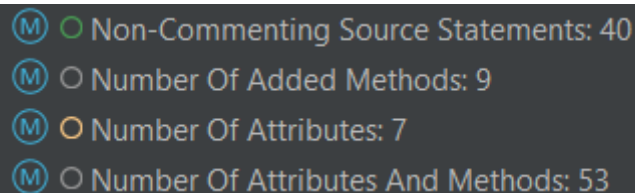
methods were removed as well. After performing all the refactoring mentioned above, the following statistics were obtained:



(M) ○ Number Of Methods: 10
(M) ○ Number Of Operations: 25
(M) ○ Number Of Overridden Methods: 2
(M) ○ Response For A Class: 18
(M) ○ Weighted Methods Per Class: 13

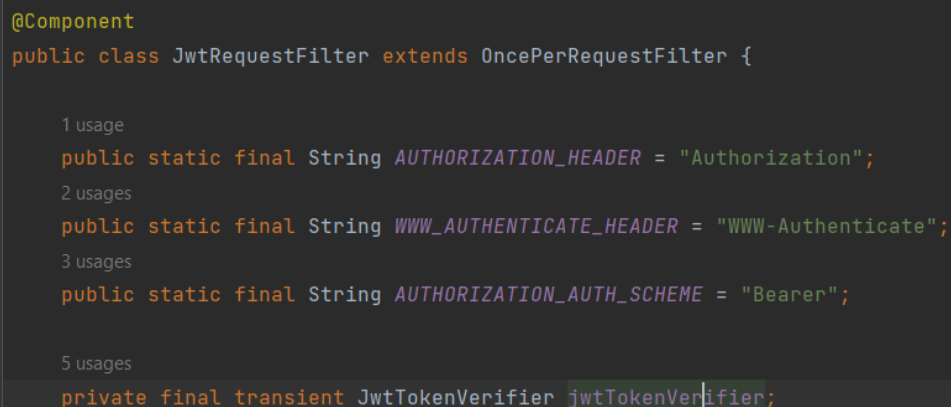
This changed the number of methods in the class to our accepted range, while also changing the number of overridden methods to the recommended range, and with that the refactoring is finished.

5. The problem with this class is the extensive number of attributes. But, because the class extends the `OncePerRequestFilter.java` class, which already has 7 attributes, as seen below, we can not reduce the number of attributes of our `JwtRequestFilter` class below this number.



(M) ○ Non-Commenting Source Statements: 40
(M) ○ Number Of Added Methods: 9
(M) ○ Number Of Attributes: 7
(M) ○ Number Of Attributes And Methods: 53

However, by looking through the attributes in the `JwtRequestFilter` class, we can see that 3 of the attributes are just static Strings, which are used for readability purposes to encapsulate hard-coded strings that are used throughout the code.



```
@Component
public class JwtRequestFilter extends OncePerRequestFilter {

    1 usage
    public static final String AUTHORIZATION_HEADER = "Authorization";
    2 usages
    public static final String WWW_AUTHENTICATE_HEADER = "WWW-Authenticate";
    3 usages
    public static final String AUTHORIZATION_AUTH_SCHEME = "Bearer";

    5 usages
    private final transient JwtTokenVerifier jwtTokenVerifier;
```

So, we have refactored this class by extracting these attributes to another class, that will not have any other functionality other than keeping these strings.

```

0 usages  Razvan Nistor
public final class AuthorizationInformation {
    1 usage
    public static final String AUTHORIZATION_HEADER = "Authorization";
    2 usages
    public static final String WWW_AUTHENTICATE_HEADER = "WWW-Authenticate";
    3 usages
    public static final String AUTHORIZATION_AUTH_SCHEME = "Bearer";
}

```

For the original class, there is no need to create another attribute of this new class, because the strings can be accessed by just calling, for example, “AuthorizationInformation.AUTHORIZATION_HEADER”, as it can be seen below.

```

// Get authorization header
String authorizationHeader = request.getHeader(AuthorizationInformation.AUTHORIZATION_HEADER);

```

The attributes and constructor of the refactored JwtRequestFilter class can be seen in the following image, alongside the new metrics for this class.

```

@Component
public class JwtRequestFilter extends OncePerRequestFilter {
    5 usages
    private final transient JwtTokenVerifier jwtTokenVerifier;

    1 usage  Hubert Nowak
    @Autowired
    public JwtRequestFilter(JwtTokenVerifier jwtTokenVerifier) { this.jwtTokenVerifier = jwtTokenVerifier; }
}

```

```

(M) ○ Number Of Added Methods: 1
(M) ○ Number Of Attributes: 8
(M) ○ Number Of Attributes And Methods: 56
(M) ○ Number Of Children: 0
(M) ○ Number Of Methods: 2

```

6. The *ReplyingTemplateConfiguration* class holds all methods used to create all *ReplyingKafkaTemplates* used in communication between gateway and other microservices. As each of the topics used for sending messages introduced 2 methods in this class, the number of methods grew quickly. At the same time, we have separate controllers responsible for communication with different microservices, meaning that many objects were dependent on the methods provided by the *ReplyingTemplateConfiguration*. After quick inspection, we have identified that around $\frac{1}{3}$ of these methods were used in communication

with the faculty microservice and have decided to move them into a new class *ReplyingTemplateFacultyConfiguration*. Such a separation reduces the number of dependencies between objects and means that the classes are easier to maintain.

Below you can see the metrics after refactoring.

ReplyingTemplateConfiguration

○ NOM	Li-Henry Metrics Set	Number Of Methods	13
○ CBO	Chidamber-Kemerer M...	Coupling Between Objects	12

ReplyingTemplateFacultyConfiguration

○ NOM	Li-Henry Metrics Set	Number Of Methods	8
○ CBO	Chidamber-Kemerer M...	Coupling Between Objects	7