ASSIGNMENT 2

SEM GROUP 01a

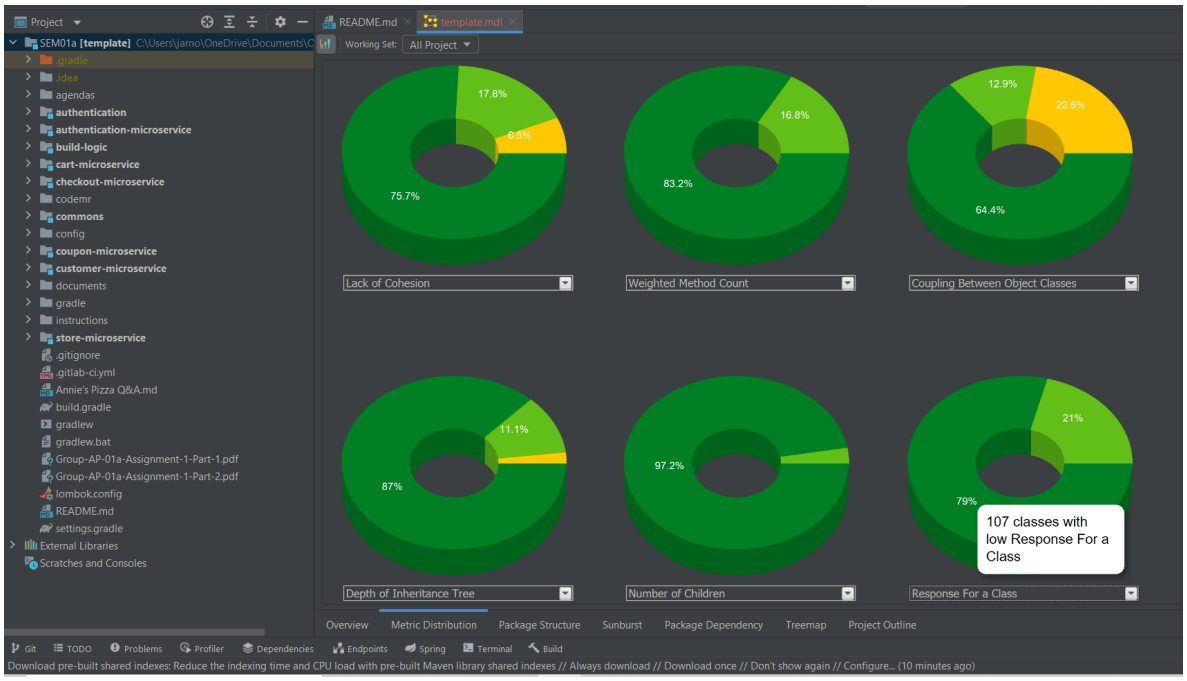
Minimum set of metrics

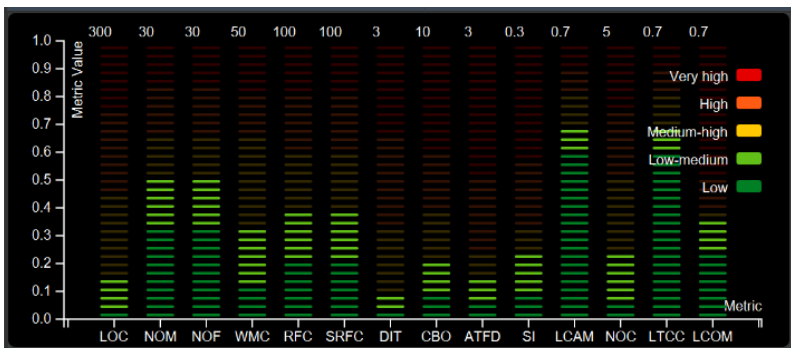
A suite of six software metrics for Object-oriented programs:

- **WMC** = Weighted Methods (Complexity) 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

As part of our task to refactor classes and methods that require improvement, we used *CodeMR,* an architectural software quality and static code analysis tool. After analysis, we ended up with the following data regarding our current code metrics, prior to any refactoring.

BEFORE REFACTORING:





Upon reviewing the results of the code analysis using the CodeMR tool, we observed good scores across all six software metrics. This made it challenging to pinpoint which methods and classes needed refactoring, meaning we had to be nitpicky with our decisions. As a result, we decided to focus on the LOC (Lines of Code) and CBO (Coupling Between Objects) metrics for our method refactoring as these seemed to have the most potential for possible improvement within our program. We also checked for Parameter Count on methods where it was relevant.
Based on these metrics, we isolated the parts of our code that were most in need of refactoring. We ended up with the following list of 5 methods:

METHODS:

1. **`selectCoupon (CouponController)`**
    - Metrics Before: High LOC (34), Medium-high CBO (10)
    - Metrics After: Low LOC (17), Low-medium CBO (6)
    - Changes:
      Instead of happening within the endpoint, the iteration over the different eligible coupon codes (ones that have not been used by the user in the past) has been extracted to a new method buildPriorityQueue(). From there, another two methods are being called, checkCouponValid(), to assure that the code exists and can be used, and applyCoupon() that gets the final price given to the order according to the coupon's type.
2. **`addCoupon (CouponController)`**
    - Metrics Before: High LOC (20), Medium-high CBO (14)
    - Metrics After: Low LOC (10), Medium-high CBO (8)
    - Changes:
      Various checks have been extracted to different methods. Specifically, all the checks regarding input fields being incomplete are gathered in one method, where the handling of any related exceptions take place. Additionally, any invalidly formatted input checks have been extracted to a second method, where they are handled respectively.
3. **`removeOrderById (OrderController)`**
    - Metrics Before: High LOC (26), Medium-high CBO (8)
    - Metrics After: Low LOC (8), Low-Medium CBO (4)
    - Changes:
      Extracted most of the code of this method into three separate methods, namely tryRemoveOrderById() which in turn calls an additional two new methods called isOrderRemovable() and removeOrderAndCoupon(). This spreads out the functionality of removeOrderById() in order to lower LOC and CBO.
4. **`addOrder (OrderController)`**
    - Metrics Before: High LOC (32), Medium-High CBO (10)
    - Metrics After: Low LOC (16), Low-Medium CBO (5)
    - Changes:
      Split the functionality of the method into two different methods, in order to avoid having a single, unnecessarily lengthy method.
5. **`doRequestWithResponse (RequestHelper)`**
    - Metrics Before: LOC (17), Medium-high CBO (9), Parameter count: 5
    - Metrics After : LOC(10), CBO(7), Parameter count : 3
    - Changes:

Refactored the request type, port and path to a different class and removed the methods of getRequest(), postRequest(), deleteRequest(). I refactored this because we could just use the refactored method doRequest() that works.

In terms of refactoring Classes, we analyzed all six of the most significant software metrics for each class in our program and, again, similarly to the Methods, struggled to decide on which classes needed refactoring due to their respective metrics already being at a satisfactory level. Taking this into consideration, we chose the ones that had the highest metrics regardless and ended up isolating the following 5 Classes for refactoring:

CLASSES:

1. **`OrderController`**
    - Metrics Before: WMC (34), CBO (15), RFC (98), LCOM (0.333), DIT (1) NOC (0)
    - Metrics After: WMC (22), CBO (14), RFC (57), LCOM (0.519), DIT (1) NOC (0)
    - Changes:
      The metrics of this class were very high, especially the RFC. We lowered the values of the metrics by using move method refactoring on the following methods: tryRemove, getAllOrders, getOrderById, getOrderPrice.
2. **`CouponController`**
    - Metrics Before: WMC (30), CBO (18), RFC (82), LCOM (0.533), DIT (1) NOC (0)
    - Metrics After: WMC (15), CBO (13), RFC (43), LCOM (0.583), DIT (1) NOC (0)
    - Changes:
      The coupon controller was a complex class mainly due to the relatively large amount of endpoints it contained, and the size and complexity of specifically two of them: the selectCoupon() and addCoupon() methods. After refactoring those methods as explained earlier, the class had been improved, but was still complex. That was caused by the fact that it now had a variety of helper methods to those two endpoints. Therefore, to decrease the values of the listed metrics even more,it was decided to create a brand new class called CouponControllerService, that could contain all those services for the endpoints in CouponController. To implement the refactoring, moving methods was required, while also accounting for the injection of a couponControllerService instance in the constructor of the controller class.
3. **`CartController`**
    - Metrics Before: WMC (27), CBO (17), RFC (95), LCOM (0.848), DIT (1) NOC (0)
    - Metrics After: WMC (14), CBO (13), RFC (43), LCOM (0.5), DIT (1) NOC (0)
    - Changes:
      The cart controller had a high coupling and low cohesion between methods. The idea was to group the methods that work on similar attributes into a service class. Helper methods like requireNotEmpty(), getCustomPizzza(), getDefaultPizza(), getCartFromSessionId(), assertInCart() I moved to the CartService class because they are mostly helper methods and do not have real business logic.
      Because we moved the methods to the service class the dependencies also changed because some dependencies could be moved to the service class (RequestHelper and DefaultPizzaRepository). This change decreased the coupling (CBO).

4. **`CustomerController`**
    - Metrics Before: WMC (11), CBO (5), RFC (37), LCOM (0.45), DIT (1) NOC (0)
    - Metrics After: WMC (11), CBO (4), RFC (23), LCOM (0.45), DIT (1) NOC (0)
    - Changes:
      A small change to this class' addCustomer method in which lines that were unnecessarily in the controller were moved to the corresponding service class. This brought down the CBO and, more significantly, the RFC stats.
5. **`AuthenticationController`**
    - Metrics Before: WMC (5), CBO (18), RFC (65), LCOM (1), DIT (1) NOC (0)
    - Metrics After: WMC (3), CBO (11), RFC (65), LCOM (1), DIT (1) NOC (0)
    - Changes:
      Refactoring the AuthenticationController was hard to refactor and as you can see in the changes of the metrics we were only able to slightly decrease the WMC and the CBO. There were a couple of things we could do to decrease the CBO of the class.
      First of all, there were two try-catch statements in the authenticate endpoint to catch exceptions in case the credentials were incorrect, or the user's account was disabled. What we did is remove these two try-catches and create a new class (AuthenticationExceptionHandler) in which we use the @ExceptionHandler annotation that Spring Boot provides. This allowed us to offload the logic to return a specific http response when these exceptions occur to another class, thus decreasing the CBO of the class.
      Also, the AuthenticationRequestModel used string's for the netId and the password. This means that in the authenticate method, we're manually constructing the actual NetId and Password objects. What we did is change the netId and password in AuthenticationRequestModel to the actual NetId and Password objects. This allows us to offload the construction of these objects to the spring boot model parser, thus decreasing the CBO since we no longer need to interact with the NetId and Password class inside of AuthenticationController.
      Finally we extracted the logic of the authenticate endpoint to its own service (AuthenticationService). This service will now handle actually checking the credentials of the user and generating a JWT token. This decreased the amount of fields of the controller by 2, and also decreases the CBO because instead of interacting with 3 fields (AuthenticationManager, JwtTokenGenerator & JwtUserDetailsService) it now only interacts with AuthenticationService inside of the authenticate endpoint.