Final Hand-In

Thomas Jackson Terry pdh248
Taraburca Radu whx862
Matthieu Massardier cxs929
Mihai-Razvan Costescu wrl572

16 June 2023

Contents

1 Project plan **3**
- 1.1 Introduction . . . 3
- 1.2 Project Communication . . . 3
- 1.3 Roles . . . 3
- 1.4 Firsts ideas for integration . . . 4
- 1.5 Tools . . . 4
  - 1.5.1 Software development tools . . . 4
  - 1.5.2 Communication tools . . . 5

2 Requirements Elicitation **6**
- 2.1 Introduction . . . 6
- 2.2 Functional model . . . 6
- 2.3 Non functional requirements . . . 10
- 2.4 Quality attributes . . . 11
- 2.5 Quality attribute scenarios . . . 12
  - 2.5.1 Scenario 1: Usability . . . 12
  - 2.5.2 Scenario 2: Availability . . . 14
  - 2.5.3 Scenario 3: Testability . . . 14

3 Architecture Document **16**
- 3.1 Stakeholder/View Table . . . 16
- 3.2 Components & Connectors . . . 17
  - 3.2.1 Primary presentation . . . 17
  - 3.2.2 Element Catalog . . . 18
  - 3.2.3 Context Diagram . . . 20
  - 3.2.4 Variability Guide . . . 20
  - 3.2.5 Rationale . . . 20
- 3.3 Security . . . 21
  - 3.3.1 Primary presentation . . . 21
  - 3.3.2 Element Catalog . . . 21
  - 3.3.3 Context Diagram . . . 21
  - 3.3.4 Variability Guide . . . 21
  - 3.3.5 Rationale . . . 22

4 Analysis **23**
4.1 Analysis Model Description 23
4.2 Object Models 24
4.3 Dynamic Models 26

5 After project **28**
5.1 Extra Diagrams 29
5.2 Development phase 30
5.2.1 Version Control and Collaboration 30
5.2.2 Code Organization 31
5.2.3 Separation of Concerns 31
5.2.4 Error Handling and Data Validation 31
5.2.5 Code Readability and Documentation 32
5.2.6 What went wrong 32
5.2.7 Future works 32
5.3 Design patterns 33
5.4 Testing 34
5.4.1 Unit tests 34
5.4.2 Integration testing 35
5.4.3 System testing 35
5.4.4 Acceptance testing 35

A **36**

Chapter 1

Project plan

1.1 Introduction

Our project is to create a group-matchmaking system to automate the group creation process for the Software Engineering & Architecture course. Our client is Tijs Slaats, the professor of this course. Our aim is to create a system that allows potential clients to enter project ideas, and subsequently allows students to select their preferred topics and teammates. Then the system matches students according to their preferences and displays the resulting groups.

1.2 Project Communication

We plan to use a few methods of communication. We are using a Discord server to keep in touch from day to day and to handle unplanned or spontaneous communication. During the planning block, we also plan to hold a meeting each week on Thursday to check in on progress and make plans for the next week. Some weeks, this time will also include a meeting with our client for requirements elicitation or other questions. At first, these meetings will likely be in-person to facilitate brainstorming, but they may move to online later - perhaps during the implementation phase, when there will be less need for sketching that benefits from face-to-face meeting. Other communication types are in the following table.

1.3 Roles

We intend to have a somewhat flat role structure, where team members communicate with each other and do not have to report to a single manager. However, we will still have a role that is responsible for knowing the progress of the project as a whole, to better communicate with the client. We will designate one person to be responsible for writing emails to the customer and scheduling calendar

ACTIVITY	TYPE	FREQUENCY	PURPOSE
Project planning meeting	In person	Once	Outline client goals, establish budget and timeline, assign first milestone task
Daily stand-up	In person	Daily	Set daily goals, update project status
Sprint retrospective	In person	Once two weeks	Discussing what went well and what can be improved in a sprint
Sprint review	In person	Once two weeks	Team shows what was accomplished, while the stakeholders provide feedback
Review meeting	Online	As needed	Update the status of an task
Postmortem meeting	In person	End of project	Assess success and failures of project

Product Manager	Jackson (responsible for communicating with client)
Software Architect	All (everyone is participating in the design during the first block)
Software Developer	All (everyone will contribute to coding, with some specific intentions)
S. D. - Backend	Radu
S. D. - Frontend	Mihai
QA/Tester	All

events, to prevent overlapping. During client meetings, team members can speak up as they wish with questions and contributions. Our role assignments are still tentative, but some possibilities are as follows.

1.4 Firsts ideas for integration

Language : We will use Java (springboot) for the backend of the system, and Angular for the frontend.

Methodology : We will do SCRUM, implementing new functionalities each SCRUM, starting with the basics. We might modify some features of SCRUM to better fit our situation of not all being on the same schedule in the same building.

Models : We will use UML models and notation to describe the systems.

Application Domain : The operating environment of the system will be Windows.

1.5 Tools

1.5.1 Software development tools

For the development of the project, we will use the following tools:

- for the frontend:

  - Visual studio code
- for the backend:
  - Eclipse/Intellij
- for keeping and sharing the code:
  - Bitbucket
  - Git

1.5.2 Communication tools

- Discord
- Jira
- Overleaf
- App.diagrams.net
- Outlook

Chapter 2

Requirements Elicitation

2.1 Introduction

We read the initial description of the system provided by our client and used it for a brainstorming session. We came up with questions to clarify the initial requirements, and we also came up with some ideas for implementation; we then formed questions to determine if these ideas were acceptable and desirable. We then took these questions to an in-person interview with the client and also discussed further questions that came up during our conversation with him. We did not need to keep strictly to an interview format with fixed questions. From our results, we created diagrams to model the system for our own and the client's understanding. Additionally, since we ourselves had used the predecessor system to select our groups for the class, we had some idea about the important points to include in the system. Other elicitation methods, such as user observation, were not applicable in this case because we are implementing Greenfield engineering - creating a new system, not modifying one that users are already using.

2.2 Functional model

In natural language, the requirements of the system are as follows. We are to create a system that will help the course coordinator automate the creation of groups for the class. We should include a way for the coordinator to populate the system with a list of students, and to create project titles and descriptions. Other potential clients should also be able to add project descriptions. Then, students should be able to access the system in order to select their top-priority projects, or possibly rank all projects in order of preference. They should also be able to select up to 2 other students with whom they would like to be in a group. There need to be security measures in place to protect student privacy and confidentiality and also to prevent tampering with others' choices. Finally, the coordinator should be able to use the system to form the final groups. He should be able to change some parameters such as how heavily a

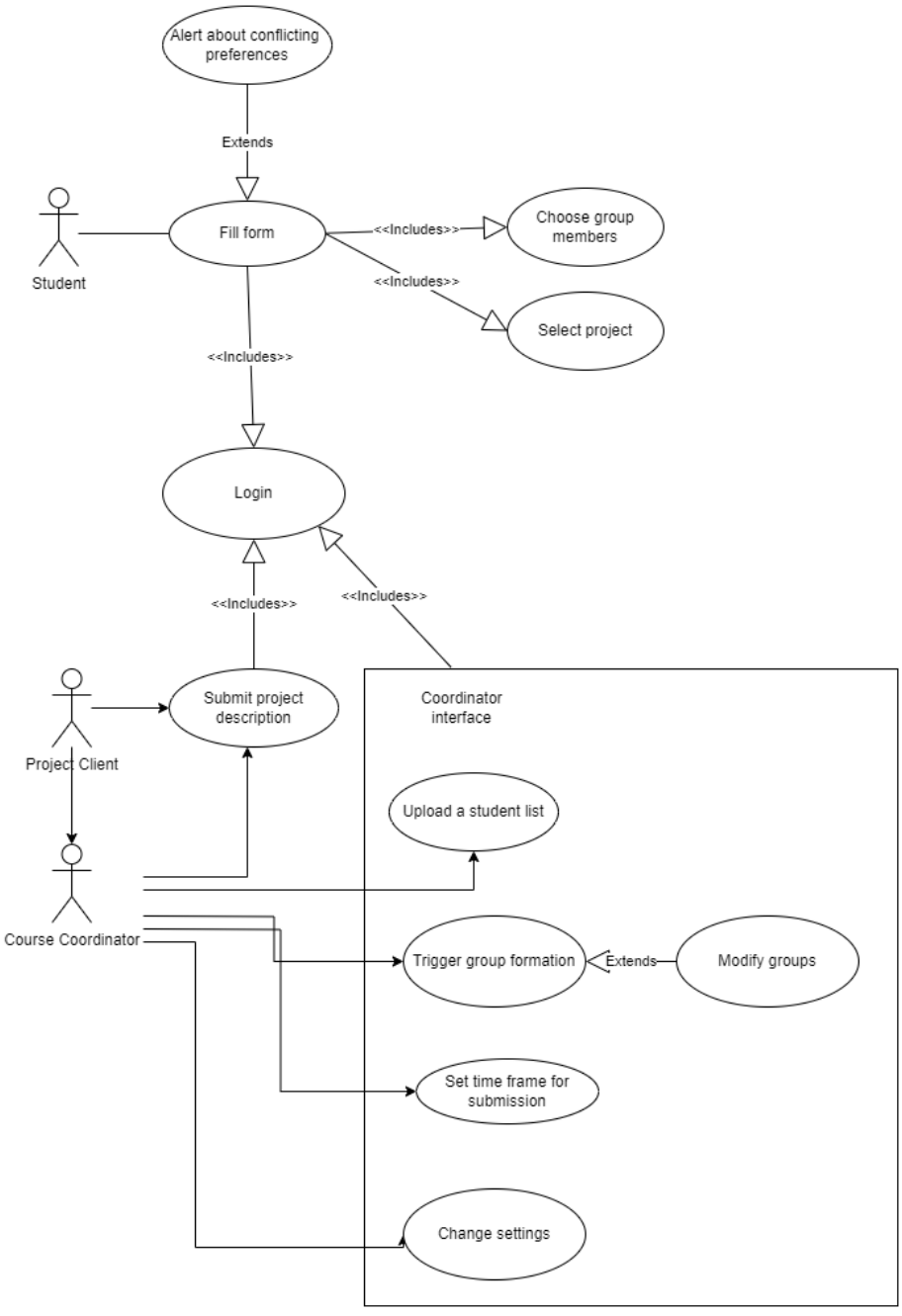

Figure 2.1: UML diagram defining the use cases

top choice is prioritized. He should also see some statistics about the generated groups. Once he is satisfied with the groups, he should be able to obtain them in a format convenient for use with Absalon. The UML diagram figure 2.1 explains it in a more normalized representation. Of note: the "Alert about conflicting preferences" case extends the "Fill Form" case because it is not a normal function, like "Choose Group Members" or "Select Project," that would need to be invoked every time. Instead, it is an extraordinary case that should only occur infrequently.

Use Case 1

Name: Fill form
Participating Actor: Student
Entry Condition: Student is part of the course
Exit Condition: Form submitted
Flow of Events:

1. (User) The student logs in
2. (System) System displays information about projects
3. (User) Student ranks projects
4. (User) Student selects group mates
5. (User) Student clicks submit
6. (System) System informs the student of successful form completion (if everything was filled out)
7. (System) The system stores the data about the student's preferences

Special Requirements: If the student has previously filled out the form, then it should not be displayed to the student blank; instead, it should display as filled out with their previous choices.

Use case 2

Name: Submit project description
Participating Actor: Project client
Entry Condition: The course coordinator has selected this user as a client for the course and has granted them access to the form
Exit Condition: Project is submitted
Flow of Events:

1. (User) The client logs in
2. (System) The system displays an empty form with fields for project name and project description
3. (User) The client types the name and description
4. (User) The client clicks submit
5. (System) The system informs the client if their submission was successful

Special Requirements: None

Use Case 3

Name: Trigger group formation
Participating Actor: Course coordinator
Entry Condition: Students filled the form and the deadline has passed
Exit Condition: The groups are formed and the coordinator is satisfied with the groups
Flow of Events:

1. (User) The coordinator logs in
2. (System) The system displays the generation parameters
3. (User) The coordinator checks, modifies and accepts the parameters and starts the generation
4. (System) The system computes the groups and shows them to the coordinator, along with some stats (for example, nb. of student having their 1st choice)
5. (User) The coordinator can move people around groups
6. (User) The coordinator submits the final group composition
7. (System) The system saves the groups and informs the professor

Special Requirements: Within a single instance of use, the flow of events can repeat (from displaying generation parameters until moving students in groups) as many times as the coordinator desires. Also, even after the final step, the use case can be re-initiated from the beginning.

2.3 Non functional requirements

User interface and human factors: It is important that the system be easy to use, and that specific training not be required for students. For instance, including reading project descriptions (but excluding the time a student takes to decide on a project), project selection should take no more than 5-10 minutes. The system does not need to actively prevent a user from making errors, as there are few errors a user could make - besides submitting an incomplete form, so the system should allow resubmission.

Documentation: Since the system will probably not need to be modified much, the documentation should be focused on explaining the system enough so that it can be maintained and bugs can be fixed. The documentation likely does not need to be longer than two pages.

Hardware considerations: Any type of hardware that can run a browser can be used for the application as it will be a Web Application.

Performance characteristics: As it comes to the performance of the application the user interface should have a fast response time so the user can go

though all the steps smoothly but after the data is submitted the time it takes for it to be processed and stored shouldn't be that important and it can have a slow response time as it will not be needed right away. It is unlikely that more than about 20 people will be using the system at a time, but the possibility for scalability up to around 200 (the max number of students in a KU class) would a nice extra.

Error handling and extreme conditions: The system should not be too badly affected if it receives an input error or undergoes extreme conditions. A small amount of downtime is permissible, but the system should be able to recover quickly.

System interfacing: Some input to the system - the student names - may come in a particular format. It will likely be the csv file that Absalon produces, so the system should accept this format as input. It should also output a similar format to be imported back into Absalon.

Quality issues: The system should have 99% uptime during the couple of weeks at a time that it is available. It is also important that the system durably retain its data, so that students do not have to be asked to re-enter their preferences.

System Modifications: After the completion of the project, there are few chances that changes will be needed, but if there is something that can be changed, it could be the improvement of performance or the removal of potential bugs that escaped during production.

Physical Environment: Most likely the system will be a cloud-based system, so its physical location will not be a problem. We will use the university's servers to host the tool components.

Security Issues: The system needs to store student information securely, as it contains student names and the courses they are taking. It needs to comply with, for example, GDPR regulations.

Resources and Management Issues: After initial handover of the project to the client, they will likely be the one to maintain it in the future. The system is responsible for data backup, once an hour it will make a backup, and it will keep the newest file made in the last 24 hours, and once a week it will clean the files older than one week to save only one backup for the respective week.

2.4 Quality attributes

- **Usability** - The usability is high priority both in our view and in the client's view. The client values having a system that is easy for students to pick up and use without needing much help. It would be a significant waste of time for the client to have to constantly respond to questions and troubleshoot students' usage of what should be a quick and simple submission at the very start of the course. We have an example of an easily-usable system in our own experience selecting groups for the course, so we understand its value. The easier the system is to use, the more students will bother to use it, allowing them to be assigned projects that

they will prefer.

- **Interoperability** - Interoperability is considered here as a high-priority quality attribute. Although we will probably only use a few external services, these few connections should work perfectly. KU login is essential for to ensure that users cannot alter anyone's selections but their own. And a significant motivation of the project is the ease of compatibility with Absalon, so the data input and output should be in a convenient format for that application.

- **Security** - Security is a medium-priority quality attribute for this project. It is always at least somewhat important to consider security in just about any online system, to avoid overlooking obvious security issues. But this system will not interact with much confidential student information - their emails, at most - and it will also be able to rely on the university's authentication system to handle some security concerns.

- **Testability** - Testability is a medium-priority attribute for this project. Having a testable system makes it easier to demonstrate to the client that the system behaves exactly as they desire, without any issues. It also eases the development process by making it easier to fix bugs, so it is mutually-desirable to moderately prioritize the system.

- **Modifiability** - Modifiability is an attribute of middle importance. The client mentioned that he would like a system that can continue to be used in the future and adapted to potentially-changing circumstances, not one that is tied directly to the current version of the course, so the user-side should be fairly flexible. However, the client does not foresee needing to significantly modify the code itself. So, modifiability is mostly only relevant for ensuring that the code is maintainable in case future bugs arise and need to be fixed.

- **Availability** - We have determined that the level of availability we require for our project is achievable without needing to prioritize it above other aspects of the project. Therefore, we will be focusing our efforts on other areas to ensure a successful outcome.

- **Performance** - While performance is always a consideration in any project, our team's focus is not on optimizing for speed or efficiency, but rather on other aspects of the project that we deem more important.

2.5 Quality attribute scenarios

2.5.1 Scenario 1: Usability

Source: A first-time user (Student)
Stimulus: Attempt to create a new group, with a particular group size
Artifact: User interface(Group creation interface)

Environment: Runtime
Response: Adds the group preferences to the database.
Response Measure: Time it takes between arriving on the form and clicking "submit;" time it takes to correct a mistake; time it takes for the system to respond that it has saved a group.
Description: We chose the given source and stimulus because a student selecting their preferences is the most common use case of the system, so focusing on it would bring the most benefit. The artifact for a usability scenario is frequently the user interface during runtime, since the interface is a major component of user experience. The response describes the best outcome of the interaction, and the response measures gives an idea of how easy we want it to be to use the system.
Tactics: The "Cancel" tactic is not likely to be useful for this scenario. The system does not perform any steps while the user is in the process of filling out the form, so any cancellations would simply take the form of the user deleting what they have entered or clearing selections. The system only acts once the user clicks submit, and afterward the behavior is covered by the "Undo" tactic.

The "Pause/Resume" tactic is mostly not relevant for this scenario. We want to keep in mind that a user may take a break in the middle of filling out a form to do other things when we consider the time it takes them to make their selection. But we do not need to implement a pause functionality in the system because it will not be consuming many resources during the form-filling process.

The "Undo" tactic is quite useful in this scenario. It helps to meet the goal of a student being to immediately correct a mistake. If the user clicks submit accidentally (or later decides they want to change their preferences), then it makes it easier for the user if their prior preferences are pre-filled into the form. That way, they can make only the changes they need, making it faster to correct mistakes. This can increase the speed of usages beyond the first.

The "Aggregate" tactic might be slightly useful in this scenario. There are unlikely to be very many project topics for the user to interact with, so there would not be much drudgery for the user. However, perhaps the projects could be given a default "no preference" ranking so that if the user only wants to select group members, they also do not have to indicate "no preference" for every topic.

The "Maintain Task Model" tactic probably isn't relevant in this situation - there is usually not much prompting or user assistance needed for filling out a form.

The "Maintain User Model" tactic could be useful in this case. The system could anticipate a possible source of user error - for example, leaving a field blank or not indicating a preference for a topic. Then, when the user clicks submit, the system could display a message asking if the user is sure of their choices and indicating the missing information. That way, the system can help users who have made a mistake.

The "Maintain System Model" tactic is unlikely to be useful in this case. There is not much system behavior that the system needs to reason about and present to a student who is only filling out the form.

2.5.2 Scenario 2: Availability

Source: A user form submission
Stimulus: A database or server crash
Artifact: Tool UI and persistent storage
Environment: Runtime
Response: Error message (while keeping data) and system recovery to normal operation
Response Measure: Time to notify the user; time to restore the system; quantity of these crashes during a 24-hour period; amount of lost inputs
Description: We chose this quality attribute and scenario because it is important to the client that user data not be lost. So, we want to consider how to inform users promptly if an error has occurred, and how to recover from faults so that fewer users will encounter such an error.
Tactics:

The "Ping/Echo" tactic would be a way to constantly monitor the system's health, so that if the system is down, the user can be notified more quickly, instead of waiting for a user to submit a request and only checking the system's health then.

We could use the "Timestamp" tactic to keep track of a user's form submission and determine if it was before or after a crash, in order to figure out if the information was saved or needs to be resubmitted.

The "Rollback" tactic would help to recover from a crash. For example, using the ARIES system of Undos and Redos might be a way to preserve user submissions that were in progress at the time of the crash.

"Transactions" would be important to ensure that user information is treated correctly. Either all of a user's submission is saved, or none of it is. We want to avoid the scenario where the system informs a user that their submission was lost, but then some of the user's choices are saved anyway - or the even worse situation in which a user thinks their preferences were saved but some portion was lost.

2.5.3 Scenario 3: Testability

Source: Integration testers, using manual tests and testing tools
Stimulus: The execution of tests regarding the integration of the user form with the system's database
Artifact: The full system pipeline, from interface to database
Development time
Response: Execute test suite and capture results
Response Measure: System coverage, length of time to perform tests, ease of performing tests
Description: We chose this source and stimulus for continuity with the Usability scenario - after a user submits the form, we want to test to ensure that it is checked and stored correctly. Therefore, the artifact involves dataflow through the system, as well as how that data is processed by the backend. Development

time is a relatively early stage, so we can more easily make changes in response to tests during this time. The response measure attempts to capture how effective the tests are, as well as how easily and quickly they can be performed, since easier tests usually result in more testing, which in turn improves the system.
Tactics:

We would implement the "Specialized Interfaces" tactic in this scenario. We would certainly not want to have to fill out dozens of forms in order to get data for the backend system to operate on; instead, we would use a specialized interface that allows us to quickly feed a large amount of simulated student preferences into the system. The input format would match the format coming from the frontend, but the data would be more quickly creatable.

We would definitely use the "Record/Playback" tactic in this scenario. We would want to record and store the input in order to later retrieve any input that causes a fault. Then we could repeatedly feed in permutations that input to determine if the fault is reproducible, and to try to find a minimum input that causes the fault, which makes it easier to locate the source of a bug.

The "Localized State Storage" tactic probably is not relevant for this situation, as we would want to feed in different data for each test to increase the system coverage.

The "Abstract Data Sources" tactic is useful, like the Specialized Interface tactic. In order to test large volumes of data that would be too much to type manually, we would want to use abstract data sources to exercise the interface and backend logic as much as possible.

The "Sandbox" tactic is of use to us in this scenario. Even though this testing scenario deals with tests performed during development, and not during live operation (so we would not be forced to distinguish our test data from live data), it is still good practice to keep separate environments. We would use a sandbox for our testing to make it less likely that we pollute the eventual live data storage with accidental leftover testing data.

The "Executable Assertions" tactic is useful in this scenario. Since these tests deal with information flow through a pipeline, it is helpful to include manual checks at each step in order to catch errors as soon as they occur. Catching them early helps track down the source of bugs, and in the final system, it could permit the system to warn the user right away if they have entered data that is somehow faulty.

The "Limit Structural Complexity" tactic is a generally good coding practice, but it does not have specific applications in this scenario.

For the "Limit Non-determinism" tactic, we need to take into account that the system might need to have a certain degree of parallelism to satisfy requests from multiple users; therefore, we should be careful about how we implement it to avoid non-determinism that might cause bugs that avoid detection during testing.

Chapter 3

Architecture Document

3.1 Stakeholder/View Table

	Modules/Decomposition	Uses	C&C/Component	Security
Users	N	H	N	O
Acquirers	N(O)*	H	O	H
Developers	H	M	H	H
Maintainers	H	O	H	M

Table 3.1: N: None. O: Overview only. M: Moderate detail. H: High detail.
* In our case, the client will probably be interested in this aspect, though usually they would not be

We are making one view combining the developers' and maintainers' view of modules/decomposition with their view of components. And we have another view that concerns security as seen by all invested stakeholders.

3.2 Components & Connectors

3.2.1 Primary presentation

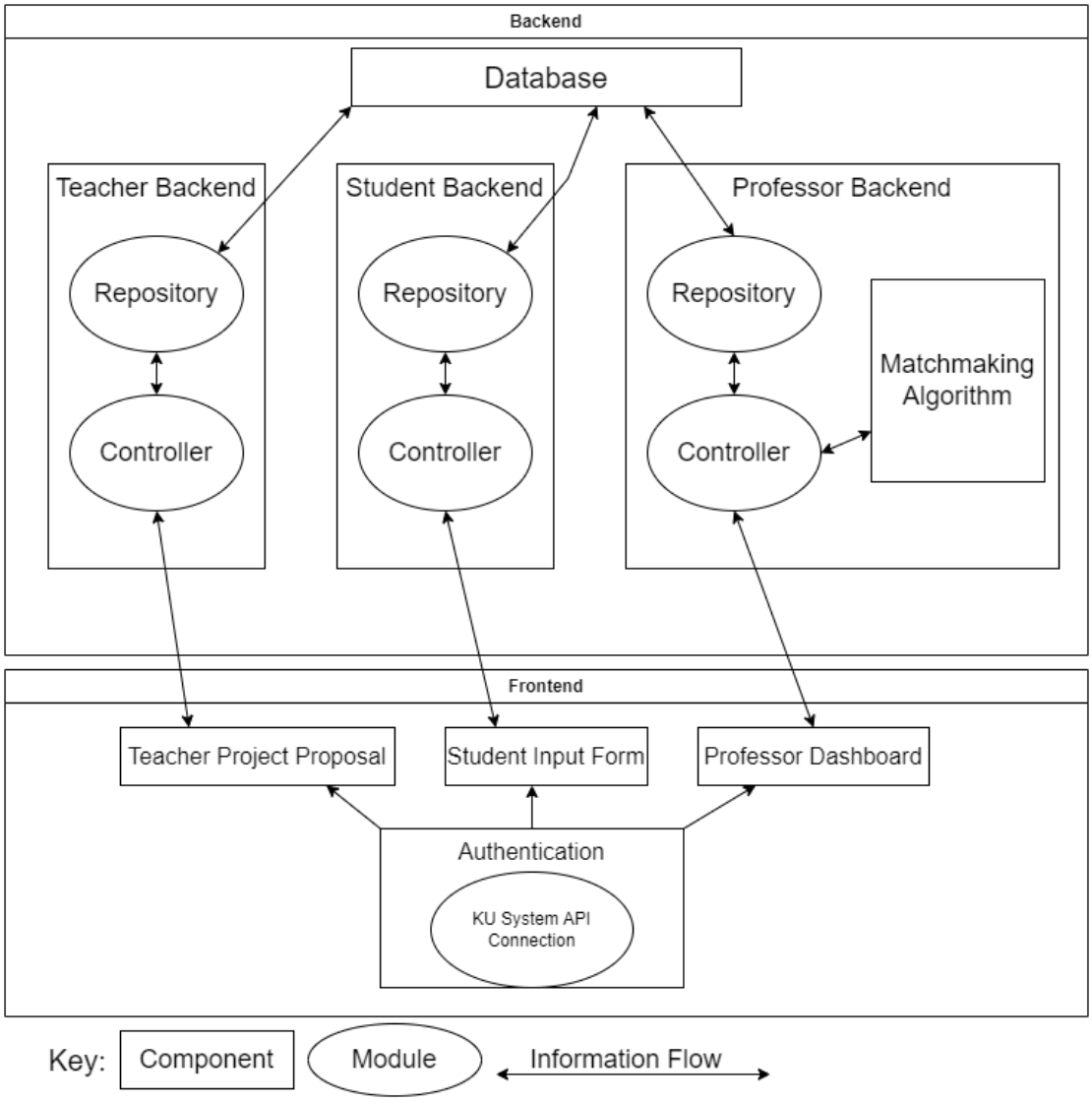

Figure 3.1: Components & Connectors view

3.2.2 Element Catalog

On the backend we have :

- Database - This component is responsible for storing information such as the project descriptions, the student list, and the saved project groups. It can only be read from or written to by the repository modules, to better separate the flow of information in the system.
- Repository - This is a module within the backend components. The repository communicates with the database by retrieving and sending data. In addition, it communicates with the controller by taking data from the controller and returning data from the database to the controller.
- Controller - This is a module within the backend components. The controller communicates with the repository by sending data taken from the frontend. It also has the role of taking the data provided by the repository and sending them as a response to the frontend.
- Student backend - This component will handle the data manipulation and logic for the student frontend. It will have a repository and a controller. It will pre-fill the forms and check for the data correctness.
- Teacher backend - The teacher backend is similar to the student backend in the modules it contains, but it handles the projects proposed by potential clients instead.
- Professor backend - This component will handle most of the logic and data-manipulation associated with the professor uses. It has a repository and a controller, as every other backend, but will also include the matchmaking algorithm.
- Matchmaking Algorithm - This sub component is part of the professor backend. It will discuss with its controller but is in an other component so that we can modify and switch it independently from the rest of the logic.

And on the front end we have:

- Teacher project proposal - This component should render the template that uses a form so that a teacher can write his project proposal. It communicates with the teacher backend controller through requests.
- Student input form - This component should render the template that uses a form so that the student can write his preferences in terms of colleagues and projects. It communicates with the student backend controller through requests.

- Professor dashboard - This component should render the template that acts like an overview and will also have a form so that the professor can decide on the projects or add more. It communicates with the professor backend controller through requests.

- Authentication - This component of the system is responsible for verifying student login, to ensure the security of the system. The primary organizational module it contains is the KU System API Connector. It is connected to the 3 types of frontend form display, such that once a user logs in, they are directed to their relevant form.

- KU System API Connector - This module describes the functionality that will interface with KU's login system. For example, it might redirect a user to the general KU login website, and then receive a token and allow the user to proceed to their relevant frontend.

In terms of relations, the relation depicted via arrows is information flow. Each portion of the system only communicates with other portions to which it is connected by an arrow. Each arrow indicates a different kind of information, as described for each element. Also, some elements are organized via a *part-of* relationship - some modules are part of components to indicate internal organization in that component, and one component contains an important part as a sub-component (the matchmaking algorithm).

3.2.3 Context Diagram

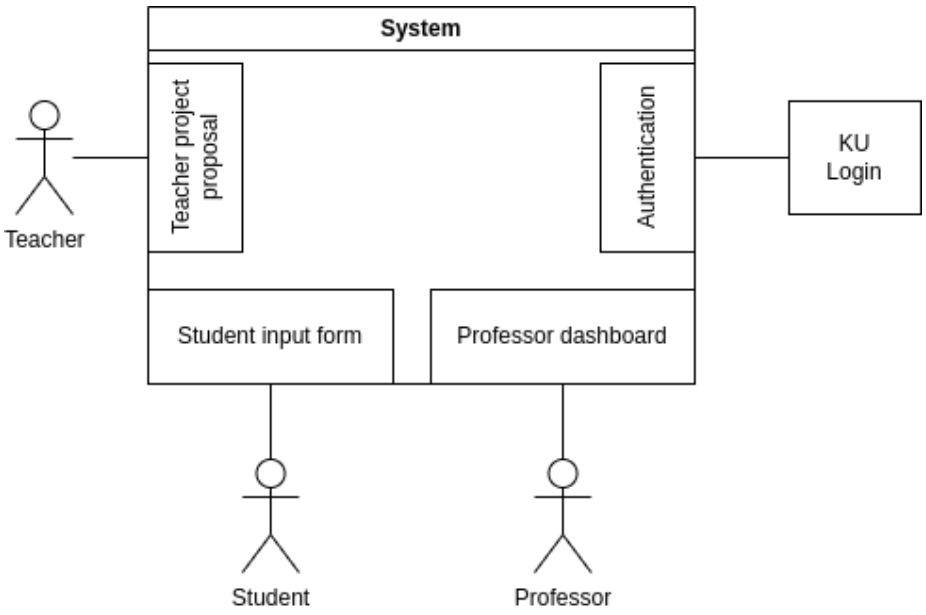

Figure 3.2: Context diagram for the Components & Connectors view (3.1)

3.2.4 Variability Guide

The authentication system is a variation point. If the KU login system changes or we don't get the right to use it, then our KU System API Connection will need to change along with it or change service. The matchmaking algorithm is another variation point - depending on which parameters we use and what we prioritize, the algorithm could vary significantly. The database is a variation point - we need to choose what kind to implement, such as SQL or not SQL. In case it happens, the repositories should then change along.

3.2.5 Rationale

Similar to the Model-View-Controller pattern described in the Architecture textbook, we have split our system into a layered architecture, in our case consisting of a frontend component, a controller module, and a repository module. The repository is the method to access the application data by reading from the database. The frontend is responsible for what is shown to the user, and the controller mediates between the two. This makes it easier to alter one portion of the system without having to change the entire system.

We decided to make the Matchmaking Algorithm its own module because of

the likely complexity of implementing it. As long as we define a standard input and output format for it, we can make changes to the interior of the component without worrying about how it will affect the rest of the system. This prevents the complexity from increasing even further.

3.3 Security

3.3.1 Primary presentation

- Login - We should be able to make sure that the person accessing the data is authorized. Moreover, a student shouldn't be able to impersonate the professor or a teacher. This operation should also be secured enough so that nobody could create a token impersonating the KU servers.
- Data storage should (ideally) rely on KU's infrastructure; this would tie the security of student names to KU's existing security, which can already be assumed to be strong. It avoids the issue of needing to rely on a host outside of the university to store information about students of the university, which could be a GDPR issue.
- The transmission must be done safely. The upload feature represents a big risk for the application, so some security protocols and data verification systems must be imposed. In addition, the data must be transmitted safely, so access tokens will be used in the data transfer to validate access to the backend.

3.3.2 Element Catalog

The sensitive points are the login, the data storage and the transmission.

3.3.3 Context Diagram

Every security concerned points are on the figure 3.1 : login correspond to the Authentication module, transmission to the arrows representing data streams, and data storage is a property of the database.

3.3.4 Variability Guide

The degree of protection that the system needs to provide against intentionally malicious attacks is variable. Although precautions should certainly be taken, an attacker could not do much damage by targeting the system - student project preferences are not the most critical information.

3.3.5 Rationale

For the safe transmission of data, we will use the mechanism called Token Authentication. Thus, in order for the requests to be completed, JWTs will be used.

Chapter 4

Analysis

4.1 Analysis Model Description

To inform our model creation, we used our previous documents to get an overall idea of what the system should be and do, and then translated these ideas into formal UML notation. We also drew from client meetings, including a meeting to determine what the minimum viable product should be. The minimum viable product is approximately as follows: The professor should be able to easily enter the student list into the system. Ideally, other teachers could enter their own project ideas, but at minimum, the professor will do so for them. Then, the students are given access to a form in which they are able to rank the projects (at minimum, they should be able to pick their top 3, but potentially they could rank all the projects). They also select their desired group members. At some later time, the professor should be able to initiate the group formation, using some algorithm to satisfy students' requests as much as possible. Extra sophistication in the algorithm, such as being able to configure the parameters by which it forms groups, are beyond the minimum product. Finally, the professor receives the group lists in a format that is convenient to use with Absalon.

To make the object diagrams, we drew from a brainstorming session in which we specified the organization of the system and how various data should be separated into objects. Then, to make the sequence diagrams, we used terminology from the object diagram, combined with the sample user interactions from the earlier use case diagrams. For the object diagrams, we focused on the data storage and flow through the system, such as tracking certain identifiers like user ID in all the locations they are relevant. We included all the classes we could think of, even those beyond the minimum viable product. For the sequence diagram, we focused on two significant use cases: a student interacting with the system to choose their preferences, and a professor interacting with the system to form the final groups. These are the most important use cases as part of the minimum viable product.

4.2 Object Models

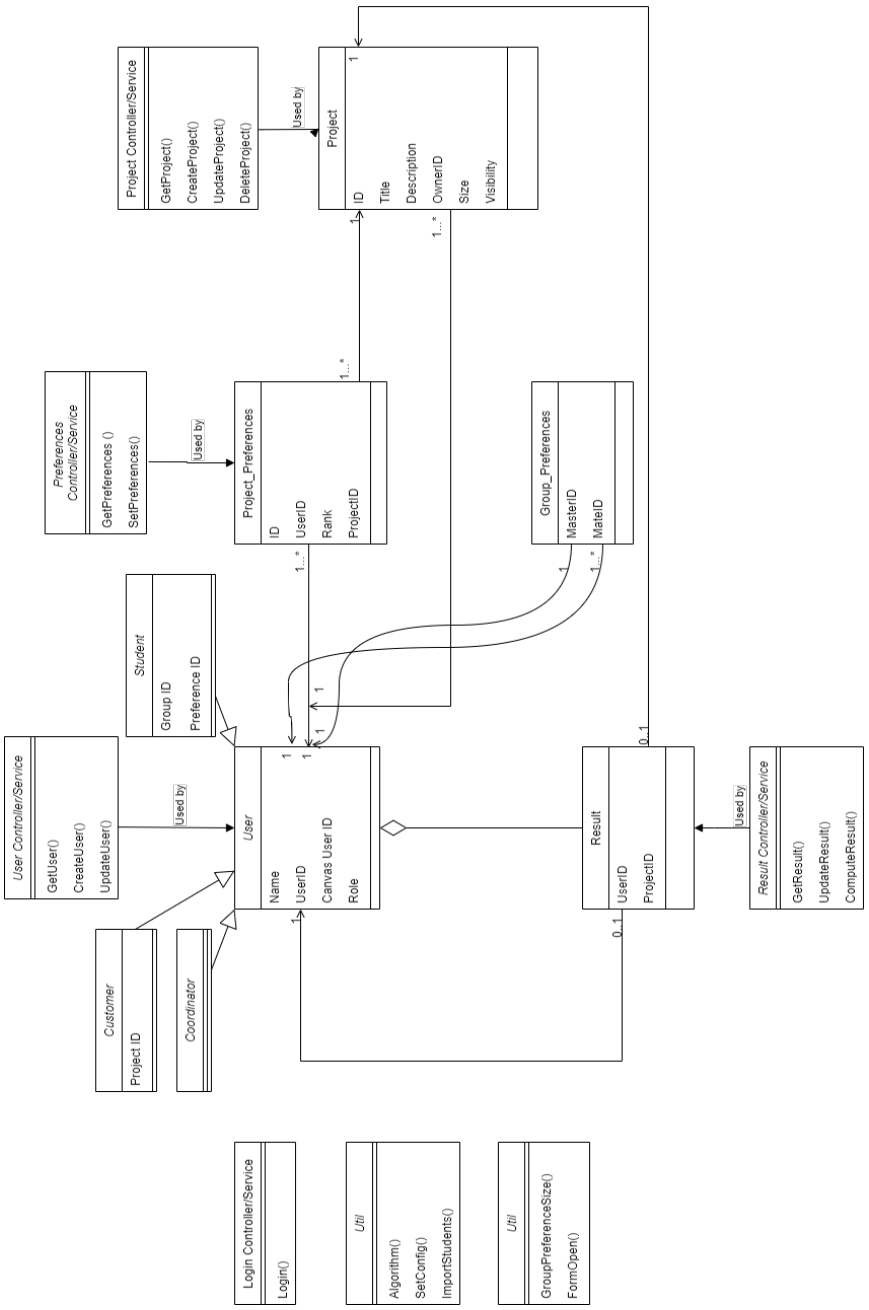

Figure 4.1: Class diagram for the backend

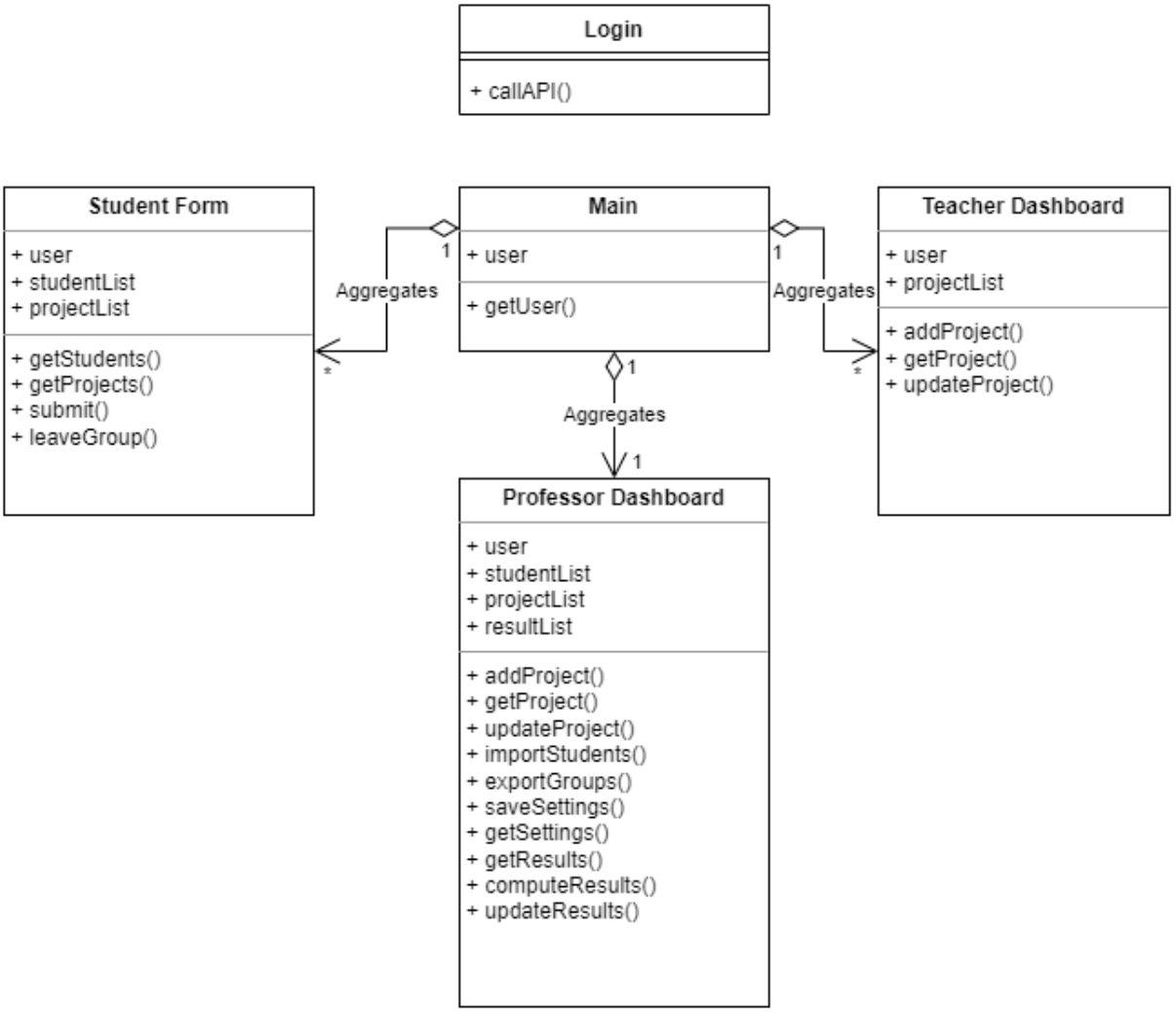

Figure 4.2: Class diagram for the frontend

4.3 Dynamic Models

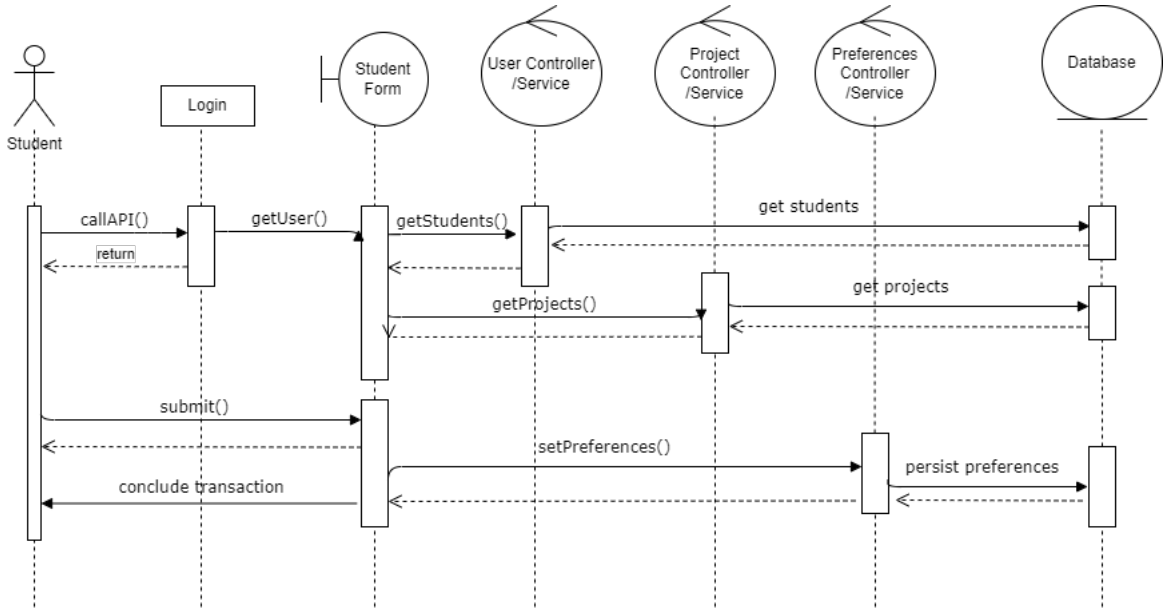

Figure 4.3: Sequence diagram for student interaction with the system

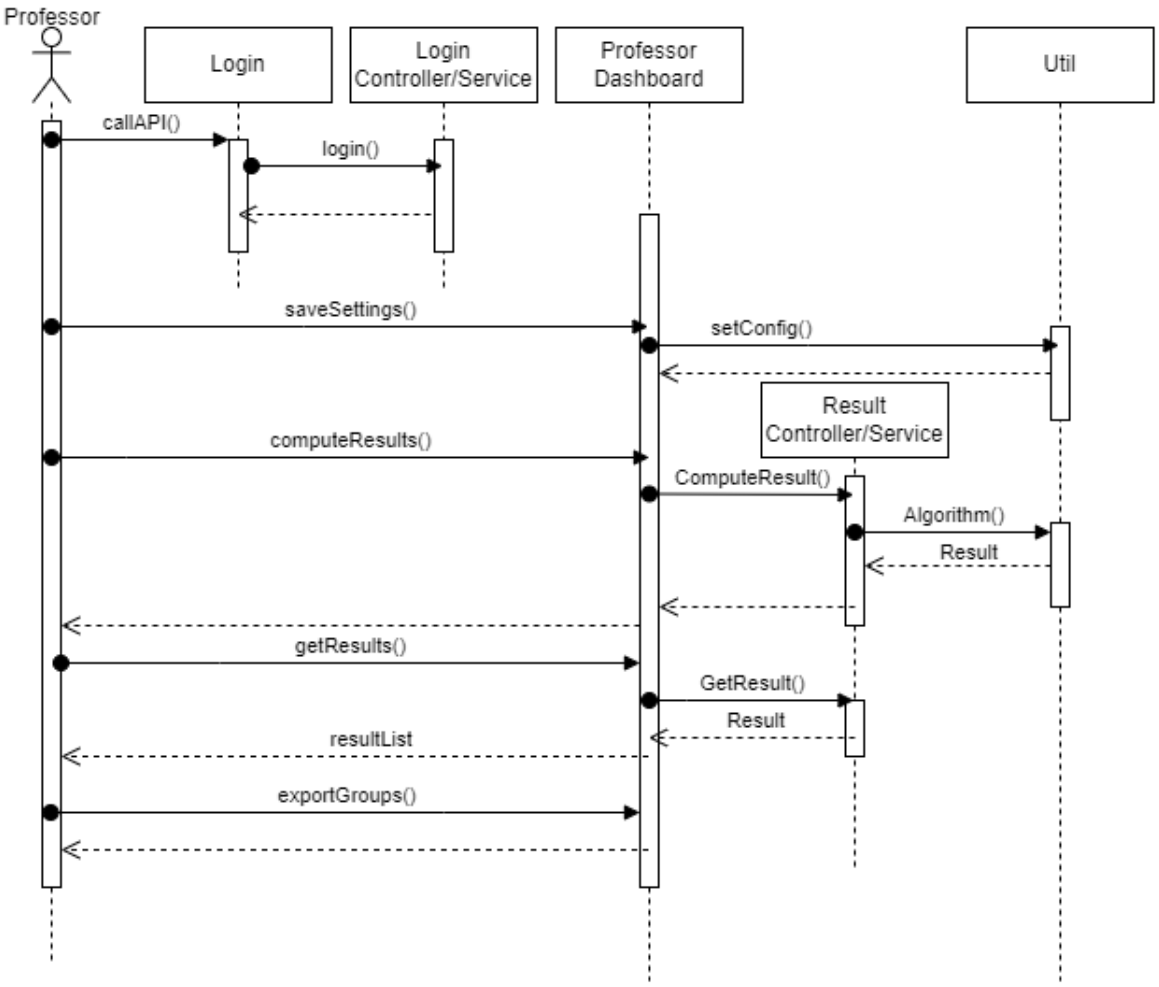

Figure 4.4: Sequence diagram for professor interaction with the system

Chapter 5

After project

5.1 Extra Diagrams

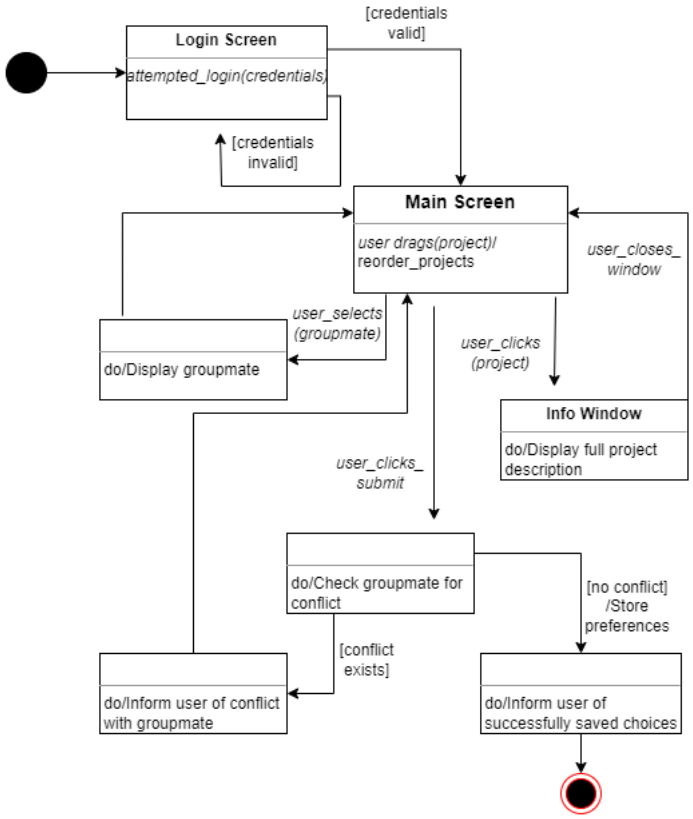

Figure 5.1: Statechart Diagram for the flow of the student user interface

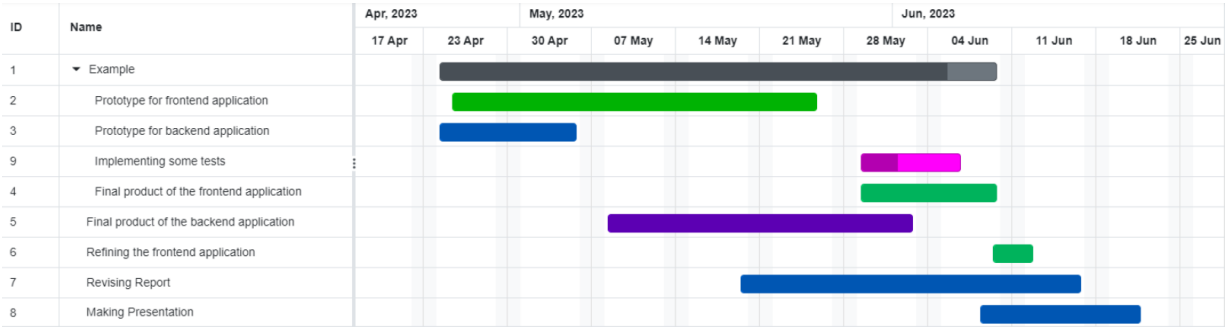

Figure 5.2: Hypothetical Gantt chart

Figure 5.2 shows an example project plan diagram. We did not make a concrete diagram before the project, so this example is of course influenced by our knowledge of the timeline at the end of the project. Making a concrete plan like this beforehand could have been helpful as a way to stay more organized and make time frames clearer to everybody.

5.2 Development phase

During the development phase, several methods were approached to ensure a smooth and efficient development process for the applications. Both applications developed in parallel and were created according to the needs of the users. The frontend was initially created on mocked data, while the backend was being developed. As the backend developed and new endpoints were added, the mocked data was exchanged with data brought by calls to the backend. The backend was built gradually starting from the entities and ending with the controllers. In the last phases of the backend, the grouping algorithm and security were added.

5.2.1 Version Control and Collaboration

Because the task of developing two applications in parallel is not an easy one and because there were several people who worked on these applications and because of the desire to have a checkpoint within the application, a versioning system was needed. I made a project in which both applications were added. Each person worked on his branch, and when a task was completed, a pull request was made to bring all the news to the master. Most of the commits were made with the name of the task from jira plus a small description of what was added, in order to have evidence of how much work was done on a certain component of the applications and to estimate how much more work is needed on that component. We used the integrated system between jira and bitbucket,

which shows for a task how many commits were made and which commits were made to keep this record easier.

5.2.2 Code Organization

The frontend code follows a modular architecture, with separate components, services and modules for different features and functionality. This modular approach promotes code reuse, separation of concerns, and easier maintenance. Each component and service has a specific role and responsibility, making the code base more organized and manageable.

The backend code follows a logical structure, with related classes and methods grouped together. The "Name-"Controller classes contain methods related to the "name" tasks, while the "Name"-ServiceImpl class implements the business logic for the "name"-related operations, such as UserServiceImpl which contains the user-related operations. This organization improves code readability and maintainability because it is easier to locate and modify specific functionality.

5.2.3 Separation of Concerns

The backend code follows the Model-View-Controller pattern, which promotes a clear separation of concerns. The Name-Controller class acts as a controller, responsible for handling HTTP requests and coordinating interactions between the model and the view. The model is represented by the "Name-"RequestDto and "Name-"ResponseDto classes, which encapsulate the data and transfer it between the controller and the view.

The frontend code follows the separation of concerns principle, keeping different concerns isolated within their respective components and services. For example, data retrieval and manipulation logic are separated into dedicated services (StudentServiceService and ProjectServiceService), while form management is handled by components such as ProfessorFormComponent. This separation facilitates code understanding, reusability, and testability.

5.2.4 Error Handling and Data Validation

The code incorporates error handling and data validation techniques.

For the backend, custom exceptions were created to correctly highlight the source of the error, the exceptions were handled in try catch but also by default a global exception handler. Besides that, in order to validate the data that is received by an endpoint, we put validators on the request dto fields. And for the endpoint that receives a csv, a file type validation was used, to accept only csv files.

For the frontend the MatSnackBar module is used to display error messages to the user when API requests fail or encounter errors. Form validation is implemented using Angular's built-in form validation mechanisms.

All these techniques used both on the backend and on the frontend aim to guide the user and ensure that it is user friendly.

5.2.5 Code Readability and Documentation

The applications are accompanied by many comments, almost every method has a java-doc block, the more complicated methods that require a more detailed understanding have comments between the lines of code. At the same time, the applications are accompanied by a text file that explains how to compile them and what settings must be made for them to work.

In order for the application code to be easy to read, all naming conventions for the used programming languages and proper indentation have been respected.

5.2.6 What went wrong

Within the project, there were things that did not go well, such as bugs, but in addition to the bugs that appeared along the way and that were resolved, some unforeseen situations appeared. One of these situations was the appearance of an extra security layer over the one from Canvas. After a lot of research, we discovered how to make calls to the API from Canvas and bring data from Absalon using the access token we generated. But everything we could do was limited using the access token, so we would have needed a developer token to be able to access more endpoints from the api. Besides that, I discovered that we cannot use the login endpoint from Canvas because that was only a part of the authentication system from Absalon, the other part was represented by Single sign-on (SSO), a rather complex authentication scheme for which we would have needed a developer key. Therefore, because the difficulty of using the Absalon API has increased, we had to stop using the API and create our own solution for authentication. Our approach was to use the institutional email as the username and password to be generated within the application, after which, after opening the project registration form, these data should be sent to the students' emails using an email service.

5.2.7 Future works

Although the project is in a functional state that satisfies the minimum viable product, it can be improved in many other ways. For the student form, improvements can be made on the frontend as well as on the backend, such as the possibility of deleting preferences, validating pair changes, notification of pair changes by email. For the part that is responsible for proposing projects, a richer interface can be made with the possibility to manage the proposed projects better, as well as to edit the projects once proposed, or to add more contributors to the project. For the most important part of the application, more precisely the "professor form", improvements can be made such as manually adding a student, adding projects from a csv, exporting the final groups as

a csv, editing the final groups, but also upgrading the matchmaking algorithm, in a more efficient and less biased one.

5.3 Design patterns

In the backend application, we used several design patterns, some of which are easier to recognize, while others are more difficult to identify because they are combined in the code. These design patterns helped us to modularize the code, and to make it easy to maintain and modify. These are the design patterns we use in the backend:

- Model-View-Controller (MVC) Pattern: The application is based on the MVC architectural model, where the "EntityName"Controller classes act as controllers, handling requests and returning responses. Request mappings and endpoint methods are defined in the controller class, while the service layer contains the business logic.
- Dependency Injection Pattern: Using the @Autowired annotation demonstrates the application of the Dependency Injection pattern. Building on Spring's dependency injection mechanism, dependencies such as UserRepository and AdminService are injected into classes such as AdminController and UserServiceImpl, promoting loose coupling and modularity.
- Builder Pattern: Although this pattern is not explicitly used, the BeanUtils.copyProperties() method from the "ServiceImpl" classes suggests that it is present in the code. We used this pattern to copy property values from one object to another, making it easy to build DTOs or entities with different property values.
- Repository Pattern: Within the application, numerous repositories were used, such as "UserRepository" and "ConfigurationRepositor", which follow the repository pattern. These repositories provide an abstraction for data access and encapsulate the logic for interacting with the data source.
- Strategy Pattern: Within the application, more precisely for security, I used the strategy model in the "SecurityConfig" class to configure the authentication and authorization strategies. The "authenticationProvider()" method defines the authentication strategy, and the "filterChain()" method configures the authorization strategy based on request matching.

Unlike the backend, where the design patterns are more or less obvious, in the frontend application, these design patterns are not so obvious. However, we can clearly distinguish these designs:

- Observer Pattern: The use of Observable and Subscription suggests the presence of the Observer pattern. Observables are used to represent asynchronous data streams, and components subscribe to these observables to receive updates when new data is available.

- Dependency Injection Pattern: Dependency injection is used quite often in the project card, as seen in the "ProfessorFormComponent" constructor. Dependencies such as "MatDialog", "HttpClient", "StudentServiceService", "ProjectServiceService", "FormBuilder", and "SettingsServiceService" are injected into the component, promoting loose coupling and easier testing.

- Singleton: Services, such as StudentServiceService, ProjectServiceService, etc. are usually implemented as singletons by default. They provide a centralized location for shared functionality between different components and data management.

5.4 Testing

Some pieces of code from the back-end are tested via automated tests. All these tests have been designed in a traditional fashion : they have been written after the code they test. As a matter of fact, a test driven development could have been executed only if we had planned that before, and traditional testing seemed better than no tests at all.

The coverage and test cases are however limited due to time and knowledge constrains. The test writing is in fact more difficult than planned and learning how to use the testing framework took most of the time allocated to this task.

The entire system has also been tested in manual tests, but this was way less structured.

5.4.1 Unit tests

Most of the services from the back-end has some kind of unit testing. The test coverage is however limited to the services due to the constrains mentioned over. It could have been interesting to unit test also the repositories and controller (controllers have been partially integration tested however).

The tests here are automated, as it was the best solution considering the amount of tests. We test following a dynamic analysis (considering basic static tests has been done while programming) with a somewhat of a grey box testing : some important calls are checked (for instance we check that we create as many random password as there is users) but we mostly test for result (for instance we don't check how the service decided that a csv file is valid but simply that it throws an error when it detects an invalid one). For these tests, we used a lot of mocked objects as the framework (bootstrap) really help with that, and the documentation advocate for such testing methods.

The front-end however hasn't been unit tested, because of the code base size, allowing to test directly the integration, with continuous integration. Moreover, learning yet an other unit test framework was too much work to get it done on time.

5.4.2 Integration testing

The few integration tests we've written concern the controllers all the way up to the routing and security layers of the back-end. These tests have been done with the framework as it provides the driver, objects and annotations allowing to tests these within a few lines. As a result, we used, as the driver helps to do, the big-bang approach. However, a lack of knowledge of these utilities made it difficult to implement test on all the controllers, and the security layer isn't correctly tested (integration tests fail while the full system actually work).

All these tests are made using mocked services to constrain the tested layers.

It could be interesting to do more extensive integration testing (for instance testing as deep as the repository), but the current knowledge of the framework and failure to correctly write the current test designs made it difficult.

We however did continuous integration to limit the impact of the failure to write these tests.

5.4.3 System testing

These tests have been realized in a very empirical fashion : we start the service and test the capability of the software to achieve what we need. This type of test is applicable to this project thanks to its relative small size. It also acts to some extent as a integration test of the whole stack that we did not test previously. However, this way we can only test the functional part, and tracking bugs is more difficult than if we were to have a full testing routine implemented.

The very few performance tests are also limited and still empirical.

5.4.4 Acceptance testing

We couldn't achieve a state of the art acceptance test. However, we did demos with the client to be as close as possible to such tests. This has been done as often as possible (at each meetings) to make sure that the product was fitting its requirements, and that the development was evolving in the right direction.

Appendix A

Interview Questions

- What should the balance of team members be? How many singles, doubles, and triples?
- Are there minimum and maximum team sizes? Is it ever OK to break these?
- Is it more important to prioritize team size or a student's desired project?
- Should we let students rank all the projects, or only pick their top X? How big should X be?
- How willing should we be to place students in their 2nd, 3rd, etc. topic, or even a topic they didn't pick?
- Should we take into account the exchange students? How do we treat them?
- What's the priority of each functionality? (like style over-reactivity, etc..)
- What's the awaited final product/package?
- How to treat a late-joining student who might not be in the system?
- Non-functional requirements questions:
  - What are the specific security requirements for the tool?
  - What kind of data will be stored, and how should it be protected?
  - Are there any regulations or compliance standards that the tool needs to adhere to?
  - How many simultaneous users does the tool need to support?
  - What is the expected peak usage time?
  - What are the expected uptime requirements for the tool?
  - What are the performance requirements for the tool?

- Are there any specific performance benchmarks that need to be met, such as response times or loading times?
- On what kind of software should the system run?
- What kind of error handling and recovery mechanisms should be put in place?

We consent to our report being used as teaching material in the future.

We consent to (parts of) our source code being published under a free open source license.