12B Delivery Lab Report

Klara, Rūta, Paul, Bogdan, Michael, Ada

January 18, 2024

Contents

1   Final List of Requirements Implemented

1.1   Functional Requirements

For the delivery system of YumYumNow, the requirements regarding functionality and service are grouped under the Functional Requirements. Within these functional requirements, four categories have been established using the MoSCoW model for prioritising requirements:

1.1.1   Must Haves

- The User shall be able to see the status of the order: "pending"/"accepted"/"rejected"/"preparing"/"given to courier"/"on-transit"/"delivered".Orders Status MRDelivered Status MROn Transit MRPreparing MR

- The Vendor shall be able to accept/reject orders. Accept order MRReject Order MR

- The Vendor shall be able to update the status of the order ("preparing" and "given to courier").Preparing to Given to courier MR

- The Courier shall be able to see where they need to pick up the order from and where to deliver it.Pick up and Delivery location MR

- The Courier shall be able to select the order they want to fulfill from a list of available orders or shall be able to be assigned to an order.Assigned to next order MR

- The Admin shall be able to set a maximum delivery zone for vendors if they don't have their own drivers.Admin Set Radius MR

1.1.2   Should Haves

- The User shall be able to see an estimated time arrival.ETA MR

- The Courier shall be able to indicate when they picked up an order.Pick up MR

- The Vendor shall be able to give a time estimation for when an order will be ready.Preparation Time MR

- The application shall manage situations such as delays for the delivery due to traffic, unexpected events, or issues with the delivery, customer service.Exceptions MR

1.1.3 Could Haves

- The User shall be able to rate the orders (stars out of 5).Rate Orders MR

- The Courier shall be able to update the status of the order. In-transit MR Delivered MR

- The Vendor shall be able to set use-own/use-other drivers.Boss update MR

- The Vendor shall be able to set a maximum delivery zone for their own restaurant if they have their own drivers.Vendor Radius MR

- The Admin shall be able to gather analytics about delivery times, driver efficiency, successful deliveries, and any issues encountered during deliveries and use this data for performance analysis and improvement.Ratings and Delivery Times MRCourier Efficiency MR

- The User shall be able to see the distance between them and the courier.Distance MR

- The Courier shall be able to constantly update the location of the order while on-transit. Updates MR

1.1.4 Would/Won't Haves

- Orders can be shared among users, option for driver to deliver multiple orders in one go, adding/modifying the order after it has been sent (by user).

- Courier can deliver multiple orders in one go.

- Order can be modified after it has been placed.

- The status of an order can be reverted to an earlier status.

- Vendors that do not have couriers see the closest requests first.

- Location of vendor can be changed after placing an order.

- Location of the customer, delivery destination can be changed after placing an order.

- Couriers can work for multiple vendors, or take orders that are not from the vendor they are working for.

- Couriers that do not work with a vendor can get an order from a vendor that has its couriers

2 Design Pattern Implementation

2.1 Strategy

2.1.1 Description

In this pattern we take a class that has multiple specific implementations of essentially similar functionality. We use this when handling the assignment of an order to a courier.

In the case where a courier works for a specific vendor, it should not have options of orders to choose from, it should be basically assigned an order. We imagine this as a "getNext" button that a

courier can press to be assigned an order, so we have a different endpoint to be called in this case. However if the courier is not working for a vendor, they should have access to a list of orders, which is the functionality of a different endpoint that we made. In both cases, the specific endpoints only give the available order options so in both cases they can call another endpoint for actually setting the respective courierId as themselves for an order. We imagined this as a page where the courier can first view the available order/orders and click an "accept" button, which makes another request, to actually set themselves as the courier of that order. Therefore we have two separate endpoints for getting the order options for two different strategies of finding available orders for a courier.

In this case our context is our OrderController that only communicates with the selected strategy through the NextOrderStrategy interface. We have two concrete strategies: GeneralOrdersStrategy - the one where the courier gets a list of options - and OrderPerVendorStrategy - where the courier only has one option from the vendor they work for. These files are located in "example-microservice/src/main/java/nl/tudelft/sem/template/example/domain/order/OrderStrategy". The commits related to the implementation can be accessed here.

2.1.2 The problem it solves

We currently have two different variants of an algorithm that handle assigning an order to a courier. Because the employer of a courier can change, we need a solution that can change the object's behavior during runtime. These variants are switched by a conditional statement that checks the employer of a courier, therefore we need this design pattern to easily switch between these behaviors and provide a scalable solution in which we can add more strategies in the future.

2.2 Chain of Responsibility

2.2.1 Description

The Chain of Responsibility design pattern establishes a chain of handlers, where each handler has the ability to either process a request or pass it to the next handler in line. This is useful in our application since some authentication for users needs to happen before changes in the system are made. In our case, controllers handle HTTP requests to our server and those requests need to get a response. When a request is received, information about the request (the ID of the user who sent it for example) is passed to the chain and each link needs to check for some conditions to hold. If some don't or an error on the server side occurs during the process, the appropriate response is sent back to the user. Because our project is small, there are only 2 links in the chain: authorization and validation. The first link will check if the user accessing the method is of the correct type (e.g. admin, vendor, courier). For example, a vendor should not have access to the methods that only the admin is meant to use. If the user is of a "wrong" type, the process does not progress to the second link. The second link will check the actual ID correspondence. For example, a vendor should only be allowed to change the order that belongs to them, and not an order of another vendor (even though they are both vendors). This is a continuation of the previous link in the sense that it does further validation. However, if the previous authorisation step decided that a user should not have access, this link will never be reached.

2.2.2 Problem solved with issue and commit

For example, if in the future the system needs to have protection against malicious users who try to brute force admin functionality by repeatedly sending unauthorized requests, a handler for this can be easily included in the chain and be developed and tested separately from the other handlers. In the
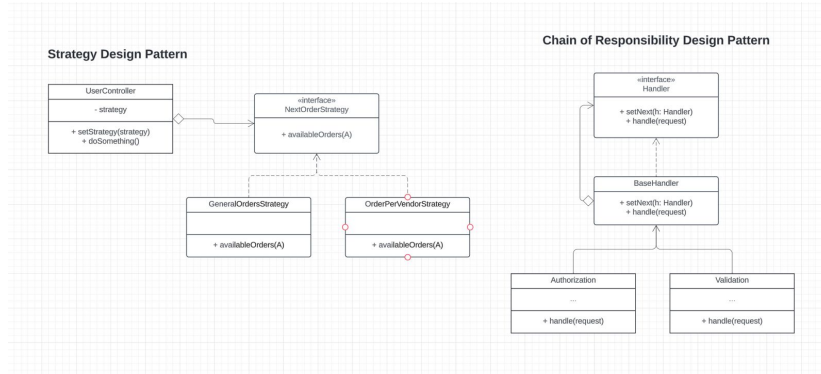
Figure 1: UML diagrams for design patterns implemented

FirstDesign commit the basic structure of the design was introduced. Also, the merge request which it was part of closed issue: Extend Authorization (implement Chain of Responsibility pattern).

3 Software Quality Report

3.1 Introduction to Metrics Used

We used the plug-in "MetricsTree" to evaluate the quality of our microservice. This tool lets us examine our code on the method, on the class, and on the project level. We first used the "Metric Profiles" section to get first impressions of our whole project. Here, the project is evaluated in 6 categories: "Brain Method", "Complex Method", "High Coupling", "Long Method", "Long Parameters List", and "Too Many Methods". After examining Figure 2, we have determined that we will focus on the 3 categories that are the most common in our project: "Long Method", "Long Parameter List", and "Too Many Methods" as these problems are common in more than 6 classes in our project.

This serves as a general overview as we determine specific classes and methods that we will aim to improve on these metrics. Since a section of our project is generated from OpenAPI Specification, during our evaluation, we focused on classes like controllers and services, on which our main contribution was concentrated. We went through metrics that were specific per class, as each class had a list of metrics with its metrics value color-coded into 3 levels: green (common value of metric), orange (casual value of metric), and red (uncommon value of metric) and lastly gray as neutral. The thresholds of these metrics are pre-determined from the plug-in.

Although most of our classes had mostly green or orange values, we observed some that had red values and in this section, we delve into these classes, why we think they need to be improved, metric levels on method level to determine methods to improve, and lastly the change after.
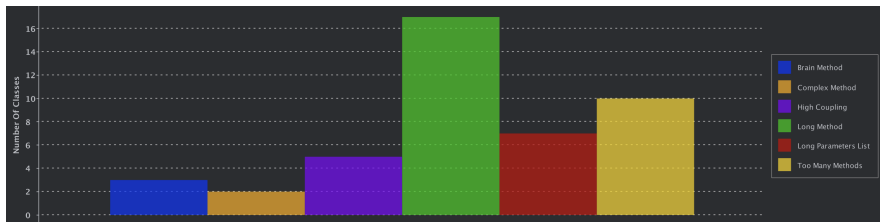

Figure 2: Metric Profiles of the whole project

3.1.1 Important Notice: What's wrong with Maintainability Index in MetricsTree

It is necessary to point out that the Maintainability Index calculated in MetricsTree gives reverse results. While in MetricsTree the lower the value the better, in reality, it's the other way round, the higher the better [1].

3.2 OrderController

This class was chosen to be improved because of its uncommon metric values in number of methods (above 14), number of overridden methods (greater than 4), and also weighted methods per class (greater than 34), all can be seen in Figure 3. These thresholds were pre-determined by the plug-in.

The problem of overridden methods is a frequent one in our project as we implement all of our controller methods by overriding the generates API methods. Since this does not reflect the actual Software quality of our project, we focus on other metrics in this class as well as the rest.

For the metric of number of methods, we determined to fix this problem by refactoring the class into 2 classes: OrderUserController and OrderManagementController. This way, OrderUserController will handle requests that will commonly be used by a user of the application, such as getting the delivery address of an order or giving a rating for an order. Whereas OrderManagementController will handle the endpoints commonly used by the other microservices and admin, which are related to general Order entity persistence, such as creating an order, updating an order, and getting all orders from the database.

This would result in 4 methods being in the OrderManagementController and the rest of 11 would be in the OrderUserController, which would put the metric of number of methods under the threshold of 14. However, this refactoring requires changing of the paths of our endpoints in order to be implemented in an organized fashion, therefore breaking our openAPI contract with other microservices. Since these order-related endpoints are essential to our integration with other microservices as the Order entity is shared, we have decided that this refactoring is not fit for this project. However, in a longer-term replication of this project, we acknowledge that this refactoring should be executed after an in-depth agreement with other microservices that would require for them to change their communication with this system, a process that is currently unfeasible in this short-term project.
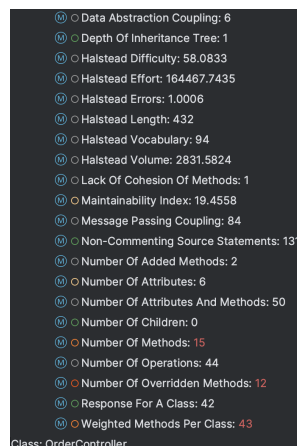

Figure 3: Class-level metric values for OrderController, red values being uncommon values

We therefore continue on evaluating this class on a method level.

3.2.1 getNextOrderForVendor

As can be seen from Figure 4, the reason this class was chosen was its unusual metric value of lines of code (above 30), as well as McCabe Cyclomatic Complexity of greater than 4, making it uncommon. To fix this problem, we refactored the end-logic of this method , which was a series of
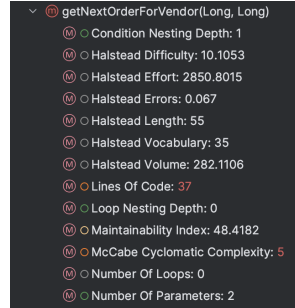

Figure 4: Method-level metrics for getNextOrderForVendor in OrderController before refactoring

if-cases mapping the return value of the service method into the right ResponseEntity, into a helper method called "getOrderResponseEntity", it can be seen in Figure 5

Figure 5: Helper method added to extract part of the logic in getNextOrderForVendor

This decreased its cyclomatic complexity to a casual metric level of 3. With this refactoring, lines of code were decreased to 31 and it was observed that 10 of these lines are occupied by the JavaDoc, although this was not removed, the actual line of codes were determined to be 21, making it a casual-level metric.

The uncommon metric values of the rest of the methods in this class were not evaluated as most of them were caused by the lines of code covered by JavaDoc and path parameters from the OpenAPI.

3.3 StatusController

This class was chosen not because of its class-level metric values but its complicated methods specifically. Therefore the only uncommon metric value here is its number of overridden methods, whose resolution was mentioned beforehand.

It was common in this class to have the cyclomatic complexity and the lines of code in unusual values, such as in method updateToAccepted and updateToGivenToCourier. This can be seen in Figure
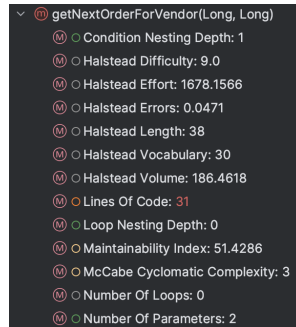
Figure 6: Metrics after refactoring for getNextOrderForVendor, Lines Of Code accounting for JavaDoc
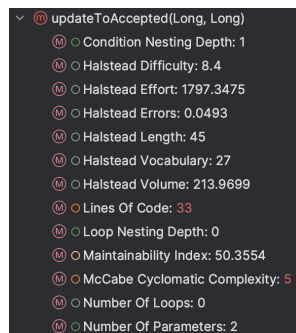
7 and Figure 8.

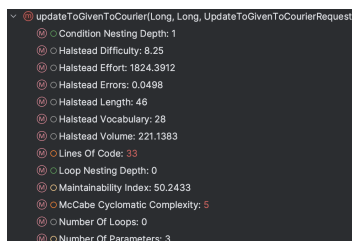Figure 7: Method-level metric values of updateToAccepted in StatusController before refactoring

Figure 8: Method-level metric values of updateToGivenToCourier in StatusController before refactoring

We observed that the reason for these values were common in almost all methods that update the status. All of them first check the previous status of an order, and then continue with their logic. We extracted this check into a helper method called "checkPrevStatus" 9 that gets the current status of an order and maps it to the needed response type, comparing it to the expected status value. Therefore we were able to solve the problem of 5 methods that had these unusual metrics by only adding 1 helper method. These changes can be seen in this commit

Part of these results can also be seen from Figure 10, as the cyclomatic complexity was lowered by 1 and the lines of code were lowered by 3, making them both casual metric values. As well as in Figure 11, where we can see the same change in cyclomatic complexity, but not in the lines of code

Figure 9: Refactored method now used in almost all StatusController methods

because of its longer JavaDoc.

Figure 10: Metric values after refactoring of updateToAccepted

Figure 11: Method-level metrics after refactoring for updateToGivenToCourier

3.4 StatusService

All of the class-level metrics seem to be labeled as "casual" by the plug-in, therefore we evaluate the class on a method level.

3.4.1 updateStatusToDelivered

This method was chosen because of its "unusual" metric value of cyclomatic complexity (over 4) and lines of code (over 30). Once again the threshold values are given by the plug-in.
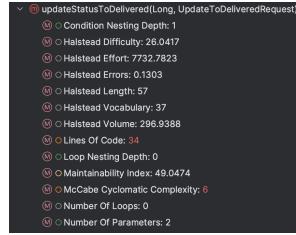
Figure 12: Method-level metric values for updateToDelivered before refactoring

We have noticed that this is because, unlike the other status updating methods, this method checks for the previous status in the same place as assigning the new status. Therefore to solve this problem we refactored both this method and the related controller method to use the previously created helper formula in Figure 9. So now this method only checks for valid time values and updates the status, checking for the valid previous status value is handled in the extracted method. These changes can also be seen in this commit.

As can be seen from Figure 13, the cyclomatic complexity was reduced by 2, making it below the threshold. Additionally, the lines of code were reduced by 8 as the creation of the timeValues object was made to be shorter and more straightforward in the method, putting this metric into a casual value.
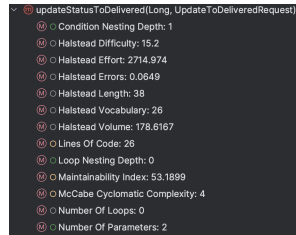

Figure 13: Method-level metrics for updateStatusToDelivered after refactoring

3.5   UserService

This class was chosen because we wanted to try and improve the Maintainability Index as much as possible as seen in Figure 14. The value of 36.3 is already good by the Microsoft scale [1], we wanted to see how much more we could improve it. Metrics that were above the norm were the Number Of Methods and the Weighted Methods Per Class. Regarding the Number Of Methods metric, we noticed that all of the methods use either VendorRepository or CourierRepository and so as an improvement, we propose a clear logical division of UserService into two classes: CourierService and VendorService. This division would split UserService into six methods for CourierService and six methods for VendorService, which for both would fit into the norm of the Number Of Methods metric. This split would also solve the Weighted Methods Per Class metric. After the split, the resulting metrics for CourierService can be seen in Figure 15. The Maintainability Index increased to 40 and even though the Number Of Methods metric shows as above norm, it is necessary to point out that it also counts constructor and so without it CourierService is exactly in the upper bound of the norm of the Number Of Methods metric. The resulting metrics for VendorService can be seen in Figure 16. The Maintainability Index increased to 39.68 and regarding the Number Of Methods

metric it's the same case as with CourierService, without counting the constructor it is in the upper bound of the norm. Since the split resulted in fewer methods per class, we also improved the Weighted Methods Per Class metric with it.

Figure 14: Class-level metric values of UserService

Figure 15: Class-level metric values of CourierService

3.6 CourierService

After reviewing the CourierService methods we noticed that there is one, specifically getting courier by ID, which is duplicate, see Figure 17. This improved our Number Of Methods metric which made it green.

3.6.1 Improving the Maintainability Index

One of the metrics that have an impact on the Maintainability Index is the Lines Of Code. We noticed that several methods have unnecessary try-catch statements around the Spring Data JPA saveAnd-Flush method. We reviewed the documentation and found out that saveAndFlush does not actually throw IllegalArgumentException and so the try-catch statements are unnecessary[2]. After removing the unnecessary code parts the Maintainability Index increased in CourierService to 43.75 and in VendorService to 41.92.

Figure 16: Class-level metric values of VendorService

Figure 17: Duplicate method

3.6.2 Results

After improving the Maintainability Index metric our Weighted Methods Per Class metric also improved because the methods decreased complexity. The final metrics can be seen in Figure 18 for CourierService and in Figure 19 for VendorService and implementation in this merge request.

Figure 18: Class-level metric values of CourierService after improvements

3.7 Notes on code style - PMD

The project has a considerable amount of PMD warnings. We have looked into them and considered implementing changes suggested, but decided that this was not productive. In this section we explain our thinking behind this decision.

Excluding irrelevant checks would either require marking all non-conforming entities or downloading a PMD ruleset to be used instead of the default one (otherwise we have not found a way to

Figure 19: Class-level metric values of VendorService after improvements

exclude the check). Since the first approach adds clutter and requires a lot of time, and the second approach runs the risk of accidentally using a ruleset different than the default one, we decided to leave the warnings as they are and ignore them. Here you can find explanations about why we disregard some of the warnings.

3.7.1 ”The String literal ... appears ... times in this file”

In all of the instances of this warning, we have found no way to avoid the usage of the repeating literal, as it is usually something deeply core to the functionality of the class, such as ”orderId” or ”403” (the error code).

3.7.2 ”Found non-transient, non-static member”

Since this warning does not affect the usability of our application in any noticeable way, it could be safely excluded from the conducted checks.

3.7.3 ”Field ... has the same name as a method”

We have found that this warning is not too helpful in our case, as in our small application the repeating names add more clarity than they remove.

4 Mutation Testing Report

4.1 Introduction to mutation testing before improvements

For mutation testing, we used the PITest library. After generating our PITest report you can see in Figure 20 we analyzed what strengths and weaknesses our code has. We went over each of our non-generated classes as each part was missing out on line coverage or mutation coverage. As you can see in the PITest report, it contains all of the elements in the package, including those that were generated from our OpenAPI specification. That is the reason that our overall mutation coverage in the report is very low 43%, disregarding the generated elements our average was 88%. The overall Jacoco coverage can be seen in Figure 21.

Pit Test Coverage Report

Project Summary

Number of Classes	Line Coverage		Mutation Coverage	
23	47%	432/914	43%	213/491

Breakdown by Package

Name	Number of Classes	Line Coverage		Mutation Coverage	
nl.tudelft.sem.template.api	5	0%	0/174	0%	0/89
nl.tudelft.sem.template.example.authorization	4	64%	45/70	75%	18/24
nl.tudelft.sem.template.example.config	1	0%	0/10	0%	0/5
nl.tudelft.sem.template.example.controllers	3	95%	194/205	90%	117/130
nl.tudelft.sem.template.example.domain.order	2	89%	84/94	77%	36/47
nl.tudelft.sem.template.example.domain.user	1	100%	31/31	100%	12/12
nl.tudelft.sem.template.example.externalservices	1	100%	6/6	100%	1/1
nl.tudelft.sem.template.model	9	28%	99/360	19%	37/196

Report generated by PIT 1.5.1

Figure 20: Pitest - before

example-microservice

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
nl.tudelft.sem.template.model		19%		5%	155	205	261	356	89	138	1	11
nl.tudelft.sem.template.api		0%		0%	92	92	174	174	62	62	5	5
nl.tudelft.sem.template.example.authorization		69%		76%	5	17	16	40	2	7	0	2
nl.tudelft.sem.template.example.controllers		95%		93%	9	80	9	205	2	27	0	3
nl.tudelft.sem.template.example.domain.order		88%		85%	7	40	11	95	3	20	1	3
nl.tudelft.sem.template.example.config		0%		n/a	2	2	9	9	2	2	1	1
nl.tudelft.sem.template.example.externalservices		67%		n/a	1	3	4	10	1	3	1	2
nl.tudelft.sem.template.example		0%		n/a	2	2	3	3	2	2	1	1
nl.tudelft.sem.template.example.domain.user		100%		100%	0	10	0	31	0	9	0	1
Total	2,192 of 3,878	43%	204 of 359	43%	273	451	487	923	163	270	10	29

Figure 21: Jacoco - before

4.2 Services

4.2.1 OrderService

For OrderService PITest generated 22 mutations out of which 17 were killed. From the 5 mutations that were not killed, we found out that we have no coverage for 2 methods which also explained that our line coverage was at 78%. After adding tests for the missing methods, we reached mutation coverage of 100% for OrderService.

4.2.2 StatusService

For StatusService PITest generated 25 mutations out of which 19 were killed. Even though we had 100% line coverage, 6 mutations survived so we had to analyze our StatusService tests and find out what they are missing. We found out that in all of our tests for status change (we have 5 methods for each status), we only check if the required attributes of the passed order were changed, however, we missed checking the attributes of the returned value. Additionally, method updateStatusToDelivered updates not only status but also several other attributes like time values. With mutation testing, we found out that these values aren't updated correctly because a mutation that removed the time values update survived. After fixing the updating, we reached 100% mutation coverage on StatusService.

4.3 Controllers

4.3.1 OrderController

For OrderController PITest generated 42 mutations out of which 36 were killed. 4 mutations had no coverage which meant our line coverage was also not 100%. For those we found out that four methods were missing tests for status code 403, meaning that the user was not authorized to make this request. 2 mutations survived which might have meant wrong implementation. However, after a careful look, we found out that two tests were missing assertions.

4.3.2 StatusController

For OrderController PITest generated 48 mutations out of which 38 were killed. 5 mutations had no coverage so we added the missing tests usually regarding status code 403. Two mutations however were not killed because our controller implementation considered also the fact that the order was deleted while the method was running. Adding these tests improved the mutation coverage to 100%.

4.4 Authorization

For the authorization directory, 6 mutants out of 24 survived making the mutation coverage 75%. 4 of them were in the Authorization class, 1 in the AuthorizationService class and 1 in the Validation class. The issue related to this was Test Authorization and the commit that brought the mutation coverage to 100% for the whole directory was Testing.

4.4.1 Authorization

The authorizeAdminOnly method was not tested beforehand, and after carefully testing the behavior all mutations were killed. Also, one mutation had to do with the method parseUserType which receives a String representing the user type returned from the user microservice and converts it to an enum.

16

The method uses a switch expression and the default case is to throw an IllegalArgumentException, meaning the user type received is invalid. A test asserting that the authorization returns a 500 internal server error and an isolated test with an invalid user type killed the mutant.

4.4.2 AuhorizationService

Again, the authorizeAdminOnly method was not tested and one test asserting its behavior killed the mutant.

4.4.3 Validation

The method check was not tested for the case where there was an error while applying the validation. That is, the try-catch block was not tested for the case where an exception was caught. One simple test killed the mutant.

4.5 Mutation testing after improvements

After our improvements, the mutation coverage increased to 100% as you can see in Figure 22 and implementation in this commit (disregarding authorization testing). We have also excluded generated classes from our report such as model or api implemented here. The Jacoco coverage also improved to 100% as seen in Figure 23

Figure 22: Pitest - after

Figure 23: Jacoco - after

5 System-level Testing

5.1 Testing through Postman

We divided our system-level tests into two: ones that require a connection to the other microservices and ones that are able to test the whole delivery system in its own logic, without dependencies to the other systems. We discuss the first kind of tests in the section "Integration" in this report.

We aimed to make the testing of the functionality for the second kind of system tests as easy as possible. For this purpose, we made a separate branch, named system-tests. In this branch, we made and exported an extensive list of calls to different endpoints that showcase the functionality of the microservice, they can be accessed in the file with the path "example-microservice/src/main/resources/system-tests-for-integration"

In this branch, all of the main functionality of the system is the same as in main, only with a few changes to the authorization to be able to test all the functionality. Since some functionality is only accessible to users with admin permissions, we had to make the tests with an admin user. However, the only way to add an admin is through the h2-console of the users microservice. Although we first manually tested our functionality with this h2-console step, we have decided that we want to alleviate the procedural complexity of making these tests and enhance the efficiency of the whole process. That is the reason we created the system tests branch with a minor change that let's us retrieve the admin ID locally, no other functionality was changed. We argue that this also does not take away from our coverage as the authorization was tested separately on the system level as stated in the section "Integration".

To design our system tests, we used high-level documents as a reference, in our case this was our requirements as they cover the usage scenarios and the overall software architecture. We have determined that each requirement is closely tied with one or more endpoints but there is not an endpoint that covers more than one requirement. Therefore, to judge our requirement coverage, we went off of the use of the endpoints for reference. This can be seen in the formatting of our functional tests, exported from Postman.

Out of the 17 requirements implemented, 15 were tested in this section through system-tests. This gives us 88.2% coverage on our requirements. This is excluding the functional testing of our functionality dependent on our integration, which - as stated above already - is discussed in the "Integration" part of the report.

Some methods cannot be tested this way, as for them the authorization id has to be the id of the vendor/courier. There are four endpoints that get a 404 in the tests, this is why our coverage is not 100%. For these to run, the User microservice should be running and users/vendors with required properties should be added to the database, sometimes manually. Without running that microservice we can't easily authenticate vendors/couriers, which is what these methods require.

6 Integration Report

6.1 Agreement with other microservices and design choices

From the start of the project, we considered adaptability of our microservices in our entity design. This is why we gathered 2 representatives from each microservice to agree on a general draft of our entities to ensure consistency and avoid compatibility problems at the start of the project. Throughout development, this contract as well as the yaml was respected as much as possible. If it was critical that something must be changed that impacted other microservices, apart from small internal changes that

did not affect other microservices like changing a method name (not path), they were communicated through issues like in here.

6.1.1 Why do we store the fields that we store?

In our entity design, our main focus was to keep the system as functional as possible, while also creating a scalable environment with not a lot of overhead. Storing too many fields with unnecessary overlaps with other microservices would mean there would be too many requests sent to keep it consistent, whereas not storing anything our functionality relies on would lead to the same issue as we would keep making requests to get the information we need. Therefore, through many drafts and discussions with TA's, we decided to only store fields of entities that our requirements modified or relied on. Most of our fields (eg. the reference from a courier to their boss), were not relevant therefore not stored in other microservices. This made the overlap with other microservices as small as possible, intersecting only in a few fields such as the status of an order which we made the decision to keep because a great part of our functionality relies on it, it is related to at least 6 of our endpoints. Similar reasons were behind our decision to keep the fields that overlapped with other services, such as keeping the statistics functionality fast and making the implementation compatible for adding more statistics features later on.

We acknowledge that this decision comes with its drawbacks as well, as stated, added overhead in other microservices' side to keep our database consistent, this being unfeasible in a large-scale system (more than 10 microservices) as there would be no way for a microservice to keep track of which other microservice needs to be updated. However, because of the aforementioned performance and flexibility reasons, as well as our communication with other systems through issues that will be referred to throughout this section and the scale of our project, we have decided to go for this approach.

6.1.2 Data propagation

We as the delivery microservice had impressions of 2 possible ways to ensure consistency among databases: whenever we execute a functionality related to an entity, querying the other's microservice and checking the whole table to see if our database has inconsistent or missing information, or whenever there is a change made in a shared entity or added entity in another microservice, that microservice propagates that change to ours using set endpoints. As an example, the first approach would mean, whenever we are dealing with a vendor in any functionality, no matter if the id is already in our database or not, we would have to make a request to the users microservice to get all the vendors, compare each one of them to references we have in our database to see if the fields match or if we have any missing fields, update our own database accordingly, and then keep on with the functionality. Whereas the second way would mean that whenever a new vendor is created, they call our endpoint to propagate the change. We figured the first way, checking almost every field of every object ever saved in the other microservice's table whenever we are executing a function would cause a lot of overhead, which would decrease the scalability of the system tremendously. We therefore opted for the second approach, and reflected this to the other microservices during the planning phase through this issue, which was made 1 month before the deadline, towards the end of November.

Since there was no disapproval of the issue, we built our design around this notion and added specific endpoints to create vendors and couriers only by their ids, not objects, in order to make this process of propagating data easier, as the common intersection in our entities were mostly only ids and we wanted to keep these consistent. This notion was communicated again with other microservices when it was closer to the time of integration, through this issue. However, the last lab before the

deadline, we realized that there was a miscommunication and the other microservice believed that we would keep the consistency using the first way that was mentioned, us querying their database whenever we deal with our entities, and disagreed with our proposal of choosing the second way, them pushing the changes to us instead of us constantly pulling the information from them. This unfortunate miscommunication caused our databases to not be fully consistent automatically in our integration.

A potential solution for a longer term project would be to communicate more with the other microservices on which propagation choice to implement, while also making sure that they give a positive response to proposals before starting implementing. Further communications could be done while weighing on each sides the drawbacks of both approaches and lastly changing our implementation according to the final decision. Unfortunately, this was unfeasible for this short-term project. Therefore, during our integration, we do not focus on the consistency of databases overall but instead consider integration in terms of the functionality in our requirements.

6.2 Results of the integration

Because of our design choices explained in the section "Why do we store the fields that we store?", the only dependency of our functionality on other microservices was in authorization, when we query the type of a user. Data propagation was not evaluated as stated in "Data Propagation" section. Therefore our integration was successful in this regard.

We configured our integration on our integration branch. We ran the main branches of the other two microservices, only the port of the users microservice had to be changed to 4269 as the previous one was already used in Mac users. We had to propagate this change in the orders microservice "applications.properties" file by changing the 17th line to "external.users-microservice.url=http://localhost:4269". After making these changes we were able to run both of them and integrate our services.

We compiled the system tests for integration features we made through postman in the file "example-microservice/src/main/resources/system-tests-for-integration" in our repository. In these tests, we have extra requests to populate our own database, as discussed in section "Data propagation", in the initial design these requests were assumed to be made by other microservices, we do this manually in our case to test our own functionality.

6.2.1 Authorization

In the initial stages, the Authorization process encountered difficulties as attempts to place an order for our service, using the ID of an admin created in the user microservice, were unsuccessful. The root cause was traced to our assumption that the user type would be provided without any additional text in the JSON response. This oversight led to a 500 internal server error response, as our method logic erroneously classified the JSON response as an invalid user type. Upon identifying the issue, we promptly addressed it by implementing proper parsing of the JSON response. The commit on the integration branch is: MakeAuthorizationWork. Despite this, the current state of text authorization using only Postman requests faced a limitation. An admin user was required to place an order in our microservice, and the user microservice lacked an endpoint for creating an admin. To make it functional, manual insertion of an admin into the user database was necessary, facilitated through the H2 console. Once this step was completed, authorization worked seamlessly. However, to enable Postman functionality, we had to hardcode the admin's ID to a predefined value. This adjustment was implemented in the HardCodeAdminId commit. Additionally, a minor oversight in the implementation was addressed in this commit. These changes collectively allowed successful testing of

authorization as well as the validation steps using Postman exclusively.

6.2.2 Updating status

Although, as was argued in section "Data propagation", we do not evaluate data propagation in terms of populating our database, we still wanted to integrate our part of the miscommunicated choice: propagating the change we make to shared fields to other microservices. To prove that our service is compatible to do this, we added requests to the Orders microservice whenever we change the status of an order. The commits can be found here.

Although creating the request was first difficult as it was hard to get past the authorization without users being propagated through microservices. Especially also because there was no way other than using the h2-console for adding an admin to the users microservice. The way we got around this in this case was to create vendors and customers in users microservice to get past authorization. To get past our own authorization, we needed a vendor to make an order, (same data propagation issue mentioned above) therefore we added and if-clause in our authorization for the sake of the test, after checking via doing it through h2-console and actually testing the authorization as mentioned above. Also through manually propagating the addition of users through requests, the integration was successful. The process is as follows: adding a customer to the users microservice, registering a vendor in users microservice, creating dishes in orders microservice, making an order in orders microservice (status pending), changing the status of the order to accepted in delivery microservice, as can be seen from the Postman file, the change being propagated to the orders microservice as the status we get from the added order is accepted. The result of the tests added through the Postman JSON files can also be seen in the Figure 24.
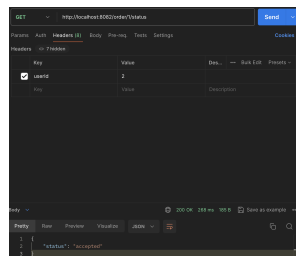

Figure 24: Last step of our system tests for status changes proving that the change was propagated

6.3 Reflection on integrations and challenges

Our main issue was caused by miscommunication between different microservices. Although our integration for our functionality was successful with little challenges - explained in section "authorization" and "Updating status" - we had to manually keep the databases consistent throughout the integration. In a large-scale application, this would be unfeasible, therefore a possible eventual solution would be to have a healthier and more open communication with other microservices to investigate the problem, the specifics of this solution is discussed in detail at the end of the section "Data propagation".

Overall, because of the success in our functional integration as well as us being able to propagate our own changes, if it were not for our miscommunication issue, we believe that a full integration with automatic entity propagation would be successful.

References

[1] Microsoft. (n.d.). Code Metrics - Maintainability Index Range and Meaning. Retrieved from `https://learn.microsoft.com/en-us/visualstudio/code-quality/code-metrics-maintainability-index-range-and-meaning?view=vs-2022`

[2] Spring Data JPA official documentation. Retrieved from `https://docs.spring.io/spring-data/jpa/docs/current/api/org/springframework/data/jpa/repository/JpaRepository.html#saveAndFlush(S)`