Final report - Team 5b

Georgios Kontos, Tudor Chirilă, Rein Lakerveld

January 18, 2024

1 Introduction

In this final report we will go into the list of requirements we implemented, the design patterns implemented, the software quality of the project, mutation testing and integration with the other microservices. For agendas, minutes and sprint retrospectives and general information about our group and the project we kindly ask you to refer to the project's README.md, where you may find helpful links for those matters.

2 List of requirements implemented

From our initial list of requirements, we managed to implement most of the must and should have requirements and 2 out of the 3 could have requirements, as we did not provide PC chairs with analytics.

Here is our initial list of requirements:

1. Must haves

    (a) API's for easy integration with other independent systems.

    (b) Support for stake holder entities and reviews.

    (c) Each stakeholder is identified by different JWT tokens.

    (d) During the bidding phase, reviewers are only allowed to see the title and the abstract of each paper.

    (e) The system assigns all submitted papers for review.

    (f) The system should identify which reviewers are eligible to review each paper, based on possible conflicts with authors.

    (g) An automated process (random selection) is used to assign papers to reviewers, which is taking information about reviewers into consideration.

    (h) The system assigns each paper to exactly three reviewers.

    (i) The PC chair is able to assign papers to reviewers.

(j) After all papers have been assigned, each reviewer is able to edit the reviews they are assigned.

(k) Each review consists of a confidence score, confidential comments, comments for authors and a final score.

(l) If a paper has only positive or only negative scores, the system accepts it or rejects it respectively.

2. Should haves

(a) Reviewers can decide whether they want to review a paper, by labeling it as "can review", "neutral" or "cannot review".

(b) In case of conflicting scores on a paper, the PC Chair can make discussion comments and request reviewers to agree on a final verdict.

(c) Reviewers can check the final decision of papers they have been assigned to.

3. Could haves

(a) The PC chairs are able to change the paper assignments before finalization.

(b) After reviewing a paper, the reviewers can see comments made on that paper by other reviewers.

(c) The system provides statistics to PC chairs and general chairs regarding the number of papers that will be accepted, rejected and with no final verdict.

4. Won't haves

(a) The reviewers have a balanced/fair assignment.

The only requirements that were not implemented in our microservice were 1 (c), 3 (c) and 4(a). We did not implement 1 (c) because it was a common decision with the other two teams not to use JWT tokens for this project. We would have liked to implement 3 (c) and 4 (a). Regrettably, because three of our team members dropped out, there was no the time to.

3 Design Patterns

3.1 Chosen Design Patterns

This changes for this part of the report were merged in merge request 52.

3.1.1 Chain of Responsibility

- **Mechanism & Inner Working Implementation:** All the classes in `nl/tudelft/sem/template/example/domain/validator`.

- **Implementation:** All endpoints in the `nl/tudelft/sem/template/example/domain/controllers` module, inside `PaperController` and `ReviewController`, specifically the `chainManager.evaluate()` method.

3.1.2 Builder

- **Mechanism & Inner Working Implementation:** All the classes in `nl/tudelft/sem/template/example/domain/builder`.

- **Implementation:** All endpoints in the `nl/tudelft/sem/template/example/domain/controllers` module, inside `PaperController` and `ReviewController`, specifically lines containing

```
CheckSubjectBuilder builder = new CheckSubjectBuilder();
// set various parameters for builder here, such as:
builder.setUserId(userId)
CheckSubject checkSubject = builder.build();
```

3.2 Why Chain of Responsibility?

Take, for example the `PaperController` class. When looking at various endpoints implemented here, it is easy to see the following:

- There are various checks being performed at the beginning of each method.

- Many of these checks get repeated in the most (if not all) the endpoints. Take, for example, the null check of the Parameters, and the User Validation.

- Each time, the same check returns the same error code. Take, for example, `HttpStatus.UNAUTHORIZED` for user validation.

- All of these checks are performed in the same (logical) order inside each endpoint, forming some kind of pipeline that validates the information this method uses before performing the actual business logic of the endpoint. If something invalid happens on the way, the program gets out of this pipeline and returns an error code.

All of these clues, and especially the last one, hint toward the idea of using the Chain of Responsibility pattern.

However, to be sure that this is a good choice before heading on to implement this pattern, let's look at the recommendations of situations where its implementation is suited.

We find in Slide 58, SlideSet 05-UML+DesignPatterns - Part 1:

"When to use?

More than one object needs to handle a request, handler is not known a priori. Should be determined automatically/dynamically.

You want to issue a request to one of several objects, without specifying the receiver explicitly."

In this case, due to the Low Coupling principle, meaning one class should have a single clear, well-defined task, the data that needs to be checked would be handled by three different validator objects. These would be :

1. Object for validating method parameters validity (including the null checks mentioned in the observations earlier) - called `ParameterValidator` in our implementation,

2. Object for validating a user - called `UserValidator` in our implementation,

3. Object for validating database entities that the endpoint works with - called `DatabaseObjectValidator` in our implementation.

Thus, it is clear that this piece of data (remember this "piece of data" for later) needs to be handled by more than one object and that the first part of condition 1 from the slides is met!

Condition 1 says that the "handler is not known a priori. [It should] be determined automatically/dynamically.". This is specifically why we have designed the methods `evaluate` and `createChain` inside `ChainManager`.

The method `createChain` deals with creating the chain *dynamically*, according to the content of the data object, which is a `CheckSubject` instance. The caller object does not know the handler by which this Validation will be handled. They just call `chainManager.evaluate(CheckSubject instance)`, which will first create the chain through `createChain`, and then performs the appropriate evaluation.

Condition 2 states that "You want to issue a request to one of several objects, without specifying the receiver explicitly." We want the validation to be performed, without specifying the receiver explicitly. This holds, due to the `ChainManager.evaluate(CheckSubject checkSubject)` abstraction! This, in turn, passes on the object to be evaluated by several `Validator` objects, as the condition requires.

So, both personal instinct and the two conditions from the lecture provide a strong argument for using the Chain Of Responsibility pattern.

3.2.1 Diagram for the Chain of Responsibility Pattern

The class structure for the Chain of Responsibility is the following (for the original image files please visit the repository):

4
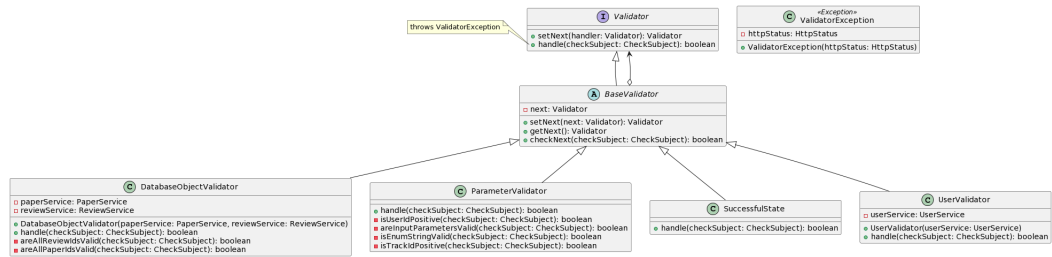
Figure 1: Chain of Responsibility diagram

3.3 Why Builder?

We now need to pass this piece of data through the chain. There are three things to have in mind here:

1. There is a lot of data to be wrapped and fed forward to the chain - like a list of parameters, a user ID, a list of paper reviews...

2. We do not know how the application might increase in the future - be it more Validators in the chain, or more data to be wrapped in this object. Thus, this class will be very hard to maintain, and can violate the open-closed principle if not created properly.

3. The final state of the object can have multiple different shapes. For example, we can have a bundle of data that contains both a list of parameters and a list of paper IDs. On the other hand, we could also have a bundle of data that only contains a user ID.

There is, luckily, a design pattern that is made specifically for this: the Builder.

The use of the pattern was obvious, considering the problem we had fit perfectly with the ones a Builder solves. When we have a look over the basic rules for using the Builder pattern presented in the slide 61 from set 06-DesignPatterns-Part 2, we see two contexts that point towards Builder:

"- The algorithm for creating a complex object should be independent of the parts that make up the object and how they're assembled.

- The construction process must allow different representations for the object that is constructed."

The first one matches with our first two observations. We have a complex object that might get even more complex in the future. Thus, decoupling the assembly of multiple parts of this object from the algorithm for creating it follows the open-close principle and leads to high maintainability. When we add a new field, we only create a new setter inside `CheckSubjectBuilder`, a new field in `CheckSubject`, and we just add an extra field in the `build` function

inside the `CheckSubjectBuilder`. However, every place where we created a `CheckSubject` instance remains unchanged due to this decoupling: the call to the `build()` method that instantiates a `CheckSubject` object stays the same.

The second one is ideal for our third observation. We can easily implement highly varying forms of a `CheckSubject`, with minimal changes: we just activate the setters in the `CheckSubjectBuilder` of the fields we want the future `CheckSubject` to have active. We can create one object with just a userId, or one with a trackId and a list of paperId's, and so on.

In the same slide, three advantages of implementing the Builder pattern are included:

"

- It lets you vary a product's internal representation.
- It isolates code for construction and representation.
- It gives you finer control over the construction process.

"

These points reiterate that this is the ideal context for using the Builder Pattern.

It is clear now that the Builder pattern would be the best solution for the implementation of this piece of data that is fed to the validating chain.

3.3.1 Diagram for the Builder Pattern

The class structure for the Builder pattern is the following (once again, the original image can be found in the repository):
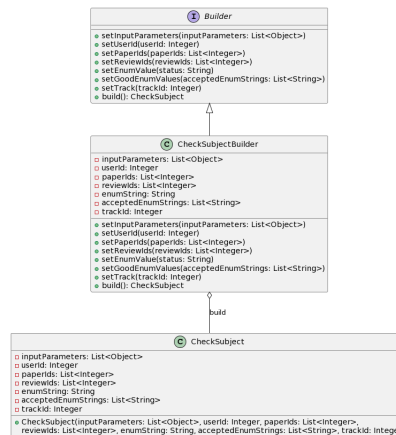
Figure 2: Builder pattern diagram.

4   Software Quality Report

In this part of the report, we will make use of software metrics tools like MetricsTree to analyze and improve code quality. We will be removing redundant code, eliminating code style issues using Checkstyle, and reducing cyclomatic complexity where necessary.

4.1   Dispensable code

Before refactoring, we had a lot of dispensable code. As examples, we got two microservices, Authentication and the Example one. We have built our project on the basis of the example-microservice. We don't use anything from the Authentication microservice at all. This is therefore totally dispensable.

In the merge request 37 we removed all the unused and dispensable code related to the Authentication microservice. The specific commit was commit f9dcd26434e7d0c657e9631fd8d7d0563d2bd1f7.

4.2   Code style

During the project, we did not take enough care of our code style. Hence, we had many checkstyle violations, 475 to be exact. This needed to be improved on, as bad code style makes code less pleasant to read, and more importantly, may make it more difficult to understand.

In the merge request 41 all the checkstyle violations were fixed. It concerns these commits specifically:

- Commit 6de327d896573fc48dd560a84843c20a30a26359

- Commit 47d60e74dbbb0f39cba367014b2521c89df6539b

4.3   Long parameter list

In the microservice there is one method with a lot of parameters, namely the constructor of PaperResponse with 8. However, this object is only to receive a paper from another microservice via their REST API. Therefore we did not feel the need to refactor this. All the other methods where fine when considering the number of parameters.

4.4   High cyclomatic complexity

In our entire project there where two methods with a high cyclomatic complexity. In the following merge request we refactored them to reduce cyclomatic complexity and to increase code comprehensibility.

This is the merge request: nr. 42

- The first method is ReviewService.assignReviewsAutomatically. Commit becba43aa99fa342fb4475e915afa5a73316c5f5 refactors it by extracting two

methods. The method assignReviewsAutomatically takes a list of a papers and assigns reviews to all of them. The first method we extracted was the part where a single paper gets assigned its reviews (this is the method assignReviewersToPaper). From there we extracted another method that assigned reviews to the paper in case no ideal match could be found.

- The seconde method is PaperController.paperUpdatePaperStatusPut. Commit 2e0faefc60ee554210ee7b1eda9efbbefc6d045b refactors it. This one was much easier to refactor, as the cyclomatic complexity was much lower. We moved some logic from the PaperController class to the PaperService class to make sure the only logic in PaperController.paperUpdatePaperStatusPut was logic to determine the http status codes.

4.5 Detecting blobs

There were four classes that are quite large with a lot of relatively complex methods that might therefore be a blob, specifically PaperController, ReviewController, PaperService and ReviewService. After these initial improvements you read about above, this final part will inspect these large classes and present some improvements for them.

4.5.1 PaperController and ReviewController

When looking PaperController or ReviewController with MetricsTree, besides the size and the number of methods, one thing jumps out, namely the high coupling of PaperController with PaperService. Reducing the number of methods for the PaperController and the ReviewController is quite difficult, as they implement the APIs as defined in the specifications. Since we prefer to not change our specification we are going to leave these controllers for now.

4.5.2 ReviewService and PaperService

The ReviewService was much easier to simplify. The methods for assigning reviewers (and thereby reviewers) to papers could be made entirely static, so all these methods were moved to the ReviewUtils class. Both PaperService and ReviewService had methods like these, so those methods and their tests were moved to a util directory. Commit 71e6b5f61a64a3dde81898d72363160d8632b387 and merge request 51 detail this. Please note there were some error is this initial commit and that there are some commits with fixes for those errors.

5 Mutation Testing Report

In this section, we will analyse our mutation score with Pitest. First, we will show our mutation score before we improved our test suite. Afterwards, we will also show the mutation score after the improvements in our test suite, and also explain which changes have helped us achieve that. All of our statistics only

account for the code written by the team members, as we have excluded the code generated from the OpenAPI specifications.

5.1 Before

5.1.1 Overview

Before implementing any changes our test suite would kill 204 out of the 289 mutations, returning a mutation score of 70%.

nl.tudelft.sem.template.example.config	1	90%	9/10	0%	0/5
nl.tudelft.sem.template.example.domain	1	0%	0/7	0%	0/2
nl.tudelft.sem.template.example.domain.controllers	2	93%	153/165	82%	146/178
nl.tudelft.sem.template.example.domain.events	1	93%	26/28	54%	13/24
nl.tudelft.sem.template.example.domain.models	3	56%	35/63	20%	2/10
nl.tudelft.sem.template.example.domain.responses	1	100%	12/12	100%	2/2
nl.tudelft.sem.template.example.domain.services	6	88%	126/143	60%	41/68

Figure 3: Overview of our mutation scores before changes

5.1.2 Controllers

Name	Line Coverage		Mutation Coverage	
PaperController.java	87%	68/78	77%	67/87
ReviewController.java	98%	85/87	87%	79/91

Figure 4: Controllers classes before changes

5.1.3 Models

Name	Line Coverage		Mutation Coverage	
PcChair.java	84%	26/31	100%	2/2
PreferenceEntity.java	0%	0/17	0%	0/4
TrackPhase.java	60%	9/15	0%	0/4

Figure 5: Models classes before changes

5.1.4 Validators

Due to implementing the Design Patterns and the mutation testing in parraler, the validators classes were not included in the initial overview. Here is the mutation score for those classes prior to changes:
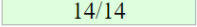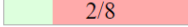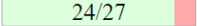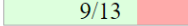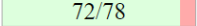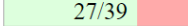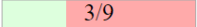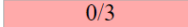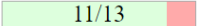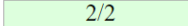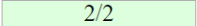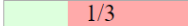
Name	Line Coverage		Mutation Coverage	
CommentService.java	100%	14/14	25%	2/8
PaperService.java	89%	24/27	69%	9/13
ReviewService.java	92%	72/78	69%	27/39
ReviewerPreferencesService.java	33%	3/9	0%	0/3
TrackPhaseService.java	85%	11/13	100%	2/2
UserService.java	100%	2/2	33%	1/3

Figure 6:   Service classes before changes
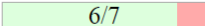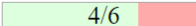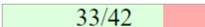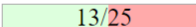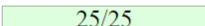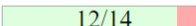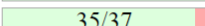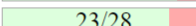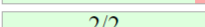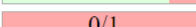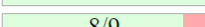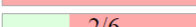
Name	Line Coverage		Mutation Coverage	
BaseValidator.java	86%	6/7	67%	4/6
ChainManager.java	79%	33/42	52%	13/25
DatabaseObjectValidator.java	100%	25/25	86%	12/14
ParameterValidator.java	95%	35/37	82%	23/28
SuccessfulState.java	100%	2/2	0%	0/1
UserValidator.java	89%	8/9	33%	2/6

Figure 7:   Validator classes before changes

5.2   After

After implementing changes, our test suite kills 292 mutations out of the 334 generated, returning a mutation score of 87.5%. Merge request 43 and 57 contain all the changes, which mainly focus on handling additional edge cases that where initially ignored. Figures 8 - 12 showcase the results after the changes.
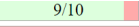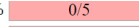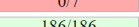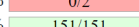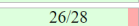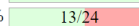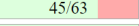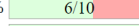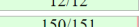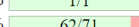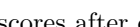
nl.tudelft.sem.template.example.config	1	90%	9/10	0%	0/5
nl.tudelft.sem.template.example.domain	1	0%	0/7	0%	0/2
nl.tudelft.sem.template.example.domain.controllers	3	100%	186/186	100%	151/151
nl.tudelft.sem.template.example.domain.events	1	93%	26/28	54%	13/24
nl.tudelft.sem.template.example.domain.models	3	71%	45/63	60%	6/10
nl.tudelft.sem.template.example.domain.responses	1	100%	12/12	100%	1/1
nl.tudelft.sem.template.example.domain.services	6	99%	150/151	87%	62/71

Figure 8:   Overview of our mutation scores after changes

6   Integration Report

This section of the report details the different aspects of the integration. There were many challenges, plenty of which were overcome. In the following there will be many examples and the way they have or have not been overcome. In the end, the integration was a failure, however we think the effort was instructive
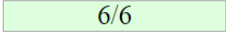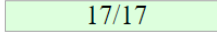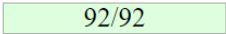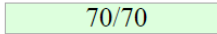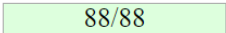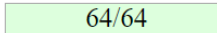
Name	Line Coverage		Mutation Coverage	
NullChecks.java	100%	6/6	100%	17/17
PaperController.java	100%	92/92	100%	70/70
ReviewController.java	100%	88/88	100%	64/64

Figure 9: Controllers classes after changes
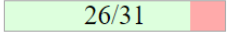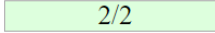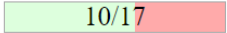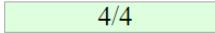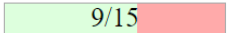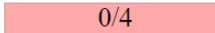
Name	Line Coverage		Mutation Coverage	
PcChair.java	84%	26/31	100%	2/2
PreferenceEntity.java	59%	10/17	100%	4/4
TrackPhase.java	60%	9/15	0%	0/4

Figure 10: Models classes after changes

for a future integration.

6.1 Setup

To start our integration process, we took the two microservices from the other team's repositories and we copied them over to our git repositories in the branch 'integration'. We changed the Gradle and Gitlab CI files to make sure all of them would be included in the build process.

6.2 Out of memory error

With this initial setup, the project did not even build locally, as we got a PMD out of memory error. We have encountered this issue before, but have never been able to find a definitive solution, not even with a teaching assistant. Since PMD's source code analysis was not a priority for this effort of integration, we decided to remove it from the pipeline.

6.3 Checkstyle error

The submissions microservice team had decided to make Checkstyle violations critical for their subproject. Since the submissions microservice contained numerous Checkstyle violations, the project still would not build, so we decided to turn of Checkstyle for the submissions microservice, as we did not feel compelled to correct all their style issues.

6.4 Broken endpoints

Now the project did compile, however when generating code from submission's specifications.yaml file, we got an error for the '/submissions/submissionId/edit'
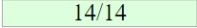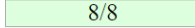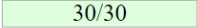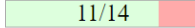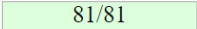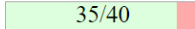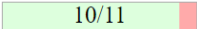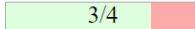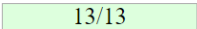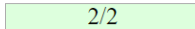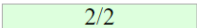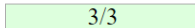
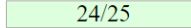| Name	Line Coverage		Mutation Coverage	
CommentService.java	100%	14/14	100%	8/8
PaperService.java	100%	30/30	79%	11/14
ReviewService.java	100%	81/81	88%	35/40
ReviewerPreferencesService.java	91%	10/11	75%	3/4
TrackPhaseService.java	100%	13/13	100%	2/2
UserService.java	100%	2/2	100%	3/3

Figure 11: Service classes after changes
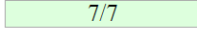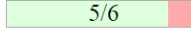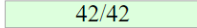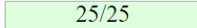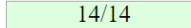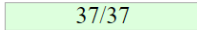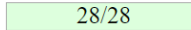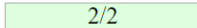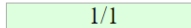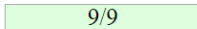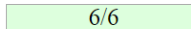
Name	Line Coverage		Mutation Coverage	
BaseValidator.java	100%	7/7	83%	5/6
ChainManager.java	100%	42/42	96%	24/25
DatabaseObjectValidator.java	100%	25/25	100%	14/14
ParameterValidator.java	100%	37/37	100%	28/28
SuccessfulState.java	100%	2/2	100%	1/1
UserValidator.java	100%	9/9	100%	6/6

Figure 12: Validator classes after changes

request, as the request body was not of the proper type. The submissions team has got this issues before and suggested we removed this endpoint from the specification, so we did.

Moreover, the 'tracks/id/deadline' endpoint of the user microservice caused a problem for the other microservices because though it returned the date in the correct format, it would include " in the beginning and end of the string. This caused an error both for our microservice and the submissions microservice. Since the review microservice only needs to call that endpoint, dealing with this issue for us only consisted of making a call to a utility method of the String class in Java. However, the submissions microservice makes calls to that endpoint in multiple requests, which resulted in us not being able to successfully test integration with the reviews microservice on retrieving track information from them.

6.5 Conflicting port numbers

One of the easier issues to solve was that we, the different teams, chose conflicting port numbers for our projects. We decided that submission should be at 'http://localhost:8081', users should be at 'http://localhost:8082' and review, our microservice, should be at 'http://localhost:8083'. Even though we agreed on this between teams, it did not go according to plan. Submissions took port 8082, we accidentally specified 8083 in one file and 8084 in another one and users microservice did not specify any port at all. We corrected this on our

team-5b 'integration' branch.

6.6   Removing MockMvc

Finally, all the OpenAPI specifications generate properly, all the microservices compile and all tests pass. However, we had not yet tried to remove any of the mock Model–view–controllers. When we did, we ran into a number of issues:

- URI set incorrectly: for some of the URIs, the string 'http://' had to be prepended to work properly.

- Not found errors: some of our tests where written such that they required certain elements to be in the other microservices' databases. This was not a problem while the responses where mocked, but for this integration step, it required adding calls to endpoints in our test to create new papers, users et cetera.

- Bad request errors: some of the request we had to make where not exactly in the format expected by the other microservices, so we had to correct those as well.

6.7   Missing endpoints

The three microservices fulfill distinct roles to result in one EasyConf system. Within this context, the review microservice was mostly a consumer of the other two microservices. We had to get a list of papers from the submissions microservice, had to get track information and verify roles using the users microservice et cetera. This was caught relatively early in the integration process, but it should of course have been caught while our teams were writing the API specifications. Because these omissions were caught so late in the process, there was little time for the teams to make proper endpoints and it became more difficult for all to adhere to clean code standards. This problem could easily have been prevented by earlier and ore thorough inspection of each other's work.

6.8   Differing standards

One of the positives in our design process was the standardization. While writing the specifications, our teams came together because we anticipated a problem of non standardized ids. Already at the start of the project we decided it was best we all use 32-bit integers as ids. The main reason was the ease of use in comparison to UUIDs. We discounted the limited size of 32 bits considering the small scale our microservices would be used at.

Still, not all standardization went correctly at first. For example, there are various deadlines in the microservices. These are of course stored as dates, but we did not specify the format for the dates in the OpenAPI specifications, so we had to agree on this later.

6.9 Result of integration

In conclusion, the integration of our microservices was unsuccessful due to the breadth of the issues discussed above, the main ones being out of memory errors, differing standards and broken, missing or not implemented endpoints. As the review microservice team, we had the relative advantage of mainly consuming endpoints of the other microservices, necessitating minimal changes for the other teams. Nevertheless, it remains challenging to consume other's endpoints and to manage the unexpected behaviours of a complex system.

We, our team and the teams of the other microservices, have done the best we can to make the integration a success, but because of a limited time window of the integration process and a small team, it ultimately fell short of its objectives.

7   Conclusion

This report has covered most of the relevant technical details of the review microservice as implemented by our team, 5b. The main difficulty for our team during this course has been the lack of communication and the ultimate withdrawal from the course of three of our team members. Despite all this, we hope to have demonstrated our newly acquired skills in the field of software engineering. We believe this course has given us valuable insight into the world of software development and aim to apply this further in future courses like the software project.