

Project Report

Exam Planner

Names of students:

Jan Vasilčenko – 293098

Karrtiagehyen Veerappa - 293076

Nicolas Popal - 279190

Patrik Horný – 293112

Supervisors:

Michael Viuff

Astrid Hanghøj

VIA University College

Bring ideas to life
VIA University College



Character Count: 32987 characters

Software Technology Engineering

Semester 1

18 December 2019

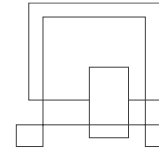
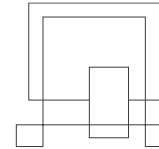


Table of content

Abstract	iii
1 Introduction	1
2 Requirements	4
2.1 Functional Requirements	4
2.2 Non-Functional Requirements	5
3 Analysis	6
3.1 Use Case Diagram	6
3.2 Use Case Description	7
3.3 Link Between Requirements and Use Cases	13
3.4 Activity Diagram	14
3.5 Domain Model	16
4 Design	17
4.1 Class Diagrams	17
4.2 UI Design Choices	22
5 Implementation	25
6 Test	47
7 Results and Discussion	48
8 Conclusion	48
9 Project Future	49
10 Sources of Information	50
11 Appendices	51



Abstract

The aim of the project is to develop a system where the user can plan exams more efficiently since the current planning method is cumbersome and inefficient. The system should be user-friendly and less time-consuming to use compared to the old planning method.

Certain requirements were established by the customer, which had to be fulfilled by the system. A use case diagram was made to define the problem domain further. Use case descriptions and activity diagrams for the use cases were made to explain the use cases and ended up with a domain model.

The design of the system was developed based on the requirements and analysis. The domain model was further developed into UML class diagrams. UI design choices were made to be user-friendly.

With the design completed, implementation of the code was started.

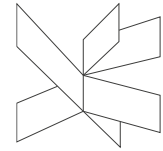
Testing of the system was made by following the use case descriptions.

This project is concerning the development of a system where examination plans can be implemented into a schedule. The customer, a Studies Administration Officer, who is responsible for scheduling examinations, will be the sole user of the system. This makes him or her the target audience.

The current way of scheduling examinations is through pdf files, where the exam planner must download and change the properties of the pdf files each time a change must be made, as seen in the pdf VIA University College., 2019. Examination plan Winter 2019-2020 (Studienet.dk., 2019). An example is shown in Diagram (1) below.

[illegible]

Diagram (1): An example of the current way of making exam plans.



This method is cumbersome and inefficient as it does not allow certain conditions to be programmed in. For example, a student should not have consecutive days of examinations. This would have to be manually checked by the planner for each student. As students in higher semester will have varying courses, this method of planning examinations gets increasingly more difficult. So, a more efficient way of scheduling examinations is needed.

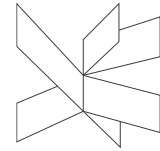
Purpose

The purpose of the project is to make a program that will create a timetable for examinations that will help the exam planner to make exam plans with ease.

Problem Statement

The exam planner is generating pdf version of schedule, that needs to be downloaded and is not lively updated. The current system is problematic because errors can be made easily, and it is inefficient. There are several sub-questions connected to the main problem:

- Which information is necessary to make a schedule?
- What is required for any particular exam?
- How should the students be registered?
- How to spread out the exams so that no one student will have consecutive days of examinations?



Delimitations

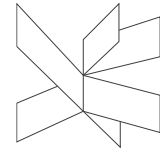
There are some delimitations in this project. These delimitations are not implemented in the project due to specifications from the customer and possessing lower importance while remaining relevant.

1. No grade system will be implemented.
2. No notification system for students or teachers will be implemented.
3. There will be no focus on any other programs beyond the engineering programs offered in VIA University College.
4. The system will not require a server.

For more on the methodology, planned time schedule, risk assessment, and Group Contract, refer to Appendix 1.

There are 8 sections after this: Requirements, Analysis, Design, Implementation, Test, Result and Discussion, Conclusion, and Project Future.

In the Requirements, the requirements made by the customer will be discussed. Further analysis of the problem domain will be made in the Analysis section. design of the system based on the requirements and analysis will be developed in the Design section. Some interesting implementation of code will be explained in the Implementation section. Testing of the system to check whether the requirements were fulfilled will be done in the Test section. Discussion about the outcome of the system will be done in the Result and Discussion. A conclusion will be drawn in Conclusion, and improvements and changes that could have been made will be discussed in the Project Future.



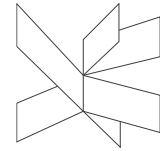
2 Requirements

In the first meeting with the customer, the customer set some requirements that the system should fulfil. Later at a customer session, the requirements were further refined by the customer, which led to some changes in the order of the requirements. The requirements are split into two parts: functional requirements and non-functional requirements. The functional requirements were further classified based on the level of importance, ranging from 'critical priority' to 'low priority.'

2.1 Functional Requirements

Critical Priority:

1. As an exam planner, I want to create plans for exams, so the students and teachers will know when and where the exam will take place.
2. As an exam planner, I want to create two different types of exams, written and oral, so that students and teachers know what type of exam they will have.
3. As an exam planner, I want to create and manage rooms for exams, so that students and teachers know where to go for the exam.
4. As an exam planner, I want to create and manage a list of students who are enrolled for the exam with student names and student numbers, so that teachers know which students are taking the exams.
5. As an exam planner, I want to assign courses to students.
6. As an exam planner, I want to assign written exams to rooms with capacity of 40 or above.



7. As an exam planner, I want to assign oral exams to rooms with HDMI and VGA connections.

High Priority:

8. As an exam planner, I do not want set students to consecutive days of exams
9. As an exam planner, I should not be able to assign a room for oral exams if the same room is already taken in the same day.
10. As an exam planner, I should not be able to assign 7th semester students to exams which take place in the last three days of the exam period.

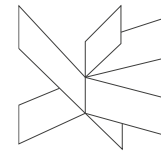
Low Priority:

11. As an exam planner, I should not be able to remove a created exam if there is only 1 more day left to the exam.
12. As an exam planner, I want to set first semester students to their regular classrooms, so they would not get confused where to go.

2.2 Non-Functional Requirements

13. The exam schedule should be displayed as a web page.

In the next section, Analysis, these requirements will be used to make a use case diagram and use case descriptions, which will provide the basic understanding of how the system should operate.



3 Analysis

To better understand the problem domain, a use case diagram is developed.

3.1 Use Case Diagram

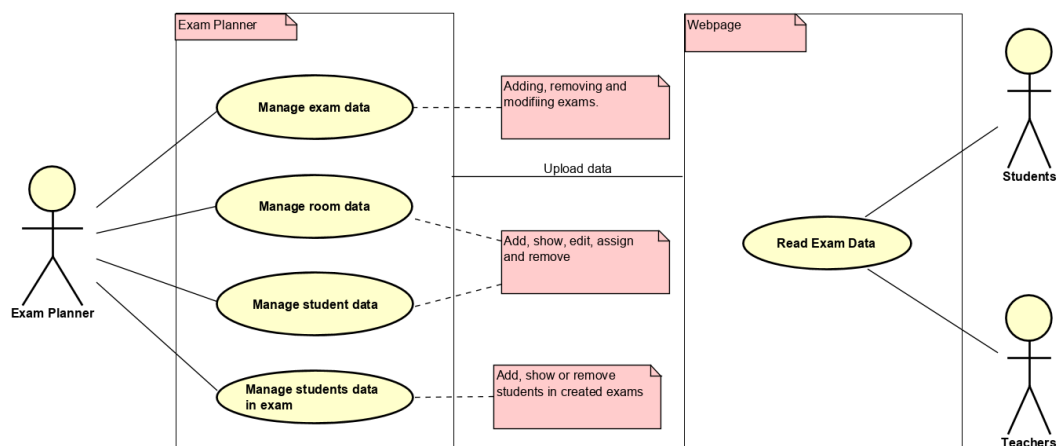
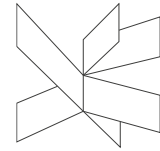


Diagram (2): A use case diagram made using the Astah Professional program.

As seen in Diagram (2), the actor on the left side, the exam planner, wants to manage exam data, manage room data, manage student data, and manage students data in a created exam. The exam planner then would upload or export the data and make it viewable as a webpage for the actors on the right, students and teachers.

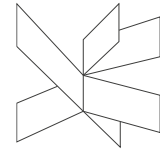
Use case descriptions are made to explain the use cases in Diagram (2).



3.2 Use Case Description

Manage Student Data Use Case Description

Use Case	Manage Student Data
Summary	Add or remove students, or edit student data
Actor	Exam planner
Precondition	Student ID, student name, and the courses that the student is enrolled in
Postcondition	A student is stored in the system, data for existing student has been updated or an existing student has been removed from the list
Base sequence	<ol style="list-style-type: none"> 1. If show go to step 6, if edit go to step 7, if remove go to step 12. <p>ADD:</p> <ol style="list-style-type: none"> 2. If add student, then enter values for: <ol style="list-style-type: none"> a. ID (6 digits) b. Name c. Course names (separated by a comma with no space) 3. Click the add button. 4. System validates data and prompts for illegal value. If the student ID is not 6 digits or the student ID is identical to another student ID, the system shows an error. Then, repeat step 2. 5. If input is valid, then the system adds a new student with the given data. <p>End the use case for ADD.</p> <p>SHOW:</p> <ol style="list-style-type: none"> 6. System shows a list of created students, each element with ID, name and courses. <p>End the use case for SHOW.</p> <p>EDIT:</p>

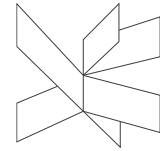


	<p>7. Select a student to be edited. 8. Enter new values for student ID, name, or/and courses. 9. Click the back button. 10. System validates data and prompts for illegal value. If the student ID is not 6 digits or the student ID is identical to another student ID, the system shows an error. Then, repeat step 7. 11. If input is valid, the system updates the data for the edited student. End the use case for EDIT.</p> <p>REMOVE:</p> <p>12. Select a student to be removed. 13. Click the remove button. End the use case for REMOVE</p>
Exception sequence	
Note	

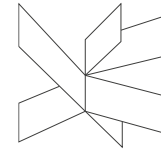
Table (1): Shows the use case description for manage student data.

Manage Room Data Use Case Description

Use case	Manage Room data
Summary	Add or remove rooms, or edit room data
Actor	Exam planner
Precondition	Room name, room capacity, whether room has equipment (HDMI and VGA)
Postcondition	A room is stored in the system, data for existing room has been updated or an existing room has been removed from the list
Base sequence	1. If show go to step 6, if edit go to step 7, if remove go to step 12.



	<p>ADD:</p> <ol style="list-style-type: none">2. If add room, then enter values for:<ol style="list-style-type: none">a. Nameb. Capacityc. Room equipment (true or false)3. Click the add button.4. System validates data and prompts for illegal value. If the room name is identical to another room name, the system shows an error. Then, repeat step 2.5. If input is valid, then the system adds a new room with the given data. <p>End the use case for ADD.</p> <p>SHOW:</p> <ol style="list-style-type: none">6. System shows a list of created rooms, each element with name, capacity and whether the room has equipment. <p>End the use case for SHOW.</p> <p>EDIT:</p> <ol style="list-style-type: none">7. Select a room to be edited.8. Enter new values for name, capacity, or/and courses.9. Click the back button.10. System validates data and prompts for illegal value. If the room name is identical to another room name, the system shows an error. Then, repeat step 7.11. If input is valid, the system updates the data for the edited room. <p>End the use case for EDIT.</p> <p>REMOVE:</p> <ol style="list-style-type: none">12. Select a room to be removed.13. Click the remove button. <p>End the use case for REMOVE</p>
--	----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

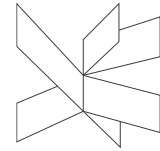


Exception sequence	
Note	

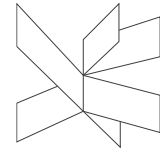
Table (2): Shows the use case description for manage room data.

Manage Exam Data Use Case Description

Use case	Manage Exam data
Summary	Add or remove exams, or edit exam data
Actor	Exam planner
Precondition	Date of exam, exam semester, exam form (Ordinary or Re-exam), exam type (Written or Oral), course, all the rooms are created, and all of the students are created.
Postcondition	An exam is stored in the system, data for existing exam has been updated or an existing exam has been removed from the list
Base sequence	<ol style="list-style-type: none"> 1. If show go to step 6, if edit go to step 7, if remove go to step 12. ADD: 2. If add exam, then enter values for: <ol style="list-style-type: none"> a. Choose a date b. Semester c. Exam form (Ordinary or Re-exam) d. Exam type (Written or Oral) e. Exam course f. Select room 3. Click the add button. 4. System validates data and prompts for illegal value. If the chosen date is not within the date parameter set, semester is not 1-7, type is oral and chosen date has another exam with type of oral, type is oral and the chosen room does not have the equipment, or type is written and the chosen room does not have a capacity of 40 or more, the system shows an error. Then repeat step 2.



	<p>5. If input is valid, then the system adds a new room with the given data. End the use case for ADD.</p> <p>SHOW:</p> <p>6. System shows a list of created exams, each element with date, semester, exam form, exam type, exam course and room. End the use case for SHOW.</p> <p>EDIT:</p> <p>7. Select an exam to be edited. 8. Enter new values for date, semester, exam form, or/and exam type. 9. Click the back button. 10. System validates data and prompts for illegal value. If the chosen date is not within the date parameter set, semester is not 1-7, type is oral and chosen date has another exam with type of oral, type is oral and the chosen room does not have the equipment, or type is written and the chosen room does not have a capacity of 40 or more, then repeat step 7. 11. If input is valid, the system updates the data for the edited room. End the use case for EDIT.</p> <p>REMOVE:</p> <p>12. Select an exam to be removed. 13. Click the remove button. 14. If the chosen exam is equal to or less than 1 day away from the current day, the system shows an error. The repeat step 12. 15. Otherwise, the exam is removed. End the use case for REMOVE</p>
Exception sequence	

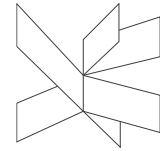


Note	
-------------	--

Table (3): Shows the use case description for manage exam data.

Manage Students Data in Exam Use Case Description

Use Case	Manage Students Data in exam
Summary	Add or remove students in an existing exam
Actor	Exam planner
Precondition	An exam is created with a list of students already loaded in
Postcondition	A new student is stored in the list of students for the selected exam or an existing student in the list has been removed from the list
Base sequence	<ol style="list-style-type: none"> 1. If add go to step 4, if remove go to step 8. <p>SHOW:</p> <ol style="list-style-type: none"> 2. Select the exam for which the students want to be managed. 3. System shows a list of students who are enrolled in the exam. <p>End the use case for SHOW.</p> <p>ADD:</p> <ol style="list-style-type: none"> 4. If add student, then enter values for: <ol style="list-style-type: none"> a.ID (6 digits) b.Name 5. Click the add button. 6. System validates data and prompts for illegal value. If the student ID is not 6 digits, the student ID is identical to another student ID, or the student have consecutive



	<p>days of exam, the system shows an error. Then, repeat step 4.</p> <p>7. If input is valid, then the system adds a new student with the given data.</p> <p>End the use case for ADD.</p> <p>REMOVE:</p> <p>8. Select a student to be removed.</p> <p>9. Click the remove button.</p> <p>End the use case for REMOVE</p>
Exception sequence	
Note	

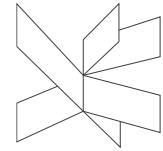
Table (4): Shows the use case description for manage students data in exam.

3.3 Link Between Requirements and Use Cases

Table (5) shows the link between the requirements in Section 2 and the use cases.

Use Case	Covered Requirements
Manage student data	4, 5
Manage room data	3
Manage exam data	1, 2, 6, 7, 9, 10, 11, 12
Manage students in exam	4, 8

Table (5): Shows the link between requirements and use cases



3.4 Activity Diagram

Now that the use case descriptions are made, activity diagrams can be drawn based on the use case descriptions. The activity diagrams are generated to obtain a better understanding of the behavior and flow of the system. The activity diagrams can also provide a visual understanding of how the system works.

Only the activity diagram for creating an exam will be shown here. For the editing and removing activity diagrams of exam, and creating, editing and removing activity diagrams of students and rooms, refer to Appendix 2.

The below diagram, Diagram (3) shows the activity diagram for creating an exam.

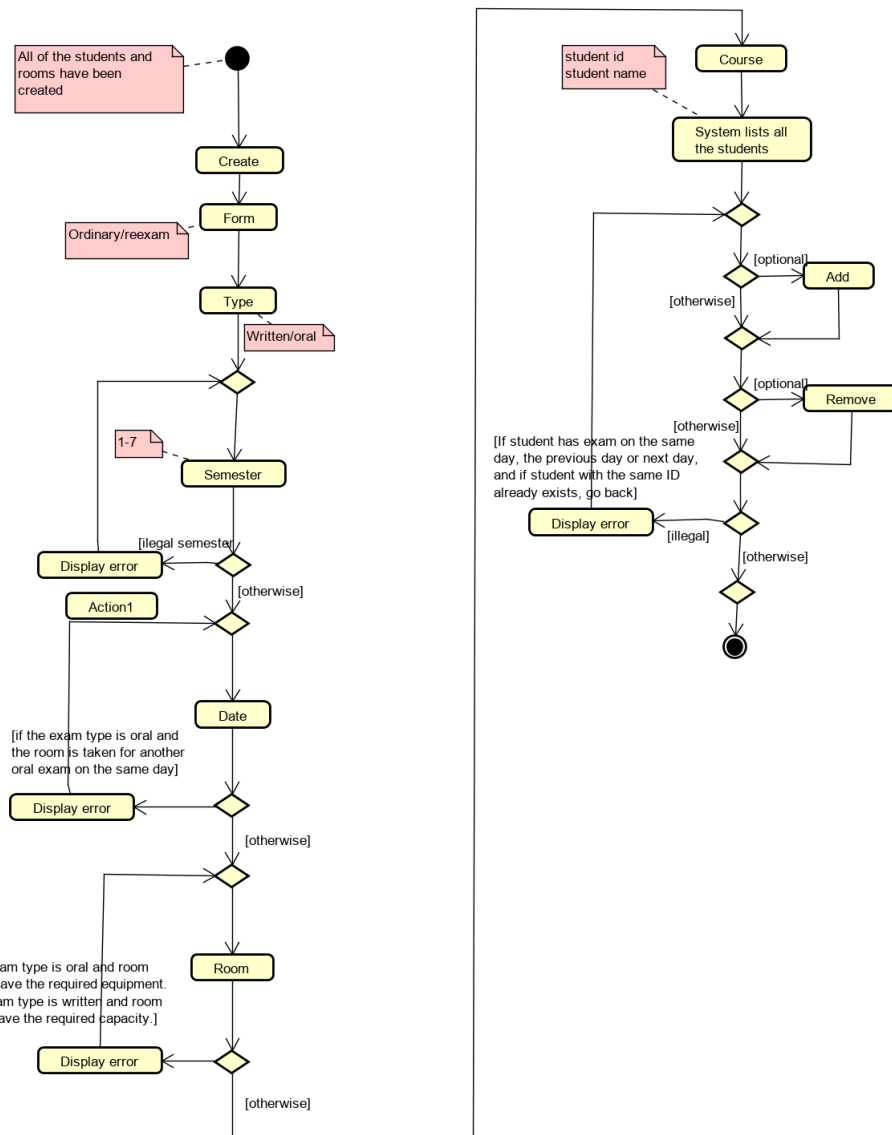
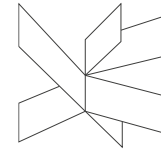
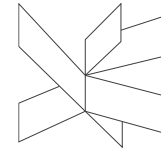


Diagram (3): Shows the activity diagram for creating an exam.

The precondition for creating an exam is that all the students and rooms are created. The form and type of the exam is then inputted. Semester is then given, but if the given semester is not 1-7, the system will display an error and ask for an input for the semester again. Otherwise, the user can continue to Date. After



that, the user is asked to choose the room where the exam takes place. If the exam type chosen earlier was oral and the room did not have HDMI and VGA connections, or if the exam type chosen was written and the room did not have a capacity of 40 or above, an error will be displayed and the user will be asked to choose a room again. Otherwise the user can continue to input the course. After a course is inputted, the system will load in the list of students who are enrolled in this course. Optionally, the user can add or remove students from this list. If there are no exceptions, it is the end of the activity diagram.

3.5 Domain Model

Now that there is a better understanding of the problem domain, a domain model can be made to give a better view of the relations between some of the classes.

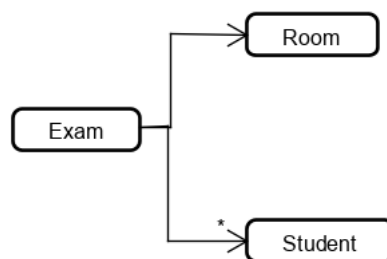
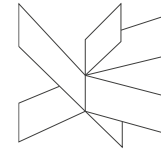


Diagram (4): Shows the domain model (relations only).

The domain model shown in Diagram (13) only shows the relations between the Exam, Room and Student classes. There is a multiplicity sign to Student indicating that there will be an array or ArrayList of Student in exam, for the list of students enrolled in certain exams. But since the number of students in the list



can vary because the exam planner can add or remove students in an exam, it will be an ArrayList.

In the next section, Design, the relations only domain model (Diagram (13)) will be developed further. Also, the technologies used, architecture, class diagrams, and UI design choices will be discussed as well.

4 Design

4.1 Class Diagrams

The domain model (relations only) established in the previous section can be expanded on, as seen in Diagram (14) below.

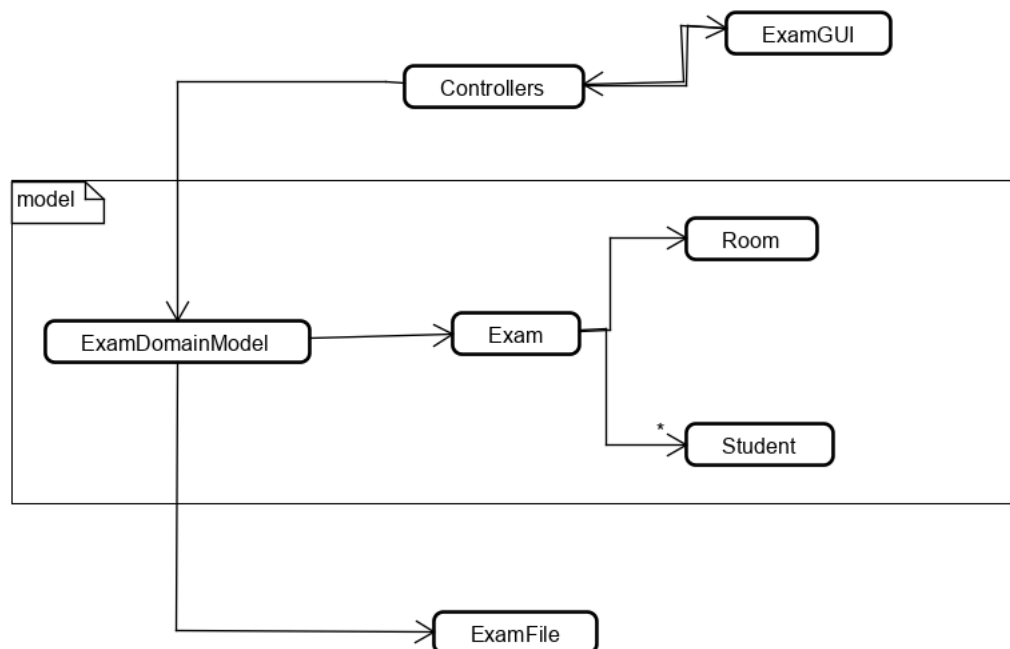
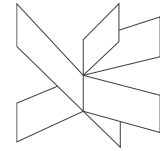


Diagram (5): Shows the expanded domain model (relations only).



In the model, there is a class called *ExamDomainModel*, which will be connected to the *Controllers* and *ExamFile*. The *ExamDomainModel* will contain all of the methods involving logic that will be implemented in the *Controllers*. *Controllers* controls the background action that happens in the *ExamGUI*. The *ExamFile* saves the data.

From this, it can be derived that there will be an interface class from which the model manager will implement the methods.

There should also be a class which will control the changing of scenes of the *Controllers* since there will be a few of them.

Now that the domain model (relations only) gives an idea of how the architecture of the system would look, UML class diagrams can be constructed based on the domain model.

Firstly, the UML class diagrams for *Student*, *Room* and *Exam* are created, as shown in Diagram (15) below.

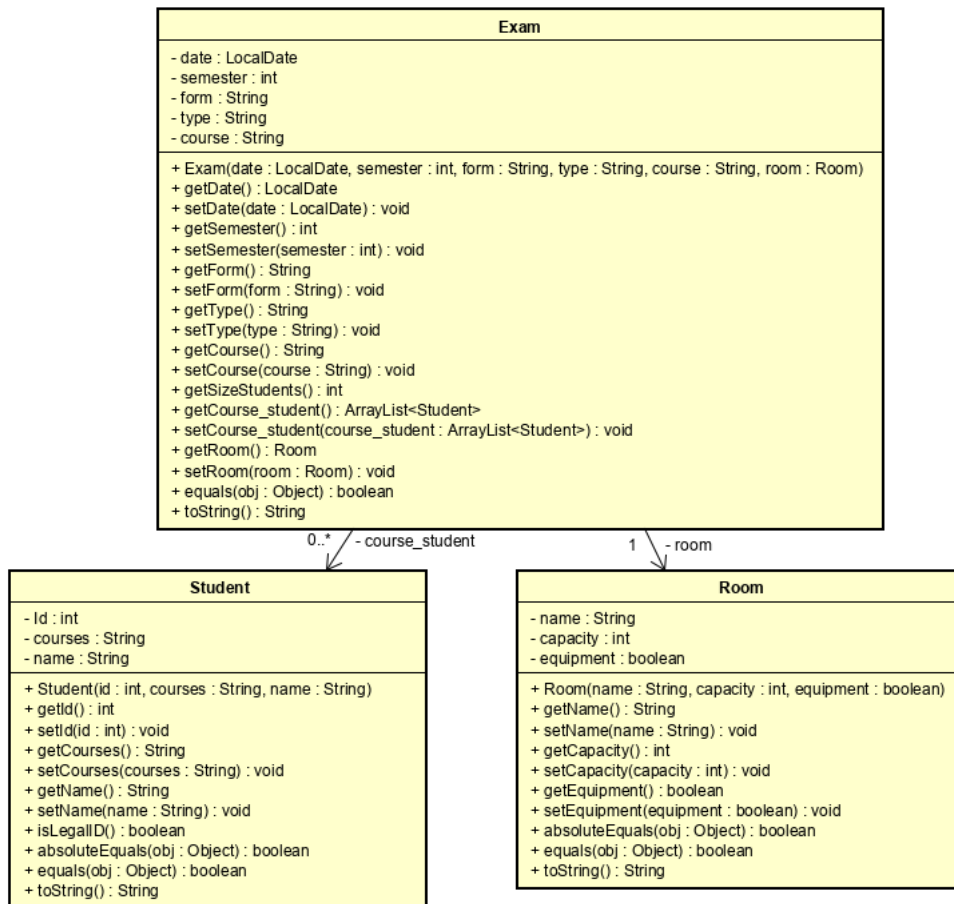
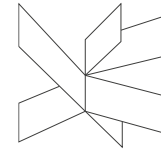
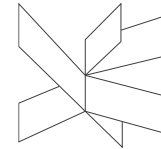


Diagram (6): Shows the UML class diagram for Student, Room and Exam.

The *Student* and *Room* classes are associated to the *Exam* classes. These classes are implemented by the model manager, *ExamDomainModelManager*, creating instances of the mentioned classes. The *ExamDomainModelManager* implements methods from an interface called *ExamDomainModel*, shown in Diagram (16) below.



Project Report – Exam Planner

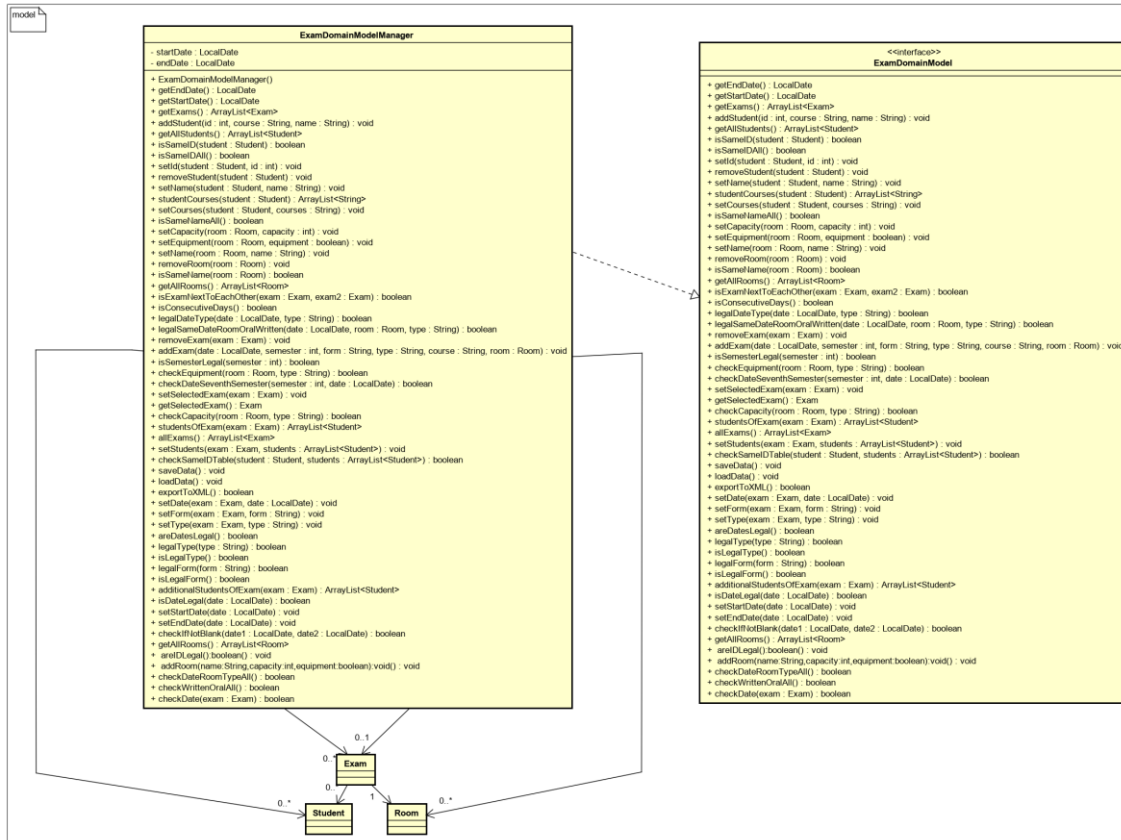


Diagram (7): Shows the UML class diagram for ExamDomainModelManager and the interface ExamDomainModel.

Diagram (16) shows the model, where the logic and methods that will be used in the controllers are implemented.

With the model complete, the view needs to be done with all the controllers and a class called *ViewHandler*, which will control the changing of scenes, which can be seen in Diagram (17).

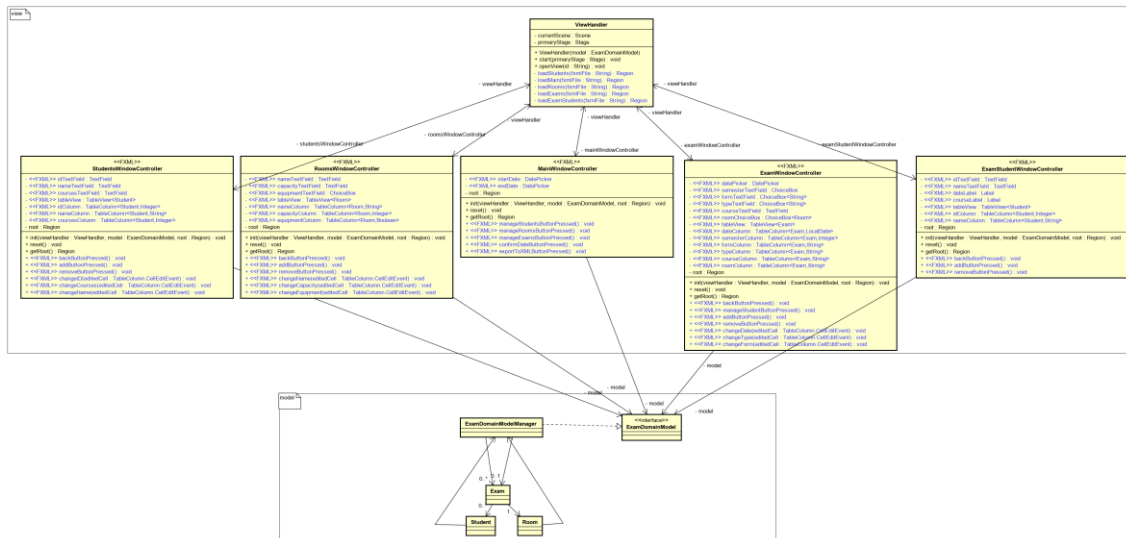
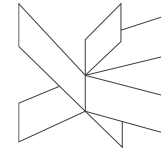


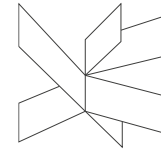
Diagram (8): Shows the UML class diagram for the view and model.

Please zoom in for readability of the class diagram.

There are five controllers called *MainWindowController*, *StudentsWindowController*, *RoomsWindowController*, *ExamWindowController*, and *ExamStudentWindowController*.

All the controllers are connected to the interface, *ExamDomainModel* since it contains all of the methods to be implemented in the controllers.

The *MainWindowController* will be the main window or start window of the system. *StudentsWindowController* will handle the addition of students, editing of student data, and removal of students. Similarly, the *RoomsWindowController* will handle the addition of rooms, editing of room data, and removal of rooms. The *ExamWindowController* will handle the addition of exams, editing of exams data, and removal of exams. And finally, the *ExamStudentWindowControllers* will handle the addition and removal of students in selected exams.



Finally, the *Main* and *MyApplication* is added, shown in Diagram (18).

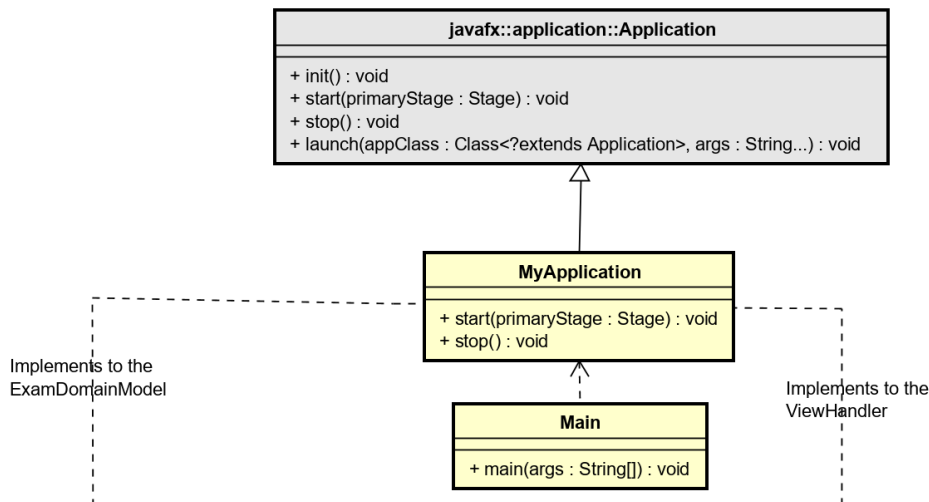


Diagram (9): Shows the UML class diagram for Main and MyApplication.

The *MyApplication* starts and stops the stage.

And with that the UML class diagrams are complete. For the complete UML class diagrams, refer to Appendix 3.

4.2 UI Design Choices

The whole point of this system is to help the customer make plans in an easy and efficient way. Therefore, the layout of the user interface is made to be simplistic and user-friendly.

Figure (1) shows the main window (or scene) of the system.

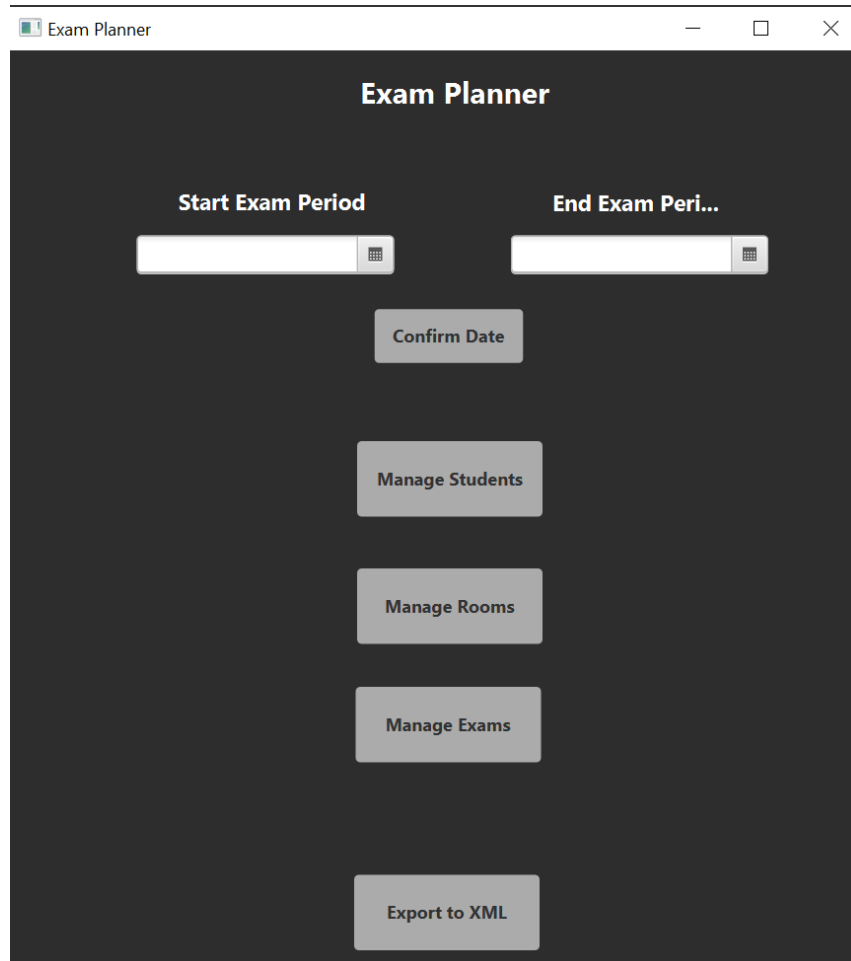
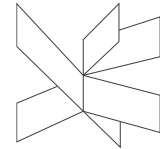
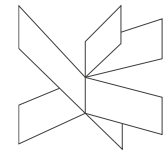


Figure (1): Shows the main window of the system.

The start and the end of exam period are placed near the top of the window since they are the first fields that the exam planner must fill out. The confirm button is placed right below the DatePickers, so that the exam planner could not miss it. Underneath the date section, there are three buttons called Manage Students, Manage Rooms, and Manage Exams respectively. The Manage Students and Manage Rooms buttons are above the Manage Exams button because students and rooms have to be created before creating an exam, as seen in the



Precondition in Table (3) in the Analysis section. Finally, the Export to XML button is placed at the very bottom since that is the last thing the exam planner would do.

The windows for manage students, manage rooms, and manage exams are similar to each other, so only the window for manage students will be discussed here. The layout of the manage students window can be seen in Figure (2).

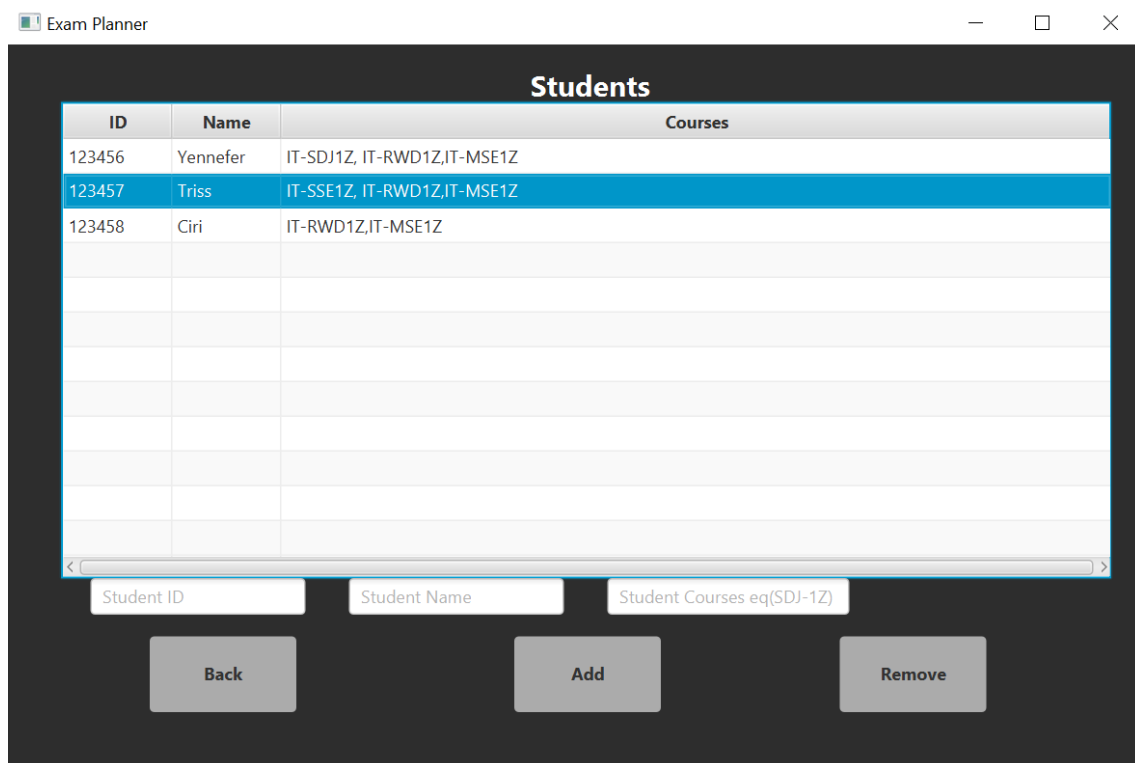
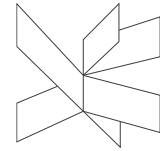


Figure (2): Shows the window for manage students.

A table with a list of created students occupies most of the window, which makes it easy for the exam planner to concentrate on the list. Adding, editing, and removing students can be done in the same window.



The background of the system, as can be seen in Figure (1) and (2), is made to be dark grey. This is because the exam planner may be working for extended periods of time, which can cause eye strain. The dark background can minimize the fatigue of the eyes.

5 Implementation

In this section, some interesting codes will be explained, and the overall layout of the code will be explained as well.

For the full code, refer to the Appendix 4, under the folder called Code, and under that folder called src.

Firstly, the Student and Room classes were created.

Student and Room Classes (Karrtiigehyen)

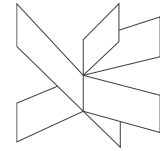
The Student class has 3 instance variables, as seen below in Code (1).

```
7      private int Id;  
8      private String courses;  
9      private String name;
```

Code (1): Shows the instance variables of the Student class.

A constructor is then made to initialize the instance variables, shown in Code (2).

```
11  @      public Student(int id, String courses, String name)  
12      {  
13          this.name = name;  
14          Id = id;  
15          this.courses = courses;  
16      }
```



Code (2): Shows the constructor for the Student class.

The constructor will not be shown again for other classes since it follows the same idea.

Getters and setters are then implemented for *Id*, *courses*, and *name*, as shown in Code (3).

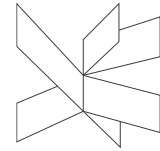
```
18  public int getId() { return Id; }
22
23  public void setId(int id) { Id = id; }
27
28  public String getCourses() { return courses; }
32
33  public void setCourses(String courses) { this.courses = courses; }
37
38  public String getName() { return name; }
42
43  public void setName(String name) { this.name = name; }
```

Code (3): Shows the getters and setters for the Student class.

The *getId()* method will return *Id* with return type *int*. The *setId()* method will set the old *Id* with a new *Id*. The same idea goes for *courses* and *name*.

The getters and setters will not be shown again for other classes since it follows the same idea.

A method called *isLegalID()* is added to ensure that *Id* should be only 6 digits, as seen in Code (4).



```
48     public boolean isLegalID()
49     {
50         if (Id > 999999 || Id < 100000)
51         {
52             return false;
53         }
54         else
55             return true;
56     }
```

Code (4): Shows the *isLegalID()* method in the Student class.

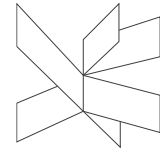
On line 50, there is an if statement which will return false when *Id* is less than or more than 6 digits.

A method called *absoluteEquals(Object obj)* is created to check whether two or more students have the same *name*, *Id* and *courses*, shown in Code (5).

```
58     //Checks if this is the particular student
59     public boolean absoluteEquals(Object obj)
60     {
61         if (!(obj instanceof Student))
62         {
63             return false;
64         }
65         Student other = (Student) obj;
66         return Id == other.Id && name.equals(other.name) && courses
67             .equals(other.courses);
68     }
```

Code (5): Shows the *absoluteEquals(Object obj)* method in the Student class.

Similarly, there is another method called *equals(Object obj)* which compares only the *Id*, shown in Code (6).



```

69      //Checks if Id is the same
70      @ public boolean equals(Object obj)
71      {
72          if (!(obj instanceof Student))
73          {
74              return false;
75          }
76          Student other = (Student) obj;
77          return Id == other.Id;
78      }

```

Code (6): Shows the *equals(Object obj)* method in the Student class.

And finally a *toString()* method is done, which returns *Id*, *name*, and *courses* of a student, shown in Code (7).

```

80      public String toString()
81      {
82          return "ID: " + Id + " Name: " + name + " Courses: " + courses;
83      }

```

Code (7): Shows the *toString()* method in the Student class.

Other classes will not be gone through as thoroughly as this, since most of have similar layout as the Student class. Only interesting and unique pieces of code will be explained.

The room class is also quite like the Student class, but it has instance variables as shown in Code (8).

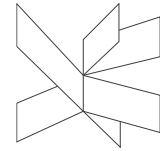
```

7      private String name;
8      private int capacity;
9      private boolean equipment;
10

```

Code (8): Shows the instance variables of the Room class.

Capacity refers to the number of students able to be in the room. This is important since one of the requirements states that written exams have to be in



room with a capacity of 40 and above. *equipment* refers to whether the room has HDMI and VGA connections, which are important for oral examinations. There is also an *absoluteEquals(Object obj)* method in the Room class, which checks whether two or more rooms have the same *name*, *capacity* and *equipment*. And there is an *equals(Object obj)* method, which only checks for the *name*. These methods can be seen in Code (9).

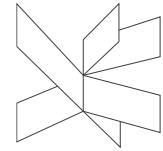
```
48      public boolean absoluteEquals(Object obj)
49      {
50          if (!(obj instanceof Room))
51          {
52              return false;
53          }
54          Room other = (Room) obj;
55          return name.equals(other.name) && capacity == other.capacity
56              && equipment == other.equipment;
57      }
58      @ public boolean equals(Object obj)
59      {
60          if (!(obj instanceof Room))
61          {
62              return false;
63          }
64          Room other = (Room) obj;
65          return name.equals(other.name);
66      }
```

Code (9): Shows the *absoluteEquals(Object obj)* and *equals(Object obj)* methods in the Room class.

Next, the Exam class can be created.

Exam

The Exam class has the follow instance variables, shown in Code (10).

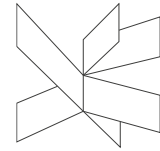


```
9      private LocalDate date;  
10     private int semester;  
11     private String form;  
12     private String type;  
13     private String course;  
14     private ArrayList<Student> course_student;  
15     private Room room;
```

Code (10): Shows the instance variables in the Exam class.

The *form* refers to whether the exam is an ordinary exam or re-exam. The *type* is whether the exam is a written or oral exam. The *course* refers to the course name of the exam. For example, IT-SDJ1Z.

Getters and setters, equals and toString methods are also implemented.



Domain model (Jan Vasilcenko)

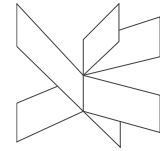
The Domain model interface has the following methods, shown in Code (11).

```
01 public interface ExamDomainModel
02 {
03     void addStudent(int id, String course, String name);
04     void removeStudent(Student student);
05     ArrayList<Student> getAllStudents();
06     boolean isSameID(Student student);
07     void setId(Student student, int id);
08     void setName(Student student, String name);
09     void setCourses(Student student, String courses);
10     boolean isSameIDAll();
11     ArrayList<String> studentCourses(Student student);
12
13     void addRoom(String name, int capacity, boolean equipment);
14     void removeRoom(Room room);
15     boolean isSameName(Room room);
16     ArrayList<Room> getAllRooms();
17     void setName(Room room, String name);
18     void setCapacity(Room room, int capacity);
19     void setEquipment(Room room, boolean equipment);
20     boolean isSameNameAll();
21
22     void addExam(LocalDate date, int semester, String form, String type,
23                 String course, Room room);
24     void removeExam(Exam exam);
25     boolean isSemesterLegal(int semester);
26     boolean checkEquipment(Room room, String type);
27     boolean isDateLegal(LocalDate date);
28     boolean checkDateSeventhSemester(int semester, LocalDate date);
29     boolean checkCapacity(Room room, String type);
30     boolean legalSameDateRoomOrWritten(LocalDate date, Room room, String type);
31     boolean legalDateType(LocalDate date, String type);
32     void setSelectedExam(Exam exam);
33     Exam getSelectedExam();
34     ArrayList<Student> studentsOfExam(Exam exam);
35     ArrayList<Student> additionalStudentsOfExam(Exam exam);
36     ArrayList<Exam> allExams();
37     void setStudents(Exam exam, ArrayList<Student> students);
38     boolean checkSameIDTable(Student student, ArrayList<Student> students);
39     boolean isConsecutiveDays();
40     boolean isExamNextToEachOther(Exam exam, Exam exam2);
41     void setDate(Exam exam, LocalDate date);
42     void setForm(Exam exam, String form);
43     void setType(Exam exam, String type);
44     boolean areDatesLegal();
45 }
```

Code (11): Shows the methods in the Domain Model interface.

These methods are used in Domain Model Manager, in which is implemented actual logic. Domain Model is used so that it is easier to implement these methods and that we also have a list of them.

Domain model manager



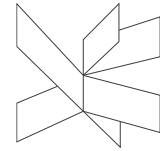
The Domain model manager class has the following instance variables, shown in Code (12).

```
11 public class ExamDomainModelManager implements ExamDomainModel
12 {
13     ArrayList<Student> students;
14     ArrayList<Room> rooms;
15     ArrayList<Exam> exams;
16     LocalDate startDate;
17     LocalDate endDate;
18     Exam selectedExam;
19 }
```

Code (12): Shows the instance variables in the Domain Model Manager class.

These variables are used to store information. For example, ArrayList students is used to store all created student. Variables startDate and endDate are used to mark exam period. And selectedExam is used to pass information of one exam in another scene. In Domain model manager are implemented methods from Domain model. They are initialized here. They are split into four categories: Student, Room, Exam and Date.

Student category has many methods, which checks the input, sets variables for each student, add or remove them in ArrayList and since we are displaying information in tables, some methods check if some information in table are valid, shown in Code (13).



```

55  @Override public boolean isSameIDAll()
56  {
57      boolean isIt=false;
58      for (int i = 0; i < students.size()-1; i++)
59      {
60          for (int j = i+1; j < students.size(); j++)
61          {
62              if(students.get(i).equals(students.get(j)))
63              {
64                  isIt= true;
65                  break;
66              }
67          }
68      }
69      return isIt;
70  }

```

Code (13): Shows method, which loops through ArrayList of students and checks if there are two or more students with same ID.

Room category has methods to add,remove and set exams in ArrayList and also have similar methods like student, because we are displaying rooms in table.

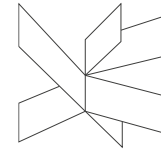
Exam category has one of the most complex methods, for example to check if two exams are next to each other, shown in Code(14) and if student in exams have consecutive exam, shown in Code(15).

```

188  @Override public boolean isExamNextToEachOther(Exam exam, Exam exam2)
189  {
190      if((exam.getDate().minusDays(1).equals(exam2.getDate()))||exam.getDate().plusDays(1).equals(exam2.getDate())){return true;}
191      else return false;
192  }

```

Code (14): Shows method, which checks if two exams are next to each other by dates.



```

194  @Override public boolean isConsecutiveDays()
195  {
196      boolean isIt=false;
197      if(exams.size()==0){return false;}
198      for (int i = 0; i < exams.size()-1; i++)
199      {
200          for (int j = i+1; j < exams.size(); j++)
201          {
202              if(isExamNextToEachOther(exams.get(i),exams.get(j)))
203              {
204                  ArrayList<Student> allstudents=new ArrayList<>();
205                  allstudents.addAll(exams.get(i).getCourse_student());
206                  allstudents.addAll(exams.get(j).getCourse_student());
207                  for (int k = 0; k < allstudents.size()-1; k++)
208                  {
209                      for (int l = k+1; l < allstudents.size(); l++)
210                      {
211                          if(allstudents.get(k).equals(allstudents.get(l)))
212                          {
213                              isIt=true;
214                              break;
215                          }
216                      }
217                  }
218              }
219          }
220      }
221      return isIt;
222  }

```

Code (15): Shows method, which checks if one or more student has consecutive exams.

Student window controller

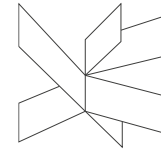
The students window controller class has the following instance variables, shown in Code (16). And also methods initialize, reset and getRoot, shown in Code (17), which are same for Rooms and Exams, so they won't be shown again.

```

12  public class StudentsWindowController
13  {
14      @FXML private Button backButton;
15      @FXML private Button addButton;
16      @FXML private Button removeButton;
17      @FXML private TextField idTextField;
18      @FXML private TextField nameTextField;
19      @FXML private TextField coursesTextField;
20      @FXML private TableView<Student> tableView;
21      @FXML private TableColumn<Student, Integer> idColumn;
22      @FXML private TableColumn<Student,String> nameColumn;
23      @FXML private TableColumn<Student, String> coursesColumn;
24
25      private Region root;
26      private ExamDomainModel model;
27      private ViewHandler viewHandler;
28

```

Code (16): Shows instance variables of student window controller class.



```

30 @ public void init(ViewHandler viewHandler, ExamDomainModel model, Region root)
31 {
32     this.root=root;
33     this.model=model;
34     this.viewHandler=viewHandler;
35     //Sets columns to store type of object in Student
36     tableView.getItems().addAll(model.getAllStudents());
37     idColumn.setCellValueFactory(new PropertyValueFactory<Student, Integer>(s, "id"));
38     nameColumn.setCellValueFactory(new PropertyValueFactory<Student, String>(s, "name"));
39     coursesColumn.setCellValueFactory(new PropertyValueFactory<Student, String>(s, "courses"));
40     //sets the table to be editable
41     tableView.setEditable(true);
42     //changes the value if cell is edited
43     idColumn.setCellFactory(TextFieldTableCell.forTableColumn(new IntegerStringConverter()));
44     nameColumn.setCellFactory(TextFieldTableCell.forTableColumn());
45     coursesColumn.setCellFactory(TextFieldTableCell.forTableColumn());
46 }
47
48 public void reset()
49 {
50 }
51 //Get root method for ViewHandler
52 public Region getRoot() { return root; }

```

Code (17): Shows methods initialize, reset and getRoot.

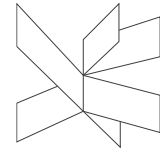
The students window controller class shows the data in table view and had add,remove and edit methods, shown in Code(18).

```

67 public void addButtonPressed()
68 {
69     try
70     {
71         int id = Integer.parseInt(idTextField.getText());
72         Student newStudent = new Student(id, coursesTextField.getText(),
73             nameTextField.getText());
74         if (newStudent.isLegalID() && !model.isSameID(newStudent))
75         {
76             Alert a1=new Alert(Alert.AlertType.ERROR);
77             a1.setTitle("Same ID");
78             a1.setHeaderText("");
79             a1.setContentText("Student already exists");
80             a1.showAndWait();
81         }
82     }
83     else if(!newStudent.isLegalID())
84     {
85         Alert a1=new Alert(Alert.AlertType.ERROR);
86         a1.setTitle("Invalid ID");
87         a1.setHeaderText("");
88         a1.setContentText("You entered invalid ID");
89         a1.showAndWait();
90     }
91     else if(newStudent.isLegalID() && !model.isSameID(newStudent))
92     {
93         model.addStudent(id, coursesTextField.getText(), nameTextField.getText());
94         tableView.getItems().add(newStudent);
95     }
96 }catch(Exception e)
97 {
98     Alert a1=new Alert(Alert.AlertType.ERROR);
99     a1.setTitle("Invalid ID");
100     a1.setHeaderText("");
101     a1.setContentText("You entered invalid ID");
102     a1.showAndWait();
103 }
104 }

```

Code (18): Shows method addStudent, which checks input, save student in domain model manager and adds student in table view.



Remove method follows the same principle, but it doesn't need to check any input. Method just removes exam from table and also from domain model manager.

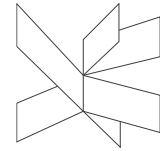
Edit method is set for each column, shown in Code (19) and if there is invalid object, check for every aspect of that object was created in back button, show in Code (20).

```
114 @ public void changeID(TableColumn.CellEditEvent editedCell)
115 {
116     Student selectedStudent=tableView.getSelectionModel().getSelectedItem();
117     model.setId(selectedStudent,(Integer)editedCell.getNewValue());
118     selectedStudent.setId((Integer)editedCell.getNewValue());
119 }
```

Code (19): Shows method changeID, which is used to edit ID in already created students.

```
56 public void backButtonPressed(){
57     if(!model.areIDlegal())
58     {
59         Alert a1=new Alert(Alert.AlertType.ERROR);
60         a1.setTitle("Invalid ID");
61         a1.setHeaderText("");
62         a1.setContentText("Some students have invalid ID");
63         a1.showAndWait();
64     }
65     if(model.isSameIDAll()){
66         Alert a1=new Alert(Alert.AlertType.ERROR);
67         a1.setTitle("Same ID");
68         a1.setHeaderText("");
69         a1.setContentText("Two or more students have the same ID");
70         a1.showAndWait();
71     }
72     if(!model.isSameIDAll()&&model.areIDlegal()){
73         viewHandler.openView( id: "Main");
74     }
```

Code (20): Shows method backButtonPressed, which is used to check if all students have legal data.



Room window controller

The room window controller class follows the same architecture as student window controller so there is no need to show the code again.

Exam window controller

The exam window controller class follows the same architecture as student window controller and room window controller. The only difference is that it has button, which takes one exam and show its students, shown in Code (21).

```
156     public void manageStudentButtonPressed()  
157     {try{  
158         model.setSelectedExam(tableView.getSelectionModel().getSelectedItem());  
159         viewHandler.openView( id: "ExamStudents");}  
160     catch (Exception e){Alert a1=new Alert(Alert.AlertType.ERROR);  
161         a1.setTitle("Invalid selection");  
162         a1.setHeaderText("");  
163         a1.setContentText("You did not select an exam");  
164         a1.showAndWait();}  
165     }
```

Code (21): Shows method `manageStudentButtonPressed`, which is used to store selected exam and take user to another window, in which you can add,remove or edit students of that particular exam.

Adding an Exam (Karrtiagehyen and Nicolas Popal)

An exam is added in the when the add button is pressed. When the add button is pressed, the `addButtonPressed()` method will activate.A sequence diagram is made to explain the `addButtonPressed()` method.

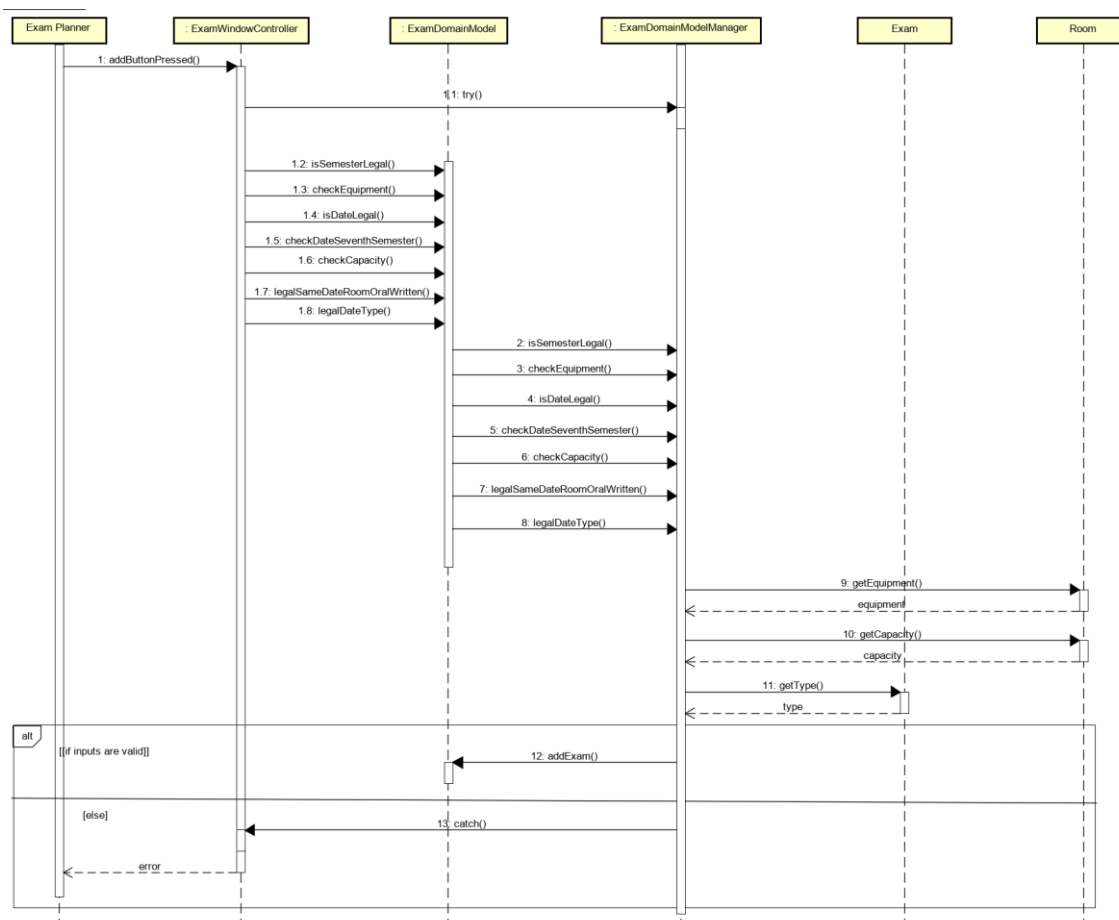
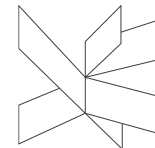
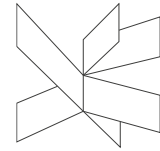


Diagram (9): Shows the sequence diagram for the addButtonPressed() method.

Sequence diagrams for add, edit and remove have not been made for students and rooms since they follow the general idea of the sequence diagram shown in Diagram (9).

Exam students window controller

The exam students window controller class follows the same architecture as student window controller. Only difference is that it automatically adds already created students by their course shown in Code (22).



```

32 @ public void init(ViewHandler viewHandler, ExamDomainModel model, Region root)
33 {
34     this.root=root;
35     this.model=model;
36     this.viewHandler=viewHandler;
37     selectedExam=model.getSelectedExam();
38     dateLabel.setText(selectedExam.getDate().toString());
39     courseLabel.setText(selectedExam.getCourse());
40     idColumn.setCellValueFactory(new PropertyValueFactory<Student, Integer>( s: "Id"));
41     nameColumn.setCellValueFactory(new PropertyValueFactory<Student, String>( s: "name"));
42     tableView.setEditable(true);
43     tableView.getItems().addAll(model.studentsOfExam(selectedExam));
44     tableView.getItems().addAll(model.additionalStudentsOfExam(selectedExam));
45 }
46 //Reset method for exam student window
47 public void reset()
48 {
49     selectedExam=model.getSelectedExam();
50     dateLabel.setText(selectedExam.getDate().toString());
51     courseLabel.setText(selectedExam.getCourse());
52     tableView.getItems().clear();
53     tableView.getItems().addAll(model.studentsOfExam(selectedExam));
54     tableView.getItems().addAll(model.additionalStudentsOfExam(selectedExam));
55 }

```

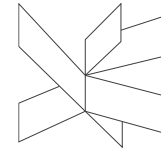
Code (22): Shows method initialize and reset in which students with same course are added and students which user created are added.

```

326 //Returns arrayList of students automatically assigned to exam by course
327 @Override public ArrayList<Student> studentsOfExam(Exam exam)
328 {
329     ArrayList<Student> returnStudents= new ArrayList<>();
330     for (int i = 0; i < students.size(); i++)
331     {
332         ArrayList<String> courses=studentCourses(students.get(i));
333         for (int j = 0; j < courses.size(); j++)
334         {
335             if(courses.get(j).equals(exam.getCourse()))
336             {
337                 returnStudents.add(students.get(i));
338             }
339         }
340     }
341     return returnStudents;
342 }

```

Code (23): Shows method students of exam in model manager, which is used to get all students with same course as course of exam.



Files (Nicolas Popal)

The following code shows the *saveData* method.

```

324  public void saveData() throws IOException
325  {
326      File file = new File( pathname: "data.bin");
327      FileOutputStream fos = new FileOutputStream(file);
328      ObjectOutputStream out = new ObjectOutputStream(fos);
329
330      ArrayList<Object> data = new ArrayList<>();
331      data.add(startDate); //0
332      data.add(endDate); //1
333      data.add(students); //2
334      data.add(rooms); //3
335      data.add(exams); //4
336
337      out.writeObject(data);
338  }

```

Code (24): Shows the *saveData* method

In the method *saveData*, an *IOException* must be thrown, which the program will deal when the method will be used.

saveData method will save program data to the file *data.bin*.

On line 330 in Code (24), the program will create an *ArrayList* of objects and save all data there, which has been created. For saving students, rooms and exams (line 333,334,335), *Serializable* must be implemented in the *Student*, *Room* and *Exam* classes, as seen in Code (25).

```

5      public class Student implements Serializable

```

```

5      public class Room implements Serializable

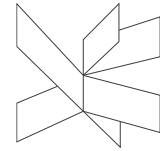
```

```

7      public class Exam implements Serializable

```

Code (25): Shows the *Student*, *Room* and *Exam* classes implementing *Serializable*.



For these occurrences serializable library must be imported in all classes which implement serializable.

```
3 import java.io.Serializable;
```

Code (26): Shows that the classes have to import the serializable library.

In the *MyApplication* class, there is a method called *stop*, which uses the *saveData* method. The *stop* method is used when the user is closing the program.

```
26 public void stop()  
27 {  
28     try  
29     {  
30         model.saveData();  
31     }  
32     catch (IOException e)  
33     {  
34         System.out.println("Error saving file");  
35     }  
36 }
```

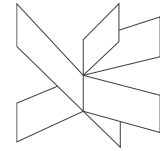
Code (27): Shows the stop method in MyApplication.

We also needed to catch an *IOException*. If the exception is caught program will return the error message “Error saving file”.

In Code (28), the *loadData* method is shown.

```
340 public void loadData() throws IOException, ClassNotFoundException  
341 {  
342     File file = new File( pathname: "data.bin");  
343  
344     FileInputStream fis = new FileInputStream(file);  
345     ObjectInputStream in = new ObjectInputStream(fis);  
346  
347     ArrayList<Object> data = (ArrayList<Object>) in.readObject();  
348  
349     startDate = (LocalDate) data.get(0);  
350     endDate = (LocalDate) data.get(1);  
351     students = (ArrayList<Student>) data.get(2);  
352     rooms = (ArrayList<Room>) data.get(3);  
353     exams = (ArrayList<Exam>) data.get(4);  
354 }
```

Code (28): Shows the loadData method.



In the method *loadData* must be thrown *IOException* and *ClassNotFoundException*, which the program will deal with the exception when the method will be used.

In line 342 in Code (28), file with filename *data.bin* is declared and loaded in line 344 and 345. In line 347 in Code (28), the program creates an *ArrayList* of objects since the method *saveData* is saving data in the *ArrayList* of objects. In line 347 after the equal sign, the program is casting *in.readObject* into an *ArrayList* of objects.

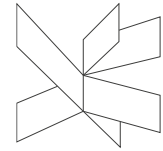
Then from line 349, the program loads data from the *ArrayList* data and saves them to the variables of the program in the same order as they were saved.

In Code (29), it shows the *start* method in *MyApplication*.

```
12  public void start(Stage primaryStage)
13  {
14      model = new ExamDomainModelManager();
15      try
16      {
17          model.loadData();
18      }
19      catch(IOException | ClassNotFoundException e)
20      {
21          System.out.println("Error loading file");
22      }
23      ViewHandler view= new ViewHandler(model);
24      view.start(primaryStage);
25  }
```

Code (29): Shows the start method in *MyApplication*.

In *MyApplication* class, the program is trying to use *loadData* method during start of the program, which can be seen on line 17 in Code (29). If the file is corrupted or doesn't exist, exception is thrown with error message "Error loading file"



In Code (30), the *exportToXML* method is shown.

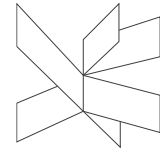
```
356 public boolean exportToXML() throws FileNotFoundException
357 {
358     if(exams.size() < 1)
359         return false;
360
361     String xml = "<?xml version=\"1.0\" encoding=\"UTF-8\"?>\n<exams>";
362     for(int i = 0; i < exams.size(); i++)
363         xml += ("<exam>\n"
364             + "<date>" + exams.get(i).getDate() + "</date>"
365             + "<type>" + exams.get(i).getType() + "</type>"
366             + "<course>" + exams.get(i).getCourse() + "</course>"
367             + "<semester>" + exams.get(i).getSemester() + "</semester>"
368             + "<students>" + exams.get(i).getSizeStudents() + "</students>"
369             + "<form>" + exams.get(i).getForm() + "</form>"
370             + "<room>" + exams.get(i).getRoom().getName() + "</room>"
371             + "</exam>");
372     xml += "</exams>";
373     PrintWriter out = new PrintWriter( fileName: "examData.xml");
374     out.println(xml);
375     out.close();
376
377     return true;
378 }
```

Code (30): Shows the *exportToXML* method.

In the method *exportToXML* must be thrown *FileNotFoundException*, which the program will deal when the function will be used. This method is also of type *boolean*, which will be explained below.

In line 358 in Code (30), the program is using an *if* statement. The *if* statement will return *false* if no exams are saved. In this occurrence, the method will return false and will not continue to line 360.

In the line 361 in Code (30), the program is creating a new variable, *xml*, with basic xml information as a type and encoding. On the next line, the program



needs to use a *for* loop, which will go through all saved exams and fill the *xml* variable with these data. In line 373 in Code (30), the program will create variable for file writer with filename *examData.xml*. In the next two lines, the program will save file and then close it. In the very next line, *true* is returned as we declared this method as a *boolean*.

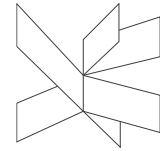
The next code, Code (31), shows the *exportToXMLButtonPressed* method.

```
91     public void exportToXMLButtonPressed()
92     {
93         try
94         {
95             if(model.exportToXML())
96             {
97                 Alert a1=new Alert(Alert.AlertType.INFORMATION);
98                 a1.setTitle("Success");
99                 a1.setHeaderText("");
100                 a1.setContentText("Exams were successfully converted to XML file.");
101                 a1.showAndWait();
102             }
103             else
104             {
105                 Alert a1=new Alert(Alert.AlertType.ERROR);
106                 a1.setTitle("Exams Empty");
107                 a1.setHeaderText("");
108                 a1.setContentText("Exams are empty. There must be at least 1 exam created.");
109                 a1.showAndWait();
110             }
111         }
112         catch (FileNotFoundException e)
113         {
114             System.out.println("Creating xml error");
115         }
116     }
```

Code (31): Shows the *exportToXMLButtonPressed* method.

Program is using method *exportToXML* in *MainWindowController* class in method *exportToXMLButtonPressed* when the actual button 'Export To XML' is pressed.

On line 95 in Code (31), the program is trying to save the file (if value *true* is



returned) and return window with success message. In line 103 when there are no exams saved, the program will open a window with error message. The program will also try to throw an exception and if an error occurs, the program returns error message "Creating xml error".

Javascript (Patrik Horny)

For more on the code for Javascript, refer to the Appendix 4, under the folder called Code.

The Javascript code has a function called a *loadDoc()*, as seen in Code (32).

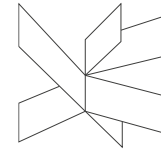
```
1  function loadDoc() {  
2      var xhttp = new XMLHttpRequest();  
3      xhttp.onreadystatechange = function() {  
4          if (this.readyState == 4 && this.status == 200) {  
5              myFunction(this);  
6          }  
7      };  
8      xhttp.open("GET", "examData.xml", true);  
9      xhttp.send();  
10 }
```

Code (31): Shows the loadDoc() function.

This method creates a *XMLHttpRequest*. This object is used to interact with the sever. After that it adds a on a line 3 function, which is activated every-time the *readyState* changes.

The *readyState* property describes and returns the status if the loading document.

The statues changes from 0 to 4. In the code our *readyState* property is equal to 4, which means that the request is finished, and response is ready.



When the *readyState* is equal to 4 and status is equal to 200 which means that request has succeeded, the response is ready. This knowledge was learned from a website page titled “AJAX - The onreadystatechange Event” (w3schools.com, 2019).

The following code, Code (32), shows the *myFunction()* function.

```

12 function myFunction(xml){
13     var i;
14     var xmlDoc = xml.responseXML;
15     var table=<tr><th>Date</th><th>Type of exam</th><th>Course</th><th>Semester</th><th>Students</th><th>Form of exam</th><th>Room</th></tr>;
16     var x = xmlDoc.getElementsByTagName("exam");
17     for (i = 0; i <x.length; i++) {
18         table += "<tr><td>" +
19             x[i].getElementsByTagName("date")[0].childNodes[0].nodeValue +
20             "</td><td>" +
21             x[i].getElementsByTagName("type")[0].childNodes[0].nodeValue +
22             "</td><td>" +
23             x[i].getElementsByTagName("course")[0].childNodes[0].nodeValue +
24             "</td><td>" +
25             x[i].getElementsByTagName("semester")[0].childNodes[0].nodeValue +
26             "</td><td>" +
27             x[i].getElementsByTagName("students")[0].childNodes[0].nodeValue +
28             "</td><td>" +
29             x[i].getElementsByTagName("form")[0].childNodes[0].nodeValue +
30             "</td><td>" +
31             x[i].getElementsByTagName("room")[0].childNodes[0].nodeValue +
32             "</td></tr>";
33     }
34     document.getElementById("demo").innerHTML = table;
35 }
36
37

```

Code (31): Shows the *myFunction()* function.

This function loops through each `<exam>` element in the XML file. Then the nodes (elements) of the XML file are extracted and in the end the element `demo` with the HTML table is filled with the XML data.

CSS (Karstiigehyen)

The system is styled using CSS.

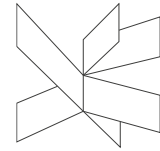
The stylesheet had to be connected to the *ViewHandler*, as shown in Code (32).

```

20 public ViewHandler(ExamDomainModel model)
21 {
22     this.model=model;
23     this.currentScene=new Scene(new Region());
24     currentScene.getStylesheets().add(getClass().getResource("name: "Styles.css").toExternalForm());

```

Code (31): Shows the stylesheet connection in *ViewHandler*.



The system is styled according to some of the design choices discussed in the Design section.

For more on the CSS stylesheet, refer to Appendix.

In the next section, Test, the system is checked whether it fulfils the requirements from the Requirements section.

6 Test

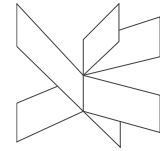
The system is tested by following the base sequences of the use case descriptions from the Analysis section from Table (1) to (4).

Refer to the Requirements section for the requirements number.

Requirements	Fulfilled (Yes / No)
1	Yes
2	Yes
3	Yes
4	Yes
5	Yes
6	Yes
7	Yes
8	Yes
9	Yes
10	Yes
11	Yes
12	No

Table (6): Shows whether the requirements are fulfilled.

The outcome and the fulfilment of the requirements will be discussed in the next section.



7 Results and Discussion

As seen in the previous section, functional requirements 1-11 were all met by the system. The only functional requirement that was not fulfilled was requirement 12, which is about setting first semester students to their regular classrooms, so they would not get confused where to go.

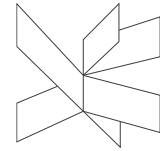
This requirement was not met due to the design of the system. In the system, students are not assigned classrooms, rather they are assigned their courses in which they are enrolled in. For example, IT-SDJ1Z is a course name. This course name gives the information that the student is studying in IT, that the name of the lesson is Software Development with Java and UML, and that the student belongs to class 1Z. But this gives no information about the classroom.

A possible fix to this is to make the user input a classroom for every student, but since this will not serve any purpose to students above the first semester, the inconvenience of the user inputting classrooms for all students outweigh the need for the requirement.

Furthermore, the requirement was categorized as low importance.

8 Conclusion

The customer, a Studies Administration Officer, who is responsible for scheduling examinations, wanted a system which will make planning exams easier and more efficient. Some requirements of varying importance levels were set by the customer. While the system did not fulfil one of the low importance requirements,



all of the other requirements were met. The user can also view the exam plans on a webpage should he or she choose so.

Additional measures were made in the design choices for the ease of planning exams for the user such as the button placements. Care for the well being of the user is was also taken into consideration while designing the system. For example, the background of the system was made to be dark to reduce eye strain. Some changes could have been made to elevate the system's quality, which will be discussed in the next section, Project Future.

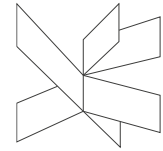
9 Project Future

Changes to the design of the system could be made to incorporate the 12th requirement. Courses or students could have been assigned classrooms, at least for the first semester students or courses.

The system could have been made with databases incorporated as well. This would certainly make the exam planning more efficient.

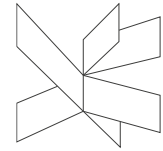
Names of examiners could be added as well, so people viewing the timetable will know who the examiners are.

The naming of some of the classes and methods could be improved in hindsight, to improve the readability of the code.



10 Sources of Information

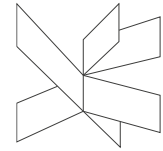
1. VIA University College., 2019. Examination plan Winter 2019-2020. [pdf]
Denmark: VIA University College. Available
at:<https://studienet.via.dk/projects/Examinations_Engineering_Campus_Horsens/Shared%20Documents/Examination%20plan%20winter%202019-2020.pdf> [Accessed 16 December 2019].
2. w3schools.com, 2019. AJAX - The onreadystatechange Event. [online]
Available at:<http://www-db.deis.unibo.it/courses/TW/DOCS/w3schools/ajax/ajax_xmlhttprequest_onreadystatechange.asp.html> [Accessed 17 December 2019].



11 Appendices

The appendices can be found in another zip file.

1. Appendix 1
 - Project Description
2. Appendix 2
 - Activity Diagrams
3. Appendix 3
 - UML class diagrams
4. Appendix 4
 - Code
5. Appendix 5
 - User manual



Process Report

Exam Planner

Names of students:

Jan Vasilčenko – 293098

Karrti gehyen Veerappa - 293076

Nicolas Popal - 279190

Patrik Horný – 293112

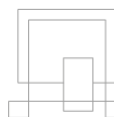
Supervisors:

Michael Viuff

Astrid Hanghøj

VIA University College

Bring ideas to life
VIA University College



Character Count: 13245 characters

Software Technology Engineering

Semester 1

19 December 2019

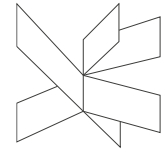
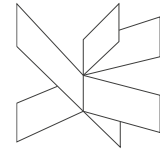


Table of content

1	Introduction	1
2	Group Description	2
3	Project Initiation	3
4	Project Description.....	4
5	Project Execution.....	5
5.1	Methods and tools	5
5.2	Project Development.....	6
6	Personal Reflections.....	8
6.1	Patrik	8
6.2	Jan.....	9
6.3	Karstiigehyen	9
6.4	Nicolas.....	10
7	Supervision.....	11
8	Conclusions	11
9	Sources of Information.....	13
10	Appendices	i



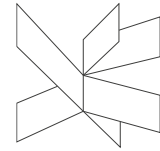
1.Introduction

We were introduced to our first project on the 30th of August 2019. We were given the basic information and concepts about the Semester project and what is needed for the project to be made successfully. Since neither of us had worked on any projects of this scale, we had very little idea how the whole process is going to look like.

After a few weeks, the course, Software Development with Java and UML (SDJ), showed us how we were going to implement the logic into the project and more. On the other hand, the Study Skill for Engineering Students (SSE) course gave us a better insight on how the group should cooperate. This course helped us to better manage our group meetings and set rules for the group.

Every Wednesday, our group had a meeting where we discussed our Semester Project. In the middle of November, our group had meetings 2-3 times a week, since we have gained more knowledge and there was more work to do. Usually, we stayed back in school well after dark to work on the project. Also, the meetings with the supervisors were held during the breaks. They helped us to navigate in the right direction and resolved any uncertainties regarding the Semester Project.

During the days of the SEP weeks (from 6th of December to 19th of December) we spent our days doing the documentation for the Semester Project and fixing the bugs that have appeared in the code. We have managed a few meetings with our supervisors during the SEP week where we discussed the last parts of the Semester Project.



2.Group Description

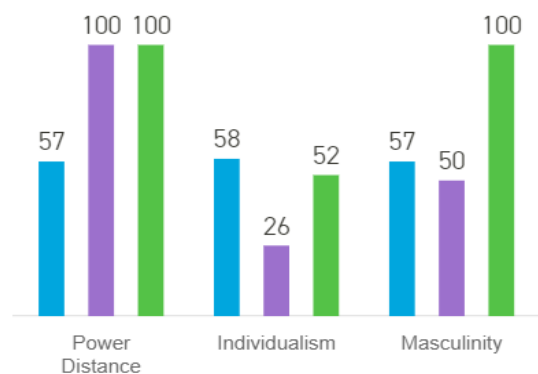


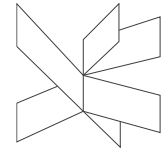
Diagram (1): Shows the Hofstede diagram

A Hofstede diagram was obtained by using an online tool called “Compare Countries” (Hofstede Insights, 2019).

The Hofstede diagram represents a country's perception of power, individualism and masculinity. In this diagram, the diagram represents those features for the Czech Republic (Blue bar), Malaysia (Purple bar), and Slovakia (Green bar).

According to the Hofstede diagram above, the power distance of our countries is on a high level, but that was not the case in our group. No one has taken the role of a leader and all decisions and ideas were approved by all the members of the group. The whole atmosphere in the group was very friendly and every discussion was held with an open mind.

Regarding individualism, some more experienced programmers like Nicolas and Jan were able to do the coding part themselves but were more than happy to explain the



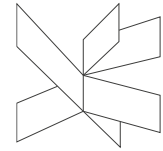
code for the others. But besides coding all members were contributing for the good of the group and nobody was minding their own business.

Even though we came from countries where our societies are represented more by masculine behavior, the team spirit was present for the whole time of doing the project. There was no competitiveness between the members of the group, and everybody was helping each other. But it could be said that there was a strive for achievement as a group instead of as an individual.

On the SSE class we had signed a Group Contract where we defined the rules and our goals. This helped us in certain ways, but even if we did not make a Group Contract, we do not think it would have changed in terms of behavior of the members, since everyone was highly motivated to do the project. For more on the Group Contract, refer to Appendix 1.

3. Project Initiation

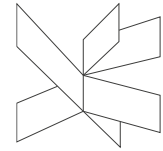
The project initiation phase started on 11th of September. On that day we had gathered in our class where the customer gave us the case in the form of a monologue. He explained in detail what he was expecting from the final product. At the end of his monologue, a Q&A session was held where we could ask about things that were not clear to us. This helped us to better understand the problem.



4. Project Description

The work on Project Description was more complicated than compared to the previous tasks, so we had divided for each of us a section to work on. After finishing our sections, each member presented his section and had to explain his part. During the presentation we had discussed about problems we had and made some changes along the way.

This method proved to be very useful since we gained a deeper understanding about the whole project. With this knowledge we were able to define what things are the most important to focus on so throughout making the project we did not lose the focus on main points about the project.



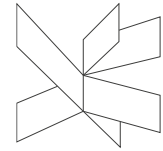
5. Project Execution

After finishing the Project Description, we moved to the Project Execution.

5.1 Methods and tools

The methods and tools that were used for this project are following:

- Waterfall Model - This method is widely used across the software development, so the selection of this method seemed natural. Much like waterfall, the progress is seen as the phases were one by one completed.
- Astah Professional – When making user case diagrams, UML Diagrams, this helped us to have a better overview on what is going on in the project.
- Google Docs – Sharing documents and editing them in real time helped us to reduce the number of meeting and save time. The convenience of not sharing the documents every-time were edited and option to edit the documents in real time was a huge benefit.
- E-Stimate Profile – Defined our roles in a group which led to a more efficient work. With this we could better determine each other's strengths and weaknesses, so everybody worked on a section which suited him the best.

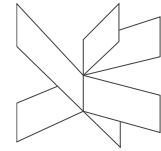


5.2 Project Development

The Project Development went smoother than expected. There were times where we needed to have personal meetings with our supervisor, but after a few meetings the uncertainties were resolved, and we were back on track with the project. Our group was working on a project thoroughly, so everything was done properly and in time.

Since we chose to follow the Waterfall model, we went stage by stage when working on the project. The stages are following:

- Analysis – Appropriate models and logics were chosen. At first, we were kind of insecure if the models we had chosen were the right ones, since there was no coding at the beginning, so we did not have any chance to test the code. We were just talking theoretically about how the whole thing is going to look like, but in the end our choices were proved to be correct.
- Design – We had little clue how the design would look like and what services we would use. As the time passed by, we learned a lot of new things in school which were later implemented. This stage was the one that we would change often, because when for example, we had a meeting with the supervisor, our reasoning was proven to be wrong and we had to change some things. Even though we had anticipated some problems that could potentially come up, there came new ones that needed fixing.
- Code – We were excited when we were finally about to implement our methods and the Graphical User Interface. But our enthusiasm quickly turned into frustration, since the connection between the Graphical User Interface and the

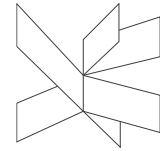


code was far more complicated than we had anticipated. But after some time, we had come to the solutions and things started make sense.

- Testing and Maintenance – After our code was finished, we started to work on errors that might have occurred during the use of the program.

Formulation and writing of the Process Report and Project Report was harder than anticipated. It took time to choose the right words and often we would find ourselves in the position where we didn't know what to do. Even though SSE classes provided a lot of information about how to proceed in terms of the structure and conventions, we were often stuck on certain chapters. Therefore, we needed help from our mentors. But in retrospective, we gained a lot on how to solve these problems in future.

To sum things up, the most difficult part of the project was the Project Execution, where we had to make things work, but that was of course expected to be the hardest. On the other hand, the whole process of doing the Semester Project was entertaining in certain way. We learned a lot about Object Oriented Programming as well as how to function as a group, how to have an effective and productive conversation and how to manage time properly. We think that this was a great introduction to the Software Engineering program and overall, we are happy with the result that the project brought.



6. Personal Reflections

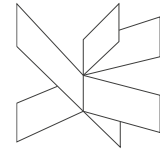
6.1 Patrik

During the work on the Semester Project I learned a lot of new things. Not only how to think like a software engineer, how to work as a group, but I got a lot of insight into how the whole process behind the whole project works. I realized that the most important thing is to understand what's expected from you at the beginning, ensure timely planning, and then slowly step by step start to work on a project.

Even though we had enough time with the whole project, our time management at the end wasn't well planned, since we didn't make the documentation thoroughly and at the end of the SEP week we had to rush a bit, but still everything was done in time. With this experience we will know how to manage time more properly, so next time hopefully there won't be any rush.

As regards our group, I couldn't be more satisfied. There never were any confrontations between us and the whole working atmosphere was very pleasant and friendly. If I didn't understand something, Jan, Nico and Karrii would help me and explain things I didn't understand.

I know that I do not have the best coding logic and my experience isn't on the same level as my colleagues, but I tried to work on a stuff that I knew and also tried to participate in the project as best as I could. Being in this group made me more open-minded to the new ideas and in my opinion being with more experienced people made me progress even more.



6.2Jan

At first it was really hard for me to even think about the group project, because it was my first complex project I have ever worked on. But thanks to the incredible support from my group, we managed to create something that I am proud of.

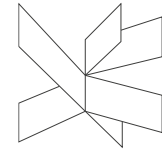
At the beginning, I had no idea how to start or even how to implement such project in code. Deadline was slowly closing in and I was getting increasingly nervous. My motto was to keep everything simple and break even the most complex problems into easier problems and start implementing them. As I said before teamwork, communication and meetings with supervisor were key to success.

Before we created the program, which solved some of the requirements, there were also 'test programs' which did not quite fail, because we learned something from them. After some meetings with the supervisor, we cleared some doubts and created something, which not might be the best program for planning exam and can be more improved upon, but I can say that I am really satisfied by what we have created and I am looking forward for future projects

6.3Karrtiigehyen

I felt that the Semester Project went better than I could have hoped for. This was possible because of the group I was in. Everybody in the group participated and showed interest in the Semester Project. There was low power distance in our, meaning everybody could voice their opinion and it would be taken into consideration.

We also utilized each of the group members' expertise to efficiently handle the Semester Project. Nicolas, who was experienced in using Astah, made the most of the diagrams since he could do it with ease. Patrik helped with making a lot of the report. Jan



contributed to group by laying the groundwork for the code. And I structured the reports since I have experience with making reports. But everybody helped make the code, and everybody helped everybody else in the group should someone have a problem.

The supervisors were also a big help. I had constant doubts about the Semester Project, and the supervisors were also available and guided me in the right direction.

We did have some time constraints as the deadline approached. Time management could have been better. Making diagrams in Astah was also quite hard and complicated at times. Maybe a tutorial or introduction to Astah in class would have helped immensely.

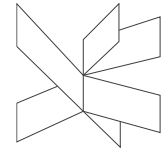
6.4Nicolas

When Michael introduced us to our semester project, I had no idea, how we were going to do it. It was too hard for me to understand, how we could implement the project, what our teacher wanted, even though coding was not strange to me. As time went by the project was more understandable, thanks to my team. They were supportive and when I had a problem, they tried to explain it until I finally understood.

Communication was a key in our group. We divided the project work and if someone did not understand something, others from our team were glad to help. If we needed help, we set up a meeting with our supervisor and everything was clear.

The only thing I would like to know more is work with Astah program. We did almost all project report in this program and it was introduced to us just in presentations, so it took us a lot of time to learn how to operate with it. In the end we managed how to do it in time, but it was tough for us.

In summary I am glad, that I chose this program and I can see my improvements in coding and writing reports. I am grateful that I could be part of our group and make new



friends. I am looking forward for the second semester to improve my knowledge about programming and report writing.

7. Supervision

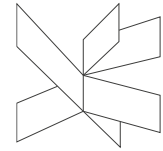
Supervision from our supervisors was available through the whole process of the Semester Project and since this is our first project, understandably, we had to make a lot of meetings. Without the supervision from our supervisors and their guidance, we would have struggled with solving the problems and the final result would not be as good as it is now.

In terms of communication we usually had meetings with our supervisors during the class, if there was enough time, and after class where there was much more time needed to explain certain things. Take for example the Graphical User Interface, our supervisor was a great help, since this task was one of the hardest to apply.

8. Conclusions

As a group we had agreed on some points that we would like to keep in mind for the next Semester Project:

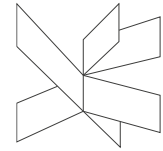
- **Don't try to underestimate how much time you got** – At first it seemed that our workflow was enough, and everything is going smoothly, in the end we found ourselves with overloaded with more documentation left than



anticipated. With this experience, we need to work a little bit more on our time management.

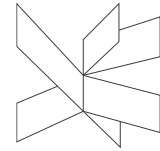
- **Pay close attention to what's most important** – During the Project Description our group would often overthink stuff, which led us to focus on things that are not that important for the Semester Project. This led us to lose a bit of track on what to focus.
- **Good communication in the group is essential** – The project would have gone much more hectic and messier if it was not for a good group. But what we realized is that good communication and honesty was essential to the group's performance.

In conclusion, we think that the main purpose of the Semester Project was at last achieved and through the development we enriched our problem-solving and software engineering knowledge that will be put in use in next Semester Project.



9.Sources of Information

- 1) Hofstede Insights, 2019. *Compare Countries*. [online] Available at:
<<https://www.hofstede-insights.com/product/compare-countries/>> [Accessed 17
December 2019]



10. Appendices

Appendix 1: The Group Contract

Appendix

Group 9

Group Name (optional):

Date: 7/10/2019

Group 9 with team leader

Bob

These are the terms of group conduct and cooperation that we agree on as a team.

Participation: We agree to....

Actively participate in all the assignments and project works, which are assigned to us.

Communication: We agree to...

Honestly and openly communicate with each other without any prejudice. Inform other group members if something is not clear, so they could help us.

Meetings: We agree to....

Meet on our discussed time. If the person is not present, he is obligated to inform the other group members of his absence.

Conduct: We agree to....

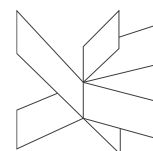
Be on time, be active, be prepared for the meetings. Listen to the others group members opinions and respect it.

Conflict: We agree to....

Talk about it and solve it, so every group member is satisfied.

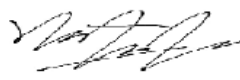


Deadlines: We agree to....

Hand in our assignments in time. Don't start to work on the projects at the last minute.



Other Issues:

Don't be a Bob.

Group member's name	Student number	Signature
Karthegeyn Veerappa	293076	
Jan Vasilcenco	293098	
Nicolas Popal	279190	
Patrik Horny	293112	