

# **REPORT**

## **Computer Science 1st Year**

### **Project Exam - RoSys**



**LILLEBAELT**  
**ACADEMY**  
UNIVERSITY OF APPLIED SCIENCES

## **OEADM16EIC - Group 1**



Roxana Ion



Hedviga Arta



Jonas Laursen



Matthew Peterson

# Table of Contents

<b>Table of Contents</b>	<b>2</b>
<b>Preface</b>	<b>4</b>
<b>Reading Guide</b>	<b>4</b>
<b>Introduction</b>	<b>5</b>
Ethics	5
<b>Roles</b>	<b>5</b>
<b>Tools and Methods</b>	<b>6</b>
Microsoft Visual Studio	6
Microsoft SQL Server	6
MeisterTask	6
GitHub	6
Google Drive	7
Facebook	7
P.O. Communication	7
SCRUM	7
SOLID	7
TDD	8
<b>Business Analysis</b>	<b>8</b>
Business Model	8
Business Case	10
FURPS+	11
<b>BPMNs</b>	<b>12</b>
<b>Use Cases</b>	<b>12</b>
<b>Personas</b>	<b>13</b>
<b>Scenarios</b>	<b>14</b>
<b>System Artifacts</b>	<b>15</b>
Domain Model	15
Object Model	16
Design Class Diagram	17
Overview	18

System Sequence Diagrams	23
Operation Contracts	24
Sequence Diagrams	24
<b>Tests</b>	<b>26</b>
Unit Tests	26
Integration Tests	27
<b>Implementation</b>	<b>28</b>
Request Reservation	28
Initialize	29
Threading	31
<b>Design Patterns</b>	<b>31</b>
Singletons - Just a single instance	32
Factory - Switching on the fly	32
Facade - Building Bridges	34
<b>GUI</b>	<b>35</b>
<b>Database</b>	<b>38</b>
Normalization	38
Final Database Schema	39
The Change table	40
Abstracting the Connection	42
<b>Conclusion &amp; Evaluation</b>	<b>43</b>
<b>Project Future</b>	<b>43</b>
<b>Next Project</b>	<b>44</b>
<b>Appendix</b>	<i>Continued</i>

## Preface

The report is made as a part of participating in a project on the courses: System Development and Programming in the first year of the Computer Science education at Erhvervsakademiet Lillebælt.

This document contains software development artifacts that were created during the sprints, a description of our process and snippets of our code.

The purpose of the report is to inform readers of the development of our project and the methods and tools we used to complete it.

## Reading Guide

The report is based around American spelling and grammar. The project is split into two main parts, the report, which is described in the preface, and the appendix, containing extra materials referred to in the report.

Words such as: us, we, our, members, etc. refer to the group as a whole.

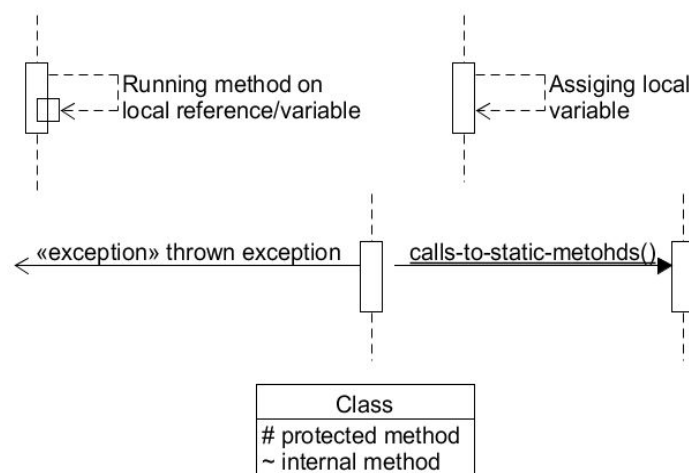
The word “Client” refers to both students and teachers.

The word “Users” refers to admins, students and teachers.

The program itself is named “Room Reservation System”, and for short we named it RoSys, pronounced as roses.

Throughout this report, we have attempted to stick to a single use-case which we have inserted the various diagrams for.

For our diagrams we used some non UML standard symbols. You can see examples of these here.



# Introduction

Erhvervsakademiet Lillebælt is an academy of higher education, which currently spans 2 campuses, one in Odense and one in Vejle.

EAL promotes project work highly, and thus it has many project rooms available for student use. Unfortunately, as we will explain in our problem description, there are some issues with the assignment of project rooms.

The software we are developing is aimed to serve as a demonstration of how to better handle room reservations, and will, hopefully, be picked up for further development and integration into the school's system.

## Ethics

When we started working on the Exam Project, one of our first decisions was to create a Group Contract, which can be seen in Appendix page 13, where we would establish rules and penalties for breaking these rules.

From our previous experience, we observed that this was a good way of managing the team and motivating the team members to stick to important rules that would lead to a high-quality development process, therefore, this was sufficient reason for us to create a group contract.

## Roles

Each member was assigned areas of main responsibility to help facilitate task assignment. Each member did some work on Code, GUI, Tests, Database, Quality Assurance, Diagrams, Use cases, Normalization and Report, but these are the things each member focused on:

### ***Hedviga Arta***

Implemented GUI  
Business Model  
Personas

### ***Roxana Ion***

Implemented GUI  
BPMN  
Object Model

### ***Jonas Laursen***

Creating tests  
Scenarios  
Domain Model

**Matthew Peterson**

Creating tests

Business Case

Operation Contracts

## Tools and Methods

This section covers the basic tools and methodologies we have used in our project, as well as a description of why this tool or method was selected.

### Microsoft Visual Studio

We chose to use Visual Studio because it is the tool we are most familiar with. It works well for the programming language C# and the markup language XAML.

### Microsoft SQL Server

We decided to use a database, rather than a flat file system, to allow better integration with other systems.

We are using MSSQL because a free one was provided to us by the school and is what we have been taught to use in our programming course, so we feel the most confident using this.

The database serves as our persistent storage. To interface with the database, we used Microsoft's SQL Server Management Studio, as this is the program we used in class, and for our program, we used stored procedures.

### MeisterTask

We used MeisterTask to facilitate the organization of the development of our project. MeisterTask also served as our SCRUM Board.

We chose to use this tool because it contains lots of useful features such as time tracking, task assignment and because it is user-friendly.

### GitHub

To help ensure proper backup and logging of our activity, we decided to use Git for version control and GitHub for remote storage.

GitHub is a very popular tool among developers and is widely supported, which is why we decided to use it.

More information about how we used Git and GitHub can be found in the Appendix; page 13.

## Google Drive

We chose to use Google Drive for storing our report and some other shared files because the cooperation features of Google Drive are in our opinion, superior to other options, such as OneDrive.

## Facebook

For the communication part of our project, we chose Facebook chat, because it is practical and everyone has easy access to it.

## P.O. Communication

The communication between us and our Project Owner happened on a weekly basis, this was in the form of face-to-face and Skype meetings. Due to his busy schedule, we did not have a specific day for the meetings, so we decided on our next meeting date at the end of each meeting.

The meetings were for updating our progress, finalizing the next week's sprint, ensuring that our visions were aligned and prioritizing our Product Backlog.

## SCRUM

While working with SCRUM, our work was based on iterative cycles (sprints), with a length of one week. For each sprint, we decided that we were going to set and fulfill one use case. This use case would serve as a guide to help us with splitting and prioritizing tasks in order to fulfill the requirements needed. Every Wednesday one sprint ended and a new one began.

We tried to have a retrospective at the end of every sprint and begin each day by having, either a sit-down or stand-up meeting.

We switched scrum masters every sprint, so that every member of the team could experience the responsibilities that came with this position.

To ensure our tasks are completed with high quality, they went through a Quality Assurance step. Criteria on how to QA a task are described in our group contract, seen in the Appendix; page 13.

## SOLID

To ensure our code can be developed in high quality, we employed the SOLID principles.

## TDD

Another way of ensuring that we wrote quality software from the beginning, is by using TDD practices.

By using TDD we assured that our code and the business requirements parallel each other; TDD also helps us to adhere to the SOLID Principles.

## Business Analysis

### Business Model

In order to understand our product owner's company better, we created a business model that represents a description of the company.

We acquired all the information from Lillebælt Academy's website and from the various documents found on the website. <sup>1</sup>

In collaboration with our product owner, we were able to receive valuable information regarding the company.

<b><i>Key Partners</i></b>	<b><i>Key Activities</i></b>	<b><i>Value Proposition</i></b>	<b><i>Customer Relationships</i></b>	<b><i>Customer Segments</i></b>
<ul style="list-style-type: none"> <li>- Other universities (for exchange programs, international experience)</li> <li>- Businesses (for lectures, events, internships)</li> <li>- Students</li> <li>- Teachers</li> <li>- Staff</li> <li>- Agents</li> </ul>	<ul style="list-style-type: none"> <li>- Teaching students</li> <li>- Organizing events and guest lectures.</li> <li>- Erasmus and other exchange programs.</li> <li>- Internships</li> </ul>	<ul style="list-style-type: none"> <li>- Improving knowledge by teaching students and offering them a pleasant study environment.</li> </ul>	<ul style="list-style-type: none"> <li>- Academy provides students with counseling.</li> <li>- Customers and all who are interested can call or get to information to get personal assistance.</li> <li>- Customers can also get offers by</li> </ul>	<ul style="list-style-type: none"> <li>- People who want to learn and study.</li> <li>- Students from other countries or schools.</li> <li>- Business owners who want to employ new workers.</li> </ul>
	<b><i>Key Resource</i></b>			

<sup>1</sup> <https://www.eal.dk/> EAL Website



<ul style="list-style-type: none"><li>- FabLab</li><li>- Odense and Vejle Municipalities</li><li>- Government</li></ul>	<ul style="list-style-type: none"><li>- Campuses</li><li>- Equipment</li><li>- Teachers knowledge</li><li>- Staff</li><li>- Government</li></ul>		<p>webpage or web shop.</p> <ul style="list-style-type: none"><li>- Teacher assistance</li><li>- Jobportal for offering student jobs.</li></ul>	<p>- Other schools or educational institutions who want to enroll new students.</p>
			<p><b>Channels</b></p>	
			<ul style="list-style-type: none"><li>- Lillebaelt Academy University of Applied Sciences creates awareness of its services through agents and other educational institutions, webpage and by working together with local businesses.</li><li>- Lillebaelt Academy University of Applied Sciences assesses their quality by making questionnaires and having surveys from the government's side and also analysis.</li></ul>	
<p><b>Cost Structures</b></p>		<p><b>Revenue</b></p>		
<ul style="list-style-type: none"><li>- Equipment</li><li>- Salaries</li><li>- Events</li><li>- Clothing line</li></ul>		<ul style="list-style-type: none"><li>- Municipality/Governments funds</li><li>- Lillebaelt Online Shop</li><li>- Non-EU International students</li></ul>		

## Business Case<sup>2</sup>

In our business case we looked at the various effects our system would have on the school and discussed the pros and cons of these. Our full business case can be found in the Appendix; page 15.

### Problem Description

Project rooms are often reserved multiple weeks ahead of time, especially during the project weeks, yet rooms are often left empty. People reserve rooms and do not use them or only use them for part of the time.

This is quite an annoyance to students, as when they request a room from the reception, their request cannot be fulfilled, and the students then must walk around the school searching for rooms not being used.

Students have also experienced that while attempting to reserve a room via email, another group of students went to the reception and were assigned the room, even though the email had already been sent and received; Hence the students who had sent the email had to give up the room to those who went to the reception.

The problem occurs on an almost daily basis, and it is most prominent during project weeks and close to the end of semesters, due to exam work.

We were initially introduced to our problem during a previous project, “Project EAL Campus,” where we had to investigate our school and find something we considered to be an issue.

During the exhibition of the previous project, we found many other students had also picked up on the project room issues, therefore, we decided to investigate this issue further.

### Main Solution

We want to develop a program to manage the reservations, rather than having people sit and slot people into calendars. It will have CRUD operations for reservations, users and rooms. It will also send notifications to users about their reservation and require them to confirm the reservation when they get to the room. We would also like it to have a queueing system.

This was the best solution to the problem because it would not be expensive for the school to implement, and it has a very quick response time compared to our other solutions.

---

<sup>2</sup>

<https://resources.workfront.com/project-management-blog/how-to-write-a-business-case-4-steps-to-a-perfect-business-case-template>

## FURPS+

Before meeting with our product owner, we looked at the materials we had gathered already and set up what we believed to be the optimal FURPS+, which we showed our project owner during the first meeting, for which he provided feedback on. Our FURPS can be seen below.

We used these FURPS+ requirements when making tasks, and to help align our product owners expectations with what we would be able to deliver.

Performance is a main criterion for our program. RoSys should work quickly and efficiently. The focus was also aimed towards reliability. The core part of the program is meant to run on a server, thus, should never be shut down. In case of expected or unexpected shutdowns, e.g, server crash or windows update, we wanted the program to return to its previous state. The program should also be easy to integrate into other systems, hence we attempted to follow SOLID for improved supportability. However, since we will be handing over the full source code of our project, it was not the highest priority.

To allow others to easily read and change our program, we wrote into our group contract various rules to help adhere to a more linear naming scheme.

**Functionality** – The system must require the user to check into their room when it is time for them to use it, otherwise, the reservation will be canceled. Users should be able to reserve rooms, cancel reservations and view their reservations. The system should alert users when they have a reservation soon. There must be different user types that have different privileges.

**Usability** – The user interface is not the priority for the project because the project owner plans to integrate it into other systems that already have user interfaces. We will, however, create a user interface to demo it. The program should be intuitive and easy to use. Users should be able to do what they need to do in a very short amount of time.

**Reliability** – The system should never have problems reserving rooms unless the room is not in the system, then it will need to be added by an admin user.

**Performance** – The program is not very complex, so it will not require much processing power. There should only be short delays while using the system.

**Supportability** – The system should be created using TDD. The system will need to be updated if the room numbers change, rooms are added or removed or information about a room changes. These things will need to be updated by an admin user.

**Design Constraints** – The system must be able to integrate with other systems. The system must use a relational database, WPF user interface, C# and TDD.

**Implementation Requirements** – The programmers must adhere to the standards set out in the group contract. TDD must be used to guarantee that everything works correctly.

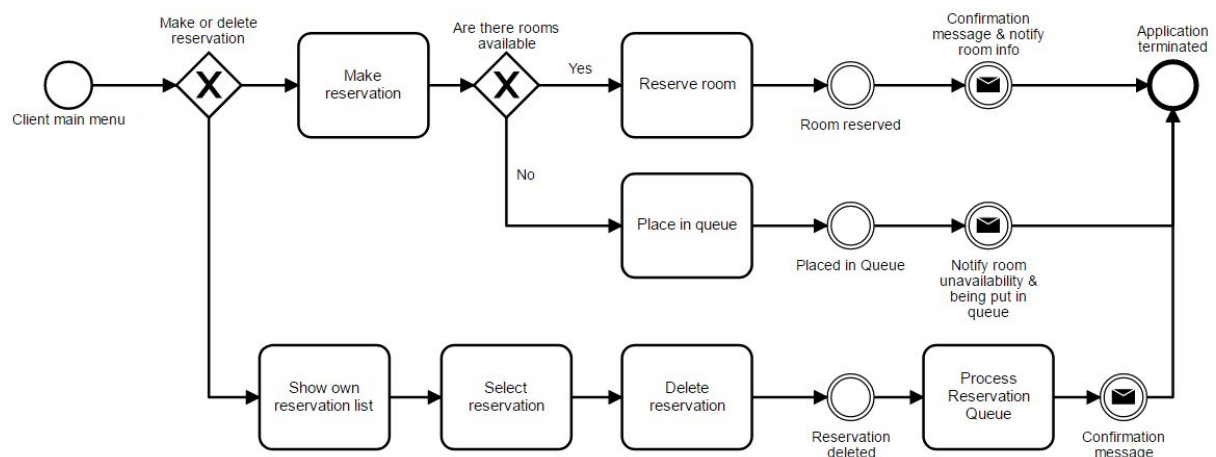
**Interface Requirements** – It must interact with any system the project owner wants to use it with.

**Physical Requirements** – It should be able to run on any modern Windows-based System.

## BPMNs

In order to illustrate and get a better understanding of our program, the flow and the multiple processes it's going through, we have created BPMNs.

Below is our BPMN for Clients, which we have split in two parts (for readability). We have also created a BPMN for Admin, which can be found in Appendix; page 23



## Use Cases

In this project, we have created five use cases that reflect the interaction between our actors and the system, as well as the requirements needed, the alternate flows and the outcomes.

These are our five use cases:

1. Client reserves a room
2. Client deletes own reservation
3. Admin reserves a room
4. Admin deletes a reservation
5. Notification system

As mentioned before in our Reading Guide, for this report, we are going to mainly focus on one use case, which will be the “Client reserves a room” use case. This way

we make sure that our report follows a logical progression and also covers all the topics we have learned throughout the year.

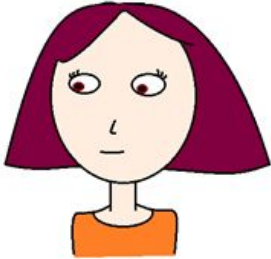

As the name suggests, the use case “Client reserves a room” is responsible for the process of students and teachers reserving a project/meeting room.

Our other use cases can be found in the Appendix; page 3.

<b>Use case name</b>	Client reserves a room
<b>Use case description</b>	A client wishes to reserve a room from the system
<b>Actor</b>	Client (Students & Teachers)
<b>Pre-conditions</b>	None
<b>Basic Flow (Success Scenario)</b>	<ol style="list-style-type: none"> <li>1. Client enters reservation system</li> <li>2. Client inputs date, time frame and amount of people</li> <li>3. System assigns the client a room</li> <li>4. Notify client which room he's been assigned</li> </ol>
<b>Post-conditions</b>	Reservation has been created in the repository, the rooms and users lists and the database
<b>Alternate Flow</b>	<p>AF1: No rooms available</p> <ol style="list-style-type: none"> <li>3. System cannot assign a room</li> <li>4. Put reservation in queue</li> <li>5. System throws exception, NoRoomsAvailableException</li> <li>6. Notify user that no rooms are available and that they have been placed in the queue</li> </ol>

## Personas

To understand the way RoSys works and how it would be used, we wanted to know who would be using it. We made two student, one admin and one teacher persona, but in the report we have only included the student personas because the “Reserve a project room” scenario only uses these two. Since students will be using our system more often than admins or teachers, we decided to make two student personas. Other personas can be found in Appendix; page 5.

Name & Picture	Persona Description	Reason
 <p data-bbox="339 616 426 649">Sarah</p>	<p data-bbox="587 291 957 772">Sarah is 22 years old. She has Filipino heritage. Her mother is half Danish and they both moved to Denmark 2 years ago. Sarah is very interested in cultures and languages and she learns Danish easily. She studies service, hospitality and tourism management at Erhvervsakademiet Lillebælt.</p>	<p data-bbox="986 291 1348 515">As a student, she needs to do project work with her group. In order to concentrate, she and her group needs to find a quiet place to work.</p>
 <p data-bbox="316 1126 446 1160">Cameron</p>	<p data-bbox="587 813 954 1108">Cameron is a very energetic and imaginative person. He is 25 and enjoys math and physics. He is studying energy technology. To study effectively he needs peace and quiet.</p>	<p data-bbox="986 813 1316 925">Cameron needs project rooms to work with his group.</p>

## Scenarios

This scenario has two flows: if there is a free room or if there is not. We needed to know how the system would interact with the user and what kind of messages we would have to create for the user to understand. Other scenarios can be found in Appendix; page 6.

### **Reserve a project room scenario:**

#### **A room is available:**

Sarah comes to school on tuesday morning at around 8 am. She meets up with her group, ready to work for the day.

Sarah opens her phone and enters the EAL App. She navigates to the section for project rooms.

Sarah's group has 4 members, including herself. She selects the option to book a new room and fills in the form: number of people, time period and date. She finishes off

by clicking “Reserve room.” After a brief delay, a message appears, notifying which room she has been assigned to.

### **No room is available:**

Cameron is sitting in class. His teacher presents a new project to the class and assigns everyone a group. Cameron’s group meets together, and they decide to start their project work tomorrow and therefore need a room.

Cameron opens the EAL App and navigates to the section for project rooms, selecting the option to book a new one.

Cameron’s group has 14 members, so he types this into the group members field and selects the time-frame: 8:00-16:00 tomorrow. He finishes off by clicking “Reserve room.”

A message appears and informs him that there are no rooms available, and that he is placed in the queue.

## System Artifacts

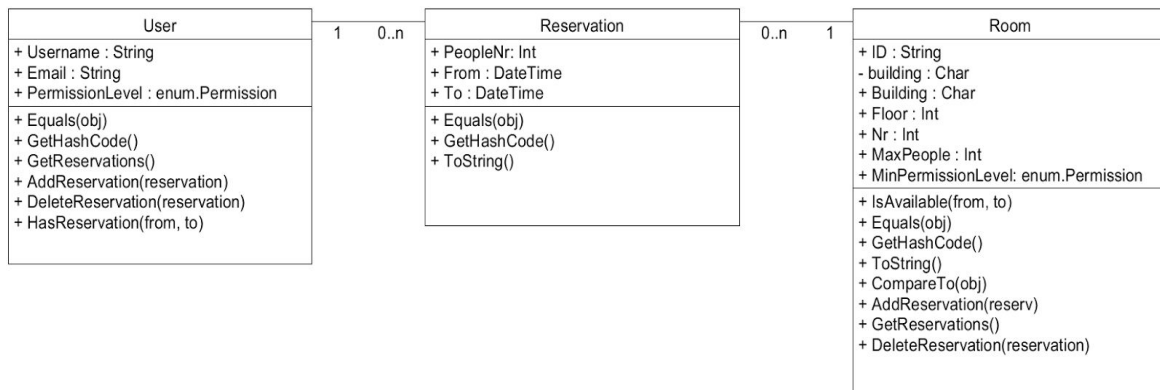
### Domain Model

Our domain is fairly small, and at the beginning of the project we were afraid that it would be too small.

We have three conceptual classes: User, Reservation and Room. Users represent all possible people that can use our system, they will be divided by permission level, which is an enum: Student, Teacher and Admin. Rooms also have a minimum permission level. To reserve a room, the user needs to have an equal or higher permission level than is required by the room, i.e. a room with Teacher permission level can only be reserved by a Teacher or Admin.

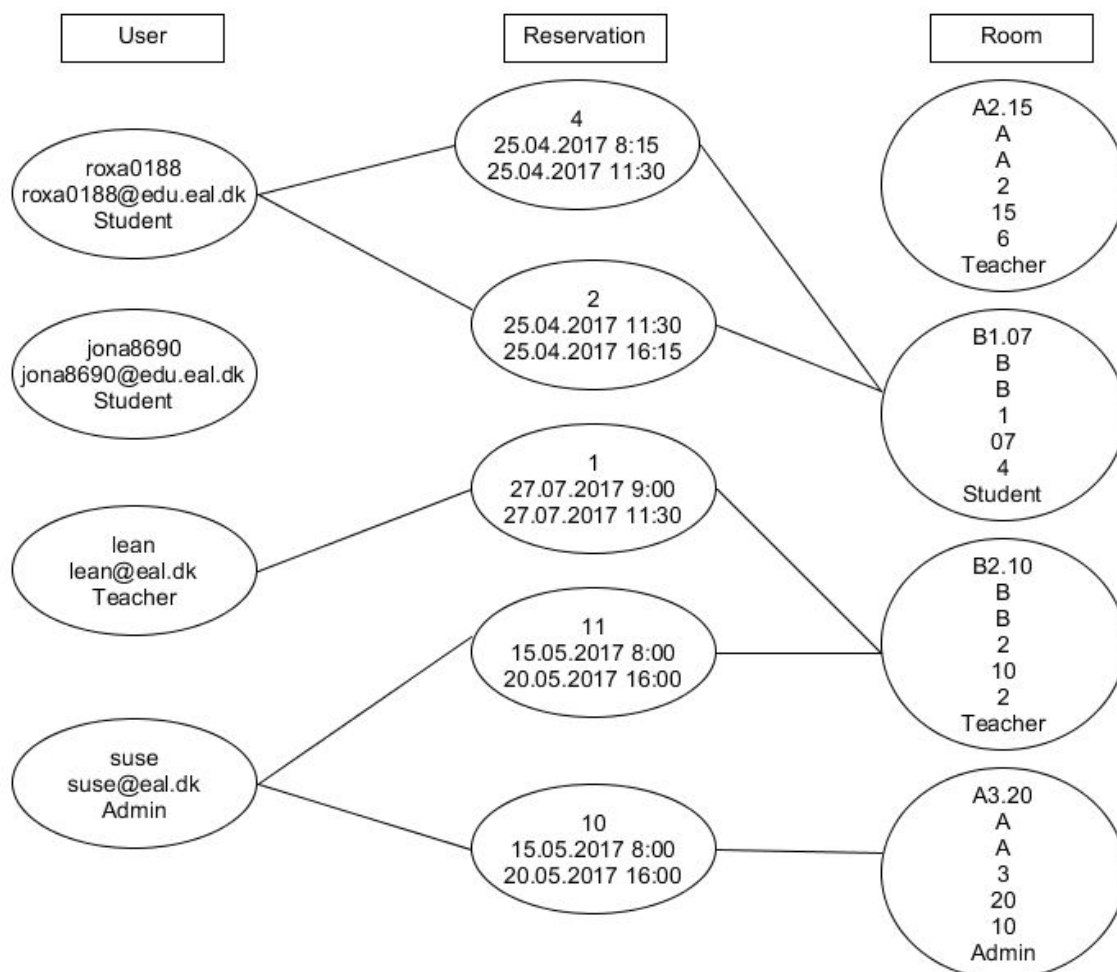
The Reservation is linking both Room and User together, and it has DateTimes that show the period of time in which the reservation will be active as well as the number of people that will be using the room.

Reservation can have only one room and only one user, but rooms can have multiple reservations at different times. Users can have many reservations at the same time, but only if an admin has given the additional reservation(s). The example data will be shown in the object model and database later.



## Object Model

To make sure that our domain is correct and that we understand what kind of data we will need to deal with, we made an object model. By using an object model, it becomes easier for us to follow the connections and to check whether these make sense between our domain classes.



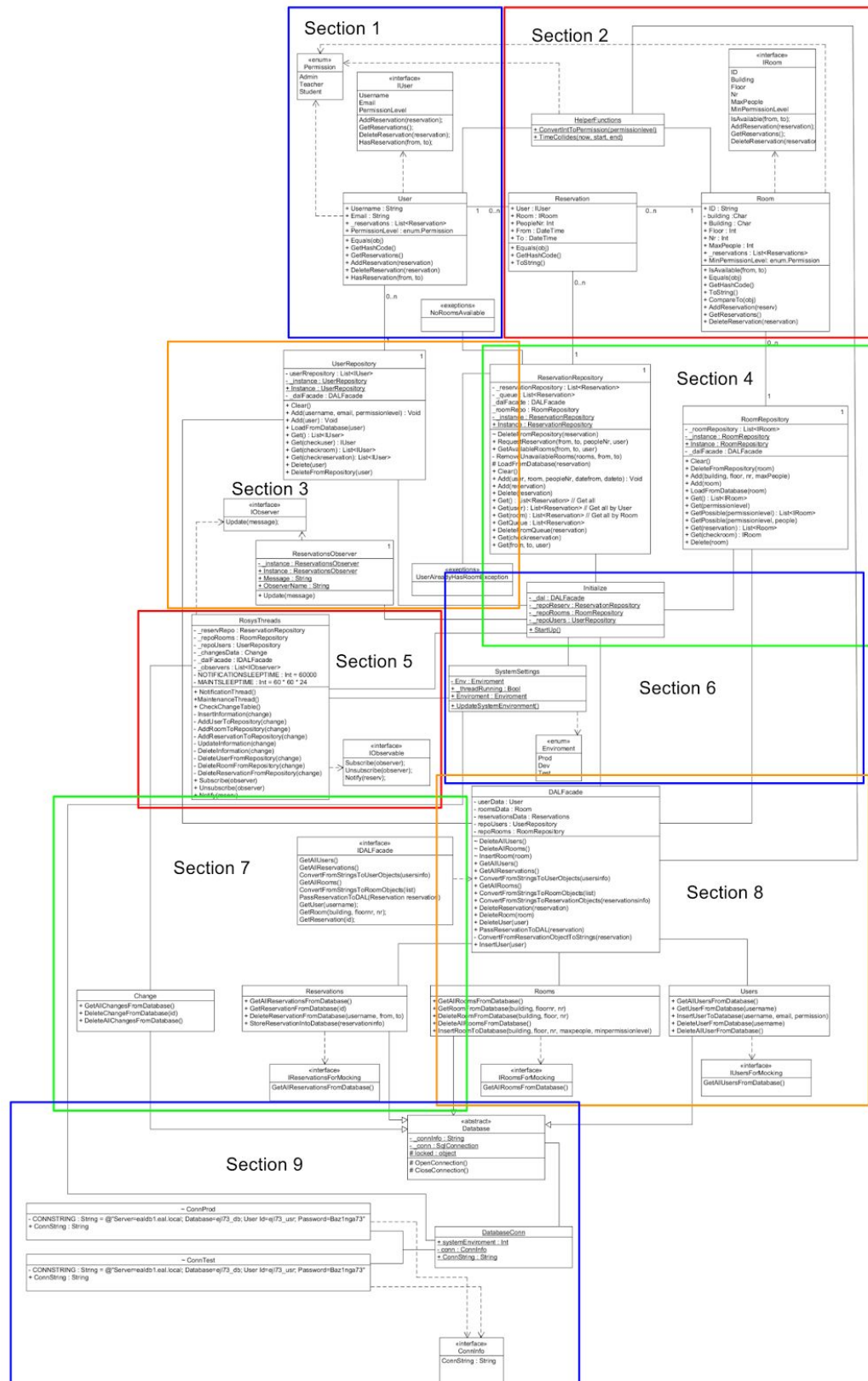


## Design Class Diagram

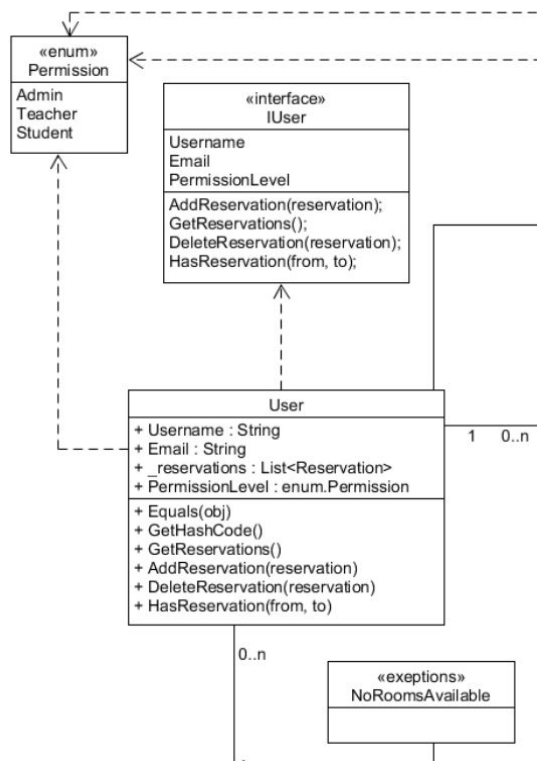
Before starting on the code, we put together a small DCD to help us visualise how the various classes would be interacting. When starting a new use case in the sprint, one of the first tasks would be updating the DCD, and then when creating methods refer to the DCD to ensure our naming scheme was kept.

Later in the project, we mainly used the DCD for references and remembering how the classes were interacting.

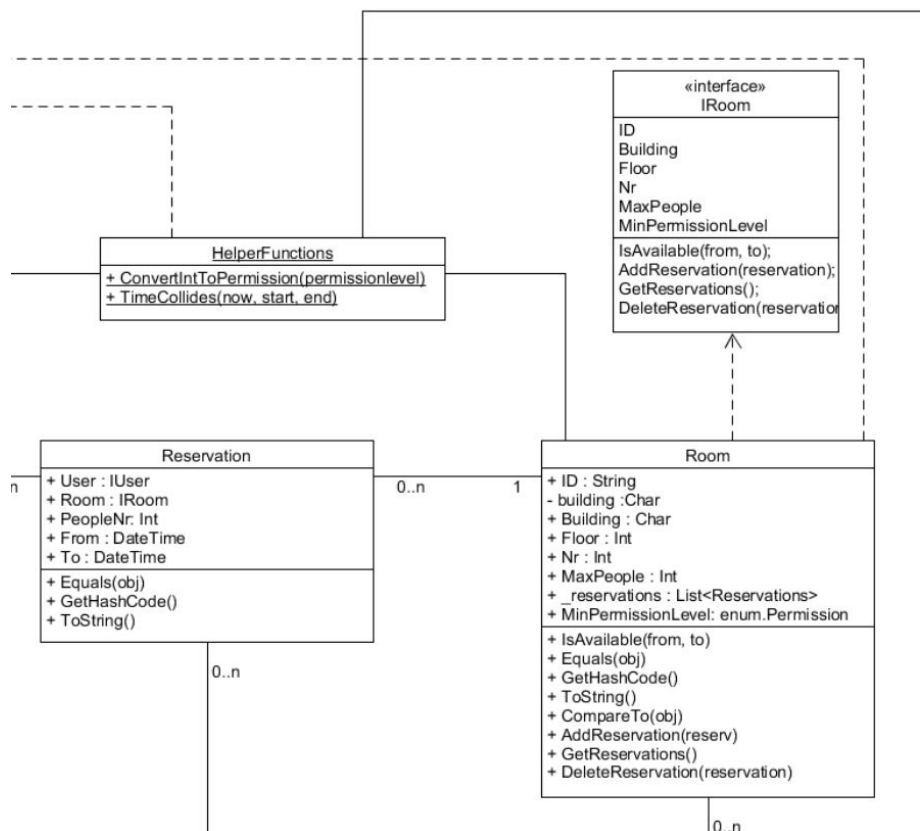
## Overview



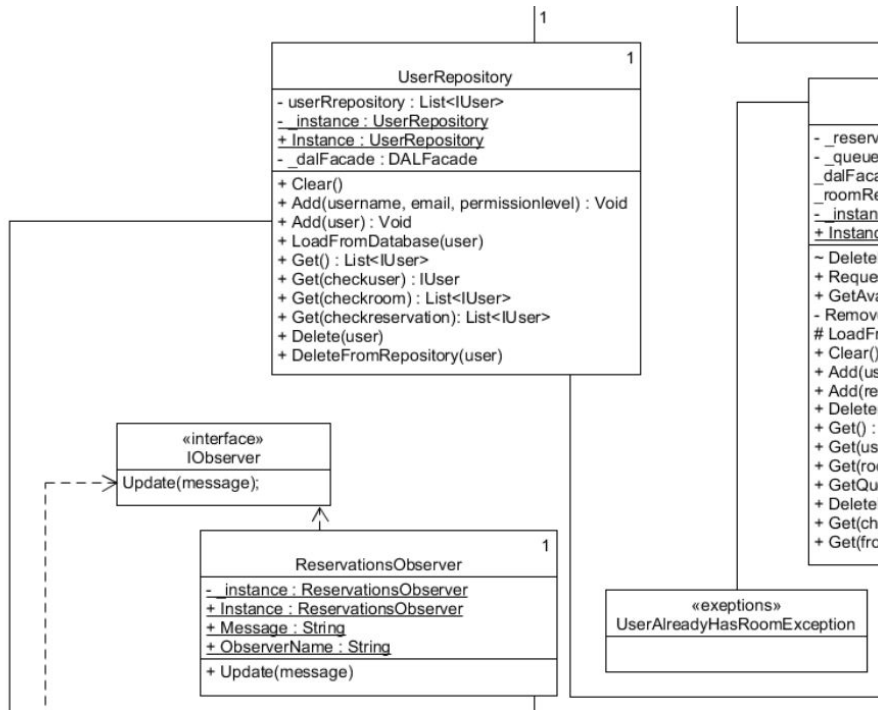
## Section 1



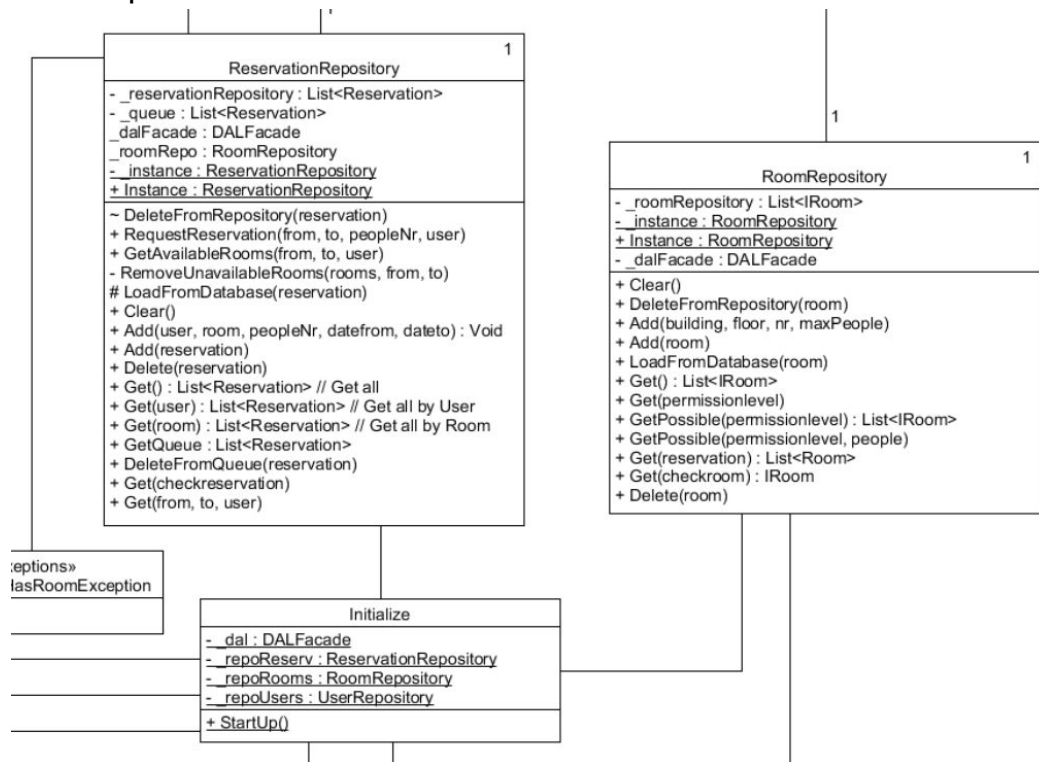
## Section 2



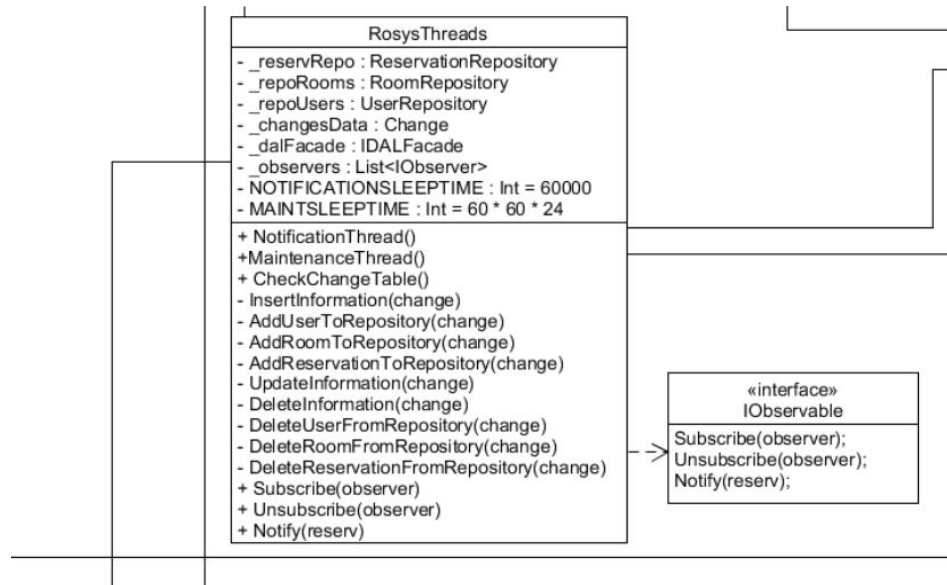
## Section 3



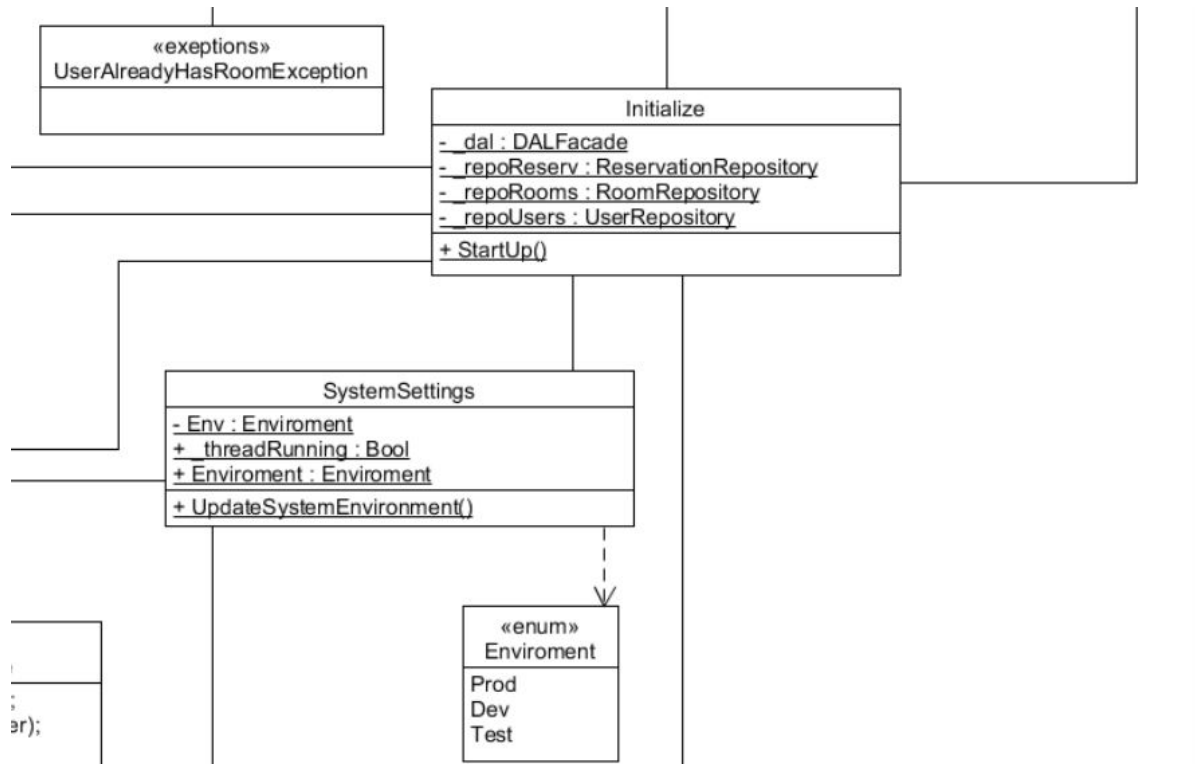
## Section 4



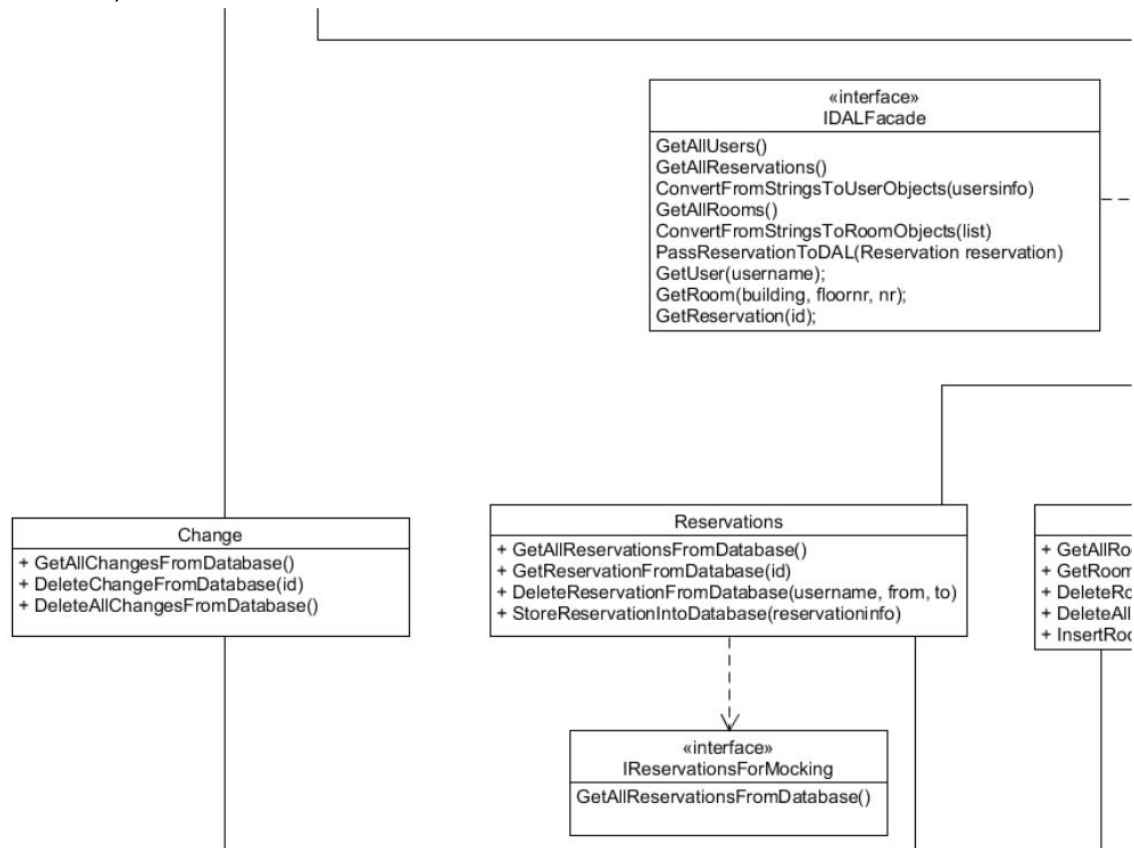
## Section 5



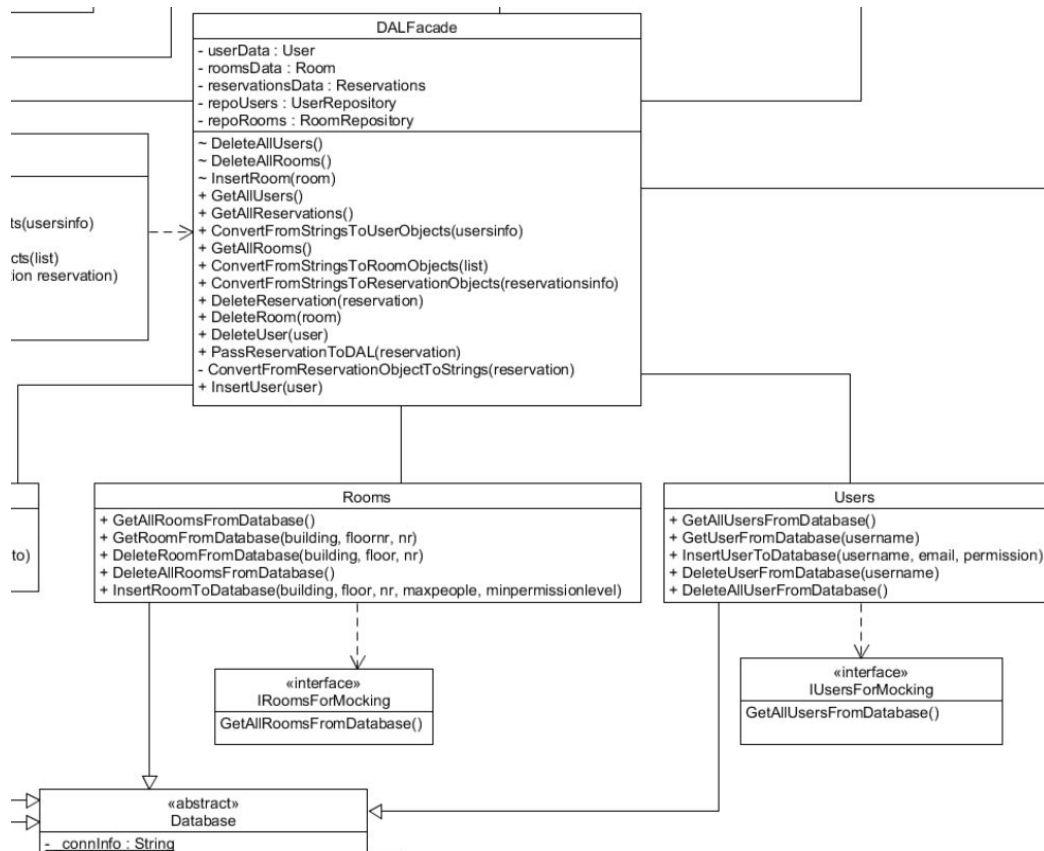
## Section 6



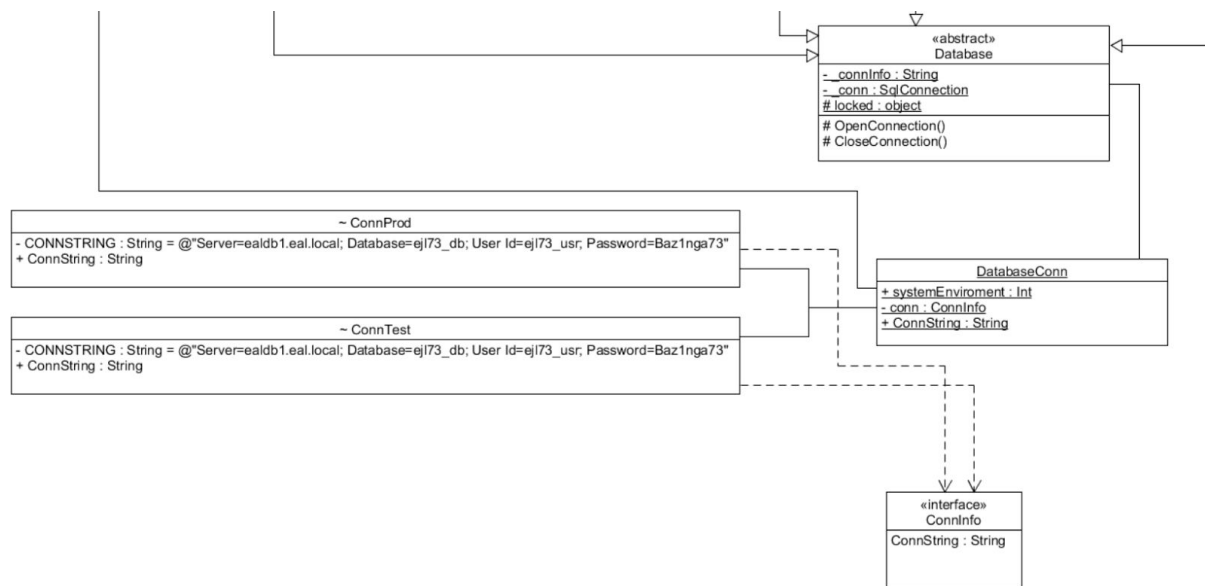
## Section 7



## Section 8



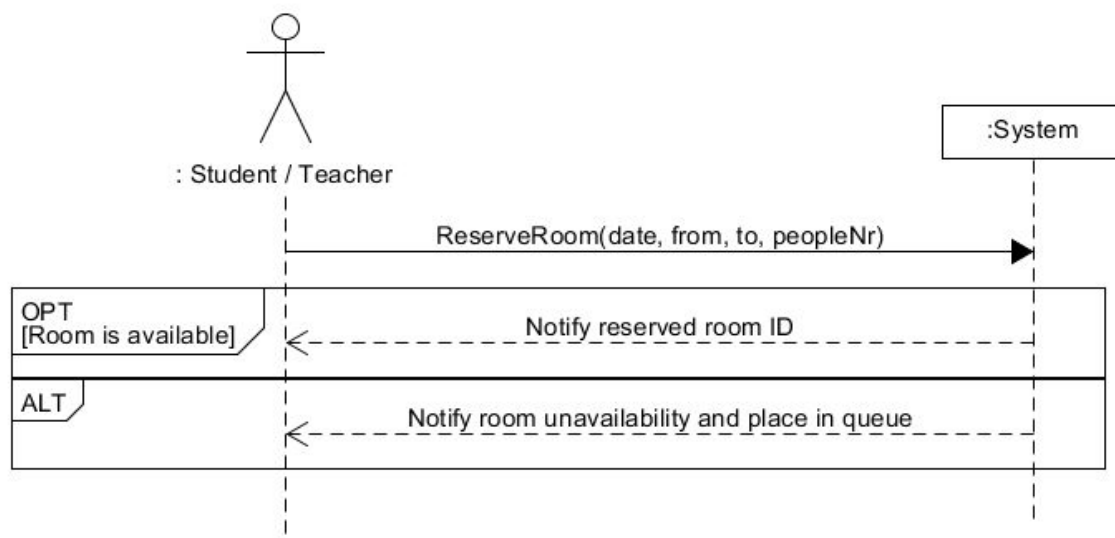
## Section 9



## System Sequence Diagrams

To see what a user would need to input and what he would receive, we made multiple system sequence diagrams. Since we have a graphical user interface and we follow the MVVM pattern, the user interacts with the View section of our code.

More of our SSDs can be found in Appendix page 7.



## Operation Contracts

This is the operation contract for when a client reserves a room in our system. We also have an operation contract for when an admin reserves a room, but it is very similar to this, so we will not show it here. The rest of our operation contracts can be found in the Appendix; page 21.

We made these operation contracts in order to clarify exactly what changes need to be made to the system after this operation has been completed. We also made it so that we would have a reference to see if the operation was working the way we intended it to.

### **Contract C01: Client ReserveRoom**

Operation:

- ReserveRoom(date : string, from : string, to : string, peopleNr : string)

Cross References:

- SSD: SSD\_Reserve
- UC: Client reserves room

Preconditions:

- No preconditions

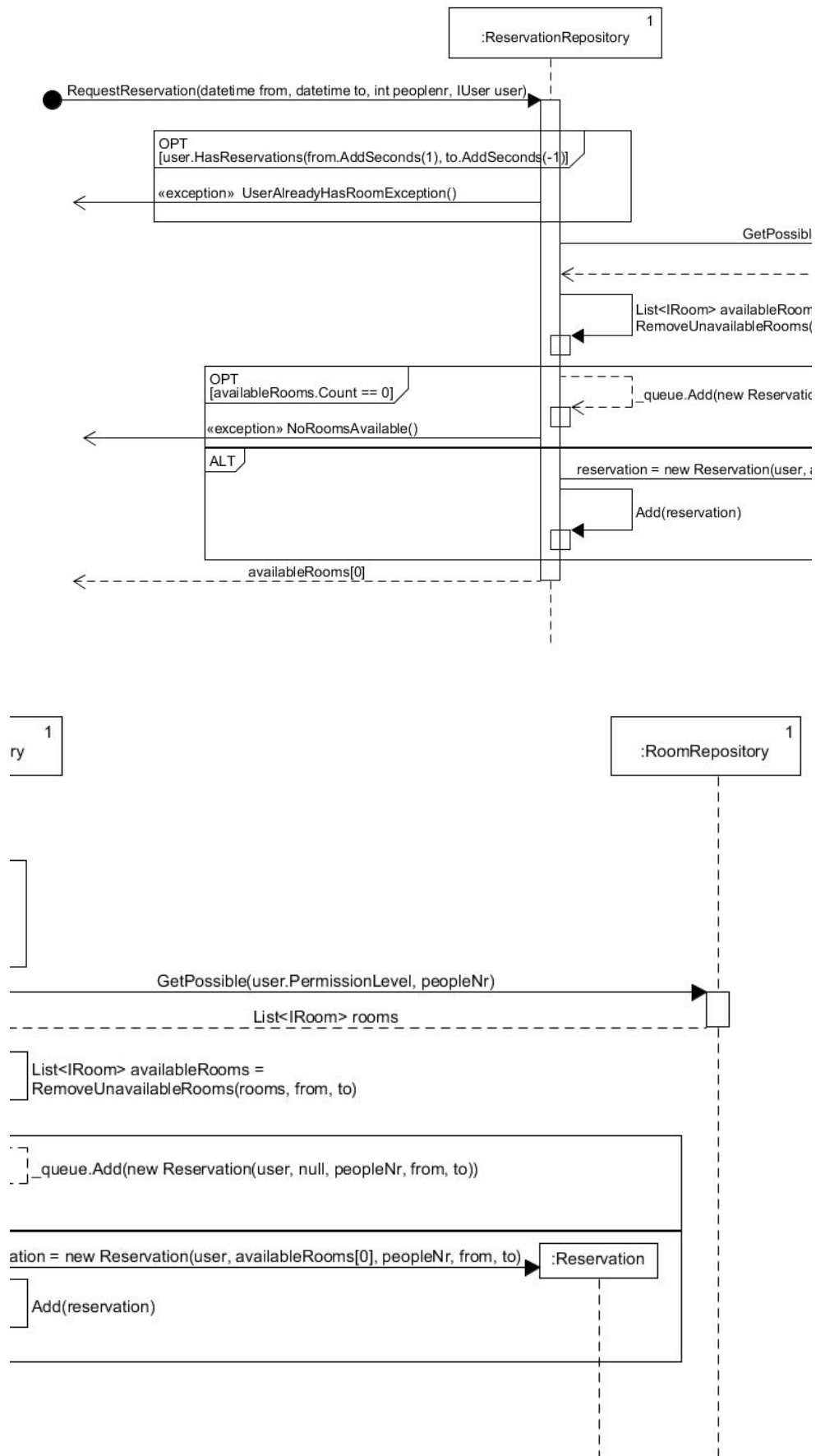
Postconditions:

- A Reservation instance reservation was created
- reservation was associated with a Room
- reservation was associated with a User
- reservation was stored in the database

## Sequence Diagrams

We used our sequence diagrams to help us understand some of the more complicated methods in our program. Such as the RequestReservation, which has some interactions with itself, and others.





# Tests

To assure the quality of code in RoSys we made unit and integration tests that are all currently passing. All tests were made prior to the code implementation. At the start of the project we worked with TDD, but at the end we strayed away from it because of time constraints. This resulted in more mistakes and messier code. All tests have a `ClassInitialize` where we set the Environment, in `SystemSettings`, to `Test`. `ClassCleanup` changes it back to `Production`.

## Summary

**Last Test Run Passed** (Total Run Time 0:00:07,1985033)

✓ 67 Tests Passed

## Unit Tests

The unit tests for the `DALFacade` were created using mocking. The `ConvertFromStringsToUserObjects` method takes a list of string dictionaries as a parameter, which are received from the database. In order for this to be a unit test we needed to isolate the behavior of the method we are testing, so we needed to create a mock of the class in the DAL that returns the list of string dictionaries. We also wanted to guarantee that we know what item will be sent as the parameter.

```

1. [TestMethod]
2. public void GetAllUsersTest()
3. {
4.     DALFacade testDALFacade = new DALFacade();
5.
6.     string username = "hedv0149";
7.     string email = "hedv0149@edu.eal.dk";
8.     string permissionLevel = "0";
9.
10.    List<Dictionary<string, string>> resultUsersInfo = new
    List<Dictionary<string, string>>();
11.    Dictionary<string, string> oneUser = new Dictionary<string, string>();
12.
13.    oneUser.Add("Username", username);
14.    oneUser.Add("Email", email);
15.    oneUser.Add("PermissionLevel", permissionLevel);
16.
17.    resultUsersInfo.Add(oneUser);
18.
19.    IUser expectedUser = new User(username, email, Permission.Student);
20.
21.    var mock = new Mock<DAL.IUsersForMocking>();
22.    mock.Setup(usersMock => usersMock.GetAllUsersFromDatabase()).Returns(()
    => resultUsersInfo);
23.
24.    List<IUser> returnedUsers =

```

```

    testDALFacade.ConvertFromStringsToUserObjects(mock.Object.GetAllUsersFromDatabase());
25.
26.     Assert.AreEqual(expectedUser, returnedUsers[0]);
27. }

```

As an example of our unit tests, we have chosen the RoomIsAvailable test. First, we create two datetimes. Then we run the method that checks if the room is available at that specific time and assign it to boolean, which afterwards is checked by assert is true.

```

1. [TestMethod]
2. public void RoomIsAvailable()
3. {
4.     DateTime from = new DateTime(2017, 05, 01, 13, 0, 0);
5.     DateTime to = new DateTime(2017, 05, 01, 14, 0, 0);
6.     bool roomAvailable = _room1.IsAvailable(from, to);
7.     Assert.IsTrue(roomAvailable);
8. }

```

To see a full example of one of our unit test classes, see Appendix; page 51.

## Integration Tests

In order to test the connection between classes and methods interacting with each other, we have decided to implement integration tests. In total, we have 12 integration tests.

In the WhenRemovingReservationFromRepoRemoveFromRoomAsWell test we start by creating a reservation, adding it to the repository and then deleting it. Next, we get a list of reservations from the room that was used to make the reservation. Finally, we expect that this list will not contain the reservation. We made this test to make sure that if a reservation is being deleted from ReservationsRepository, it is also deleted from other lists in the User and Room classes and specifically in this test, the Room class.

```

1. [TestMethod]
2. public void WhenRemovingReservationFromRepoRemoveFromRoomAsWell()
3. {
4.     Reservation testRes = new Reservation(_student, _room1, 4, _from, _to);
5.     _repoReserv.Add(testRes);
6.     _repoReserv.Delete(testRes);
7.
8.     List<Reservation> reservations = _room1.GetReservations();
9.
10.    Assert.IsFalse(reservations.Contains(testRes));
11. }

```

# Implementation

In the implementation section we will give descriptions of the methods and classes we highlight and give our reasoning for the decisions we made during the implementation of the code.

## Request Reservation

Now we can look at the implementation of the RequestReservation method. This method is, obviously, one of the most important methods in the program, since it does all the work of reserving rooms and reserving rooms is the point of our program.

The first thing we check is if the user already has a reservation as the User class has its own list of reservations that is updated whenever a new reservation is created for that user. If the user does not have a reservation at that time, we get a list of rooms that have a permission level that is less than or equal to the users permission level and that have a people number value that is higher than the one the user inputted. Next we eliminate any rooms that are already booked at that time. The Room class has a method called IsAvailable that checks the list of reservations in the Room class to see if the room has a reservation at that time. If it does not, it gets added to the new list of available rooms. If there are no available rooms, the reservation gets added to the queue with null as the room value, but if there are rooms available, then the reservation gets assigned to the first room in the available rooms list and gets added to the repository.

In the early version of the method we used stacks instead of lists for the rooms. We would pop the stack of rooms and check if the room was available at the specified time. This would be repeated until a room has been found or there are no more rooms in the stack. When we went on to work on the queueing system, having a stack for the rooms became problematic, since it meant we could only look at the first room and would have to get a new stack each time. Having a list was a much easier way to iterate over and find the right rooms for queued reservations.

```

1. public IRoom RequestReservation(DateTime from, DateTime to, int peoplenr, IUser
   user)
2. {
3.     if (user.HasReservation(from.AddSeconds(1), to.AddSeconds(-1)))
4.     {
5.         throw new UserAlreadyHasRoomException();
6.     }
7.
8.     List<IRoom> rooms = _roomRepo.GetPossible(user.PermissionLevel,
   peoplenr);
9.     List<IRoom> availableRooms = RemoveUnavailableRooms(rooms, from, to);
10.
11.    if (availableRooms.Count == 0)
12.    {
13.        _queue.Add(new Reservation(user, null, peoplenr, from, to));
14.        throw new NoRoomsAvailableException();
15.    }
16.    else
17.    {
18.        Reservation reservation = new Reservation(user, availableRooms[0],
   peoplenr, from, to);
19.        this.Add(reservation);
20.        return availableRooms[0];
21.    }
22.
23. }

```

## Initialize

The Initialize class only has one method, `Startup`. This method is the one that is called right after we initialize the main window in the GUI. First we simply create an instance of the `RosysThreads` class, called `threads`, as this will be used later. Then we have `threads` subscribe to the observer so that it can use the observer pattern.

The SD for the `Startup` method can be found in Appendix; page 9.

Next, the users, rooms and reservations are loaded from the database through the `DALFacade` and are stored into lists. These lists are then put into the respective repositories.

We need to get the reservations from the database after storing the users and rooms into the repositories because the reservation has a room and user associated with it, and the user and room come from the repositories. Thus, the user and room repositories must be filled before the reservations can be received from the database.

After the repositories have been loaded, we update the system environment in the `SystemSettings` class so that the connection string is set to the correct database.

Finally, two new threads are created, one thread is the thread that sends notifications to users when they have a reservation in 15 minutes, and the other thread is the

thread that checks for information in the change table. The notification thread and change table will both be covered in greater detail later on in the report. Both threads are set to be background threads so that they will not keep the program from stopping, and then both threads are started.

```

1. public static class Initialize
2. {
3.     private static DALFacade _dal = new DALFacade();
4.
5.     private static ReservationRepository _repoReserv =
ReservationRepository.Instance;
6.     private static RoomRepository _repoRooms = RoomRepository.Instance;
7.     private static UserRepository _repoUsers = UserRepository.Instance;
8.
9.     public static void StartUp()
10.    {
11.        RosysThreads threads = new RosysThreads();
12.        threads.Subscribe(ReservationsObserver.Instance);
13.
14.        List<IUser> users = _dal.GetAllUsers();
15.        List<IRoom> rooms = _dal.GetAllRooms();
16.
17.        foreach (IUser user in users)
18.        {
19.            _repoUsers.LoadFromDatabase(user);
20.        }
21.
22.        foreach (IRoom room in rooms)
23.        {
24.            _repoRooms.LoadFromDatabase(room);
25.        }
26.
27.        List<Reservation> reservations = _dal.GetAllReservations();
28.
29.        foreach (Reservation reservation in reservations)
30.        {
31.            _repoReserv.LoadFromDatabase(reservation);
32.        }
33.
34.        // Tell the DAL what enviroment we're in
35.        SystemSettings.UpdateSystemEnvironment();
36.
37.        Thread notificationThread = new Thread(new
ThreadStart(threads.NotificationThread));
38.        Thread changeTableThread = new Thread(new
ThreadStart(threads.CheckChangeTable));
39.        notificationThread.IsBackground = true;
40.        changeTableThread.IsBackground = true;
41.        notificationThread.Start();
42.        changeTableThread.Start();
43.    }
44. }

```

## Threading

This is one of the threads that was mentioned earlier, namely, the notification thread. It continues running until the `_threadRunning` field in the `SystemSettings` is set to false. The thread gets all the reservations from the repository, then, it iterates through the entire list and checks if any of the reservations are starting in 15 minutes. If there is a reservation in 15 minutes, it will notify the observer, and the observer will then display a message to the user.

When we check if a reservation is due within 15 minutes, we only check the reservation time down to the minute. This is because checking the seconds would require the thread to run every second to ensure we hit the right time-slots. Even then, reservations can only be placed on the minute, and don't currently allow time-slots allocated any further down.

We decided to keep it this way, since handling reservations down to the second would do nothing to further the functionality or usability of the program, but create unnecessary complications in the code.

```

1. public void NotificationThread()
2. {
3.     List<Reservation> reservations = new List<Reservation>();
4.
5.     while (SystemSettings._threadRunning)
6.     {
7.         reservations = _repoReservations.Get();
8.
9.         foreach (Reservation reserv in reservations)
10.        {
11.            DateTime timeCheck = reserv.From.AddMinutes(-15);
12.            if (timeCheck.Date.Equals(DateTime.Now.Date) &&
13.                timeCheck.Hour.Equals(DateTime.Now.Hour) &&
14.                timeCheck.Minute.Equals(DateTime.Now.Minute))
15.            {
16.                this.Notify(reserv);
17.            }
18.            Thread.Sleep(NOTIFICATIONSLEEPTIME);
19.        }
20.    }

```

## Design Patterns

We used design patterns to assist future programmers of our software, in understanding how our code works. Design Patterns are great because one design pattern can be implemented in many different ways depending on the language.

In our program, we implemented 3 common design patterns, namely singletons, facade and factory.

## Singletons - Just a single instance

Our project mainly uses repositories to keep our objects in memory. This meant that all parts of the program needed the same instances of the repositories.

An alternative way this could have been done, is via dependency injection, where you would simply instantiate a class, and by constructors or other methods inject the references into the other classes.

We felt this could potentially cause issues, as it would require a lot of work to go into making sure the reference is kept passed around.

Hence, we used singletons. Singleton is basically a pattern, that when followed, it only allows a single instance of an object to be created. It requires you to close down the constructor, so it can only be instantiated by the class itself. Then via a static property, you can fetch the instance, or create it if it is not yet created.

A full copy of the reservation repository can be found in the Appendix; page 44. A snippet is inserted below, to showcase the singleton.

```
1. private static ReservationRepository _instance = new
   ReservationRepository();
2. public static ReservationRepository Instance { get { return _instance; } }
3. private ReservationRepository() { }
```

*Singleton Pattern used in Reservation Repository*

These 3 lines combined, only allows a single instance of the repository to be used. This ensures that no other copies of the repository can exist.

As the `_instance` is static, the object is created as the program is launched, and is then fetched via the property `Instance`. The private constructor enforces that no other instances can be created, other than by the class itself.

## Factory - Switching on the fly

Because we did not get around to properly mocking our database, when we started running tests, we quickly encountered an issue, that when testing was complete, our databases would have been cleared, and we had to re-create rooms in the program to showcase the features.

We agreed that running the tests against their own database would solve this issue. At this point, the connection string was hard-coded into the abstract Database class. Changing the string was easy, to connect to a different database, but having the program do this in runtime, would require a factory.



We implemented a factory by having multiple classes implement an interface, and then instantiating the different classes depending on a system-set variable. Our factory can be seen below.

```

1. using System;
2.
3. namespace DAL
4. {
5.     internal interface ConnInfo { string ConnString { get; } }
6.
7.     public static class DatabaseConn
8.     {
9.         public static int systemEnviroment = 0;
10.        private static ConnInfo conn;
11.        public static string ConnString
12.        {
13.            get
14.            {
15.                switch (systemEnviroment)
16.                {
17.                    default: throw new IndexOutOfRangeException();
18.                    case 0: conn = new ConnProd(); break;
19.                    case 1: conn = new ConnTest(); break;
20.                }
21.
22.                return conn.ConnString;
23.            }
24.        }
25.    }
26.
27.    internal class ConnProd : ConnInfo
28.    {
29.        private const string CONNSTRING = @"Server=ealdb1.eal.local;
Database=ejl73_db; User Id=ejl73_usr; Password=Baz1nga73";
30.        public string ConnString { get { return CONNSTRING; } }
31.    }
32.
33.    internal class ConnTest : ConnInfo
34.    {
35.        private const string CONNSTRING = @"Server=ealdb1.eal.local;
Database=ejl73_db; User Id=ejl73_usr; Password=Baz1nga73";
36.        public string ConnString { get { return CONNSTRING; } }
37.    }
38. }
39.

```

We decided to put the whole factory in one file, as it was because of how closely related the classes where, and since they were all so small, having it together did not cause confusion.

Now, to execute our tests on another database we would simply have to change the systemEnvironment int. In the future, other connection strings can easily be added to support other states of the program. For example, if a separate database should be used for development.

## Facade - Building Bridges

Facades are a sort of bridge between large classes, or even whole projects. In our case, we put a facade up between our DAL and Core project, since our DAL deals mostly with strings, and our Core with their objects and references.

Hence, we made the DAL Facade, to help translate between this relational model and the object model.

All the methods inside the DALFacade are either converting between types or passing data between the DAL and the Core.

The methods for converting are for converting string values into Reservation, Room or User types and vice versa. The reason we need these is because the databases return values in strings and take parameters as strings but our program uses the objects.

When we first made the method shown below we had some trouble with converting the DateTime values because they were not in the correct format, so we used the StringWriter class to format the date the way it needed to be in the database.

```

1. private Dictionary<string, string>
   ConvertFromReservationObjectToStrings(Reservation reservation)
2. {
3.     Dictionary<string, string> reservationInfo = new
   Dictionary<string, string>();
4.
5.     StringWriter DateToString = new StringWriter();
6.     StringWriter DateFromString = new StringWriter();
7.
8.     string dateTo = reservation.To.Year + "-" + reservation.To.Month +
   "-" + reservation.To.Day;
9.     DateToString.Write(dateTo + " ");
10.    string dateFrom = reservation.From.Year + "-" +
   reservation.From.Month + "-" + reservation.From.Day;
11.    DateFromString.Write(dateFrom + " ");
12.
13.    string hourFormat = "{0:00}:{1:00}:{2:00}";
14.    DateToString.Write(hourFormat, reservation.To.Hour,
   reservation.To.Minute, reservation.To.Second);
15.    DateFromString.Write(hourFormat, reservation.From.Hour,
   reservation.From.Minute, reservation.From.Second);
16.
17.    reservationInfo.Add("PeopleNr",
   Convert.ToString(reservation.PeopleNr));
18.    reservationInfo.Add("DateTo", DateToString.ToString());
19.    reservationInfo.Add("DateFrom", DateFromString.ToString());
20.    reservationInfo.Add("Building",
   Convert.ToString(reservation.Room.Building));
21.    reservationInfo.Add("FloorNr",
   Convert.ToString(reservation.Room.Floor));
22.    reservationInfo.Add("Nr", Convert.ToString(reservation.Room.Nr));
23.    reservationInfo.Add("Username",
   Convert.ToString(reservation.User.Username));
24.
25.    return reservationInfo;
26. }

```

## GUI

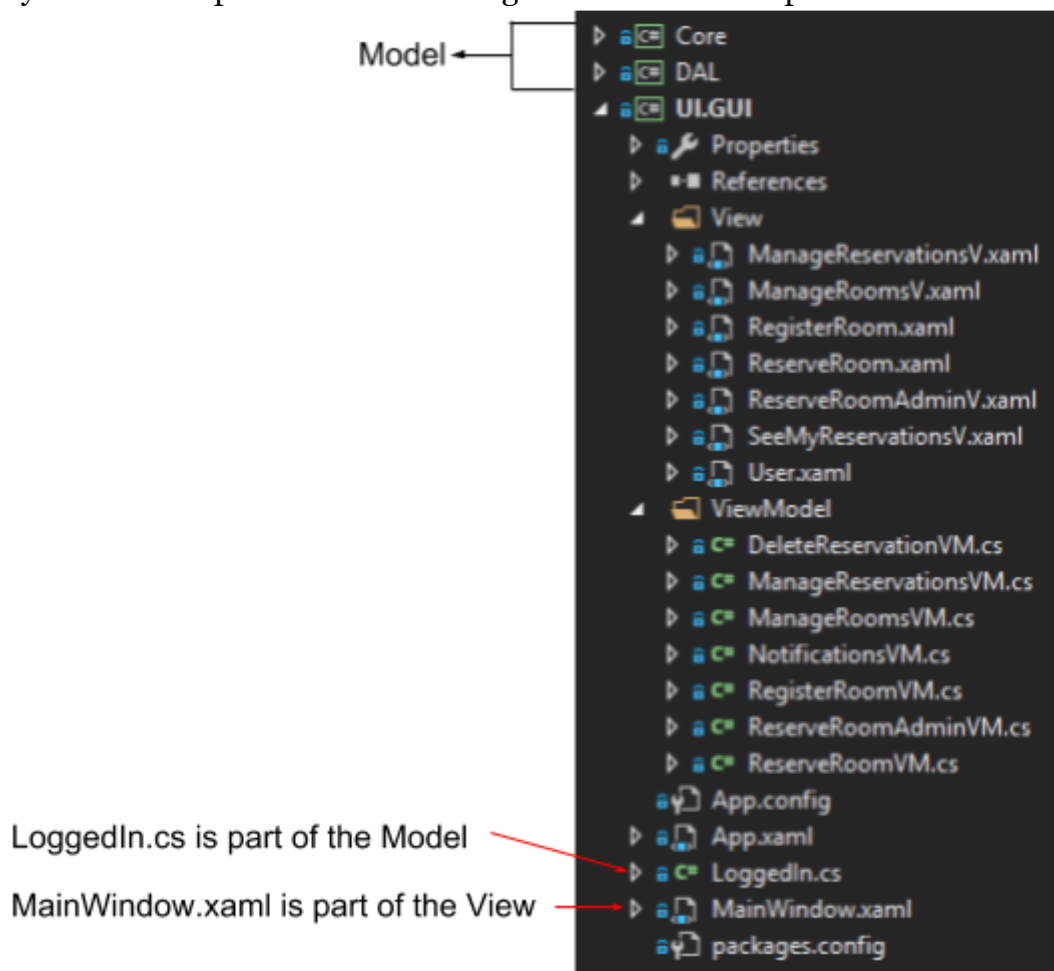
As part of our project, we have created a Graphical User Interface. The main purpose of this is to demonstrate our program's functionalities.

During the process of creating the code and the GUI, we considered it to be a good idea to implement the MVVM (Model–View–ViewModel) Pattern so it would be easier for us to distribute responsibilities between classes and to enable separation between the “View” and the “Model,” so in case our model needs to change, we won't need to change the view or vice versa.

The way we have used MVVM is by putting all of our back-end logic (Core project and DAL Project) in the Model. When we started working on the project, we started by implementing the tests and the code first, before GUI. This is the reason why we have decided not to create a specific “Model” folder and because after we created the GUI, it was obvious that our Core project, respectively DAL project, represents the Model in the pattern.

We continued by creating the GUI, where we created the “View” folder, which is responsible for our program’s layout and appearance. Eventually, we created the “ViewModel”, which was mainly used as a bridge to “translate” between the View and the Model as well as to convert the data objects from these folders and to retrieve and convert the data from the model so it can be easier to handle in the view.

Below you can see a picture of how we organised our MVVM pattern:



Our “View” consists of Pages containing only XAML files and respectively the code behind that does not contain the business logic.

In Appendix; page 24, we have created a Manual for our Product Owner to show him what our user interface looks like, to explain how the GUI functionalities work and how certain methods work.

Below is an example of the XAML code of the “Reserve Room” View.

```

1. <Grid Margin="5,0,-50,0">
2.
3. <Label x:Name="ReserveRoomLabel" Content="Reserve Room" Grid.Column="1"
   HorizontalAlignment="Left" Margin="5,0,0,0" VerticalAlignment="Top" Width="118"
   Height="29" FontSize="16" FontWeight="Bold"/>
4.
5. <DatePicker x:Name="SelectDate" SelectedDate="{x:Static sys:DateTime.Now}"
   Grid.Column="1" HorizontalAlignment="Left" Margin="10,90,0,0"
   VerticalAlignment="Top" Height="27" Width="142" FontStyle="Italic"/>
6.
7. <TextBox x:Name="NumberOfPeopleTextBox" HorizontalAlignment="Left" Height="23"
   Margin="111,41,0,0" TextWrapping="Wrap" Text="" VerticalAlignment="Top"
   Width="62"/>
8.
9. <Label x:Name="NumberOfPeopleLabel" Content="Number of people"
   HorizontalAlignment="Left" Margin="4,36,0,0" VerticalAlignment="Top"
   Height="28"/>
10.
11. <Label x:Name="FromLabel" Content="From" Grid.Column="1"
   HorizontalAlignment="Left" Margin="10,129,0,0" VerticalAlignment="Top"/>
12.
13. <Label x:Name="ToLabel" Content="To" Grid.Column="1" HorizontalAlignment="Left"
   Margin="10,160,0,0" VerticalAlignment="Top"/>
14.
15. <ComboBox x:Name="FromComboBox" Grid.Column="1" HorizontalAlignment="Left"
   Height="22" Margin="52,129,0,0" VerticalAlignment="Top" Width="100">
16.     <ComboBoxItem>9:00</ComboBoxItem>
17.     <ComboBoxItem>9:30</ComboBoxItem>
18.     <ComboBoxItem>10:00</ComboBoxItem>
19.     <ComboBoxItem>10:30</ComboBoxItem>
20.     <ComboBoxItem>11:00</ComboBoxItem>
21. </ComboBox>
22.
23. <ComboBox x:Name="ToComboBox" Grid.Column="1" HorizontalAlignment="Left"
   Height="22" Margin="52,164,0,0" VerticalAlignment="Top" Width="100">
24.     <ComboBoxItem>10:00</ComboBoxItem>
25.     <ComboBoxItem>10:30</ComboBoxItem>
26.     <ComboBoxItem>11:00</ComboBoxItem>
27.     <ComboBoxItem>11:30</ComboBoxItem>
28.     <ComboBoxItem>12:00</ComboBoxItem>
29. </ComboBox>
30.
31. <Button x:Name="ReserveRoomButton" Content="Reserve&#xD;&#xA; Room"
   HorizontalAlignment="Left" Margin="189,107,-30,0" VerticalAlignment="Top"
   Width="136" Height="85" FontWeight="Bold" FontSize="18"
   Click="ReserveRoomButtonClick"/>
32.
33. <Label x:Name="MessageLabel" Content="" HorizontalAlignment="Left"
   Margin="10,224,0,0" VerticalAlignment="Top" Width="315" Height="66"
   FontWeight="Bold" FontSize="14"/>

```

```

34.
35. <Label Content="Select a date" HorizontalAlignment="Left" Margin="4,64,0,0"
    VerticalAlignment="Top"/>
36.
37. </Grid>

```

## Database

The program uses the database, so that it can be shut down and restarted without losing data, while also allowing other programs to easily read data from ours.

Processing speed was the main concern of ours, so we did not wish to query the database for everything, but we wanted to keep our objects in repositories, so our program could easily get these from RAM. Hence our decision to use the database as a “backup,” was in case the program or server would crash or otherwise have to be shut down.

Changes in the program are put into the database via stored procedures stored on the SQL server. Furthermore, as integration is a main concern to us, we thought it would be great if our program could also process changes that happens to our database. We discussed this idea with our project owner, which he seemed to like.

## Normalization

This is our reservations relation:

RESERVATION(Username, Email, PermissionLevel, PeopleNr, DateFrom, DateTo, Building, FloorNr, Nr, MaxPeople, MinPermissionLevel).

In an unnormalized form, our data structure looks like this:

Username	Email	PermissionLevel	PeopleNr	DateFrom	DateTo	Building	FloorNr	Nr	MaxPeople	MinPermissionLevel
roxa	r@eal.dk	Student	2	###	###	B	1	16	2	Student
hedv	h@eal.dk	Student	3	###	###	A	2	49	4	Student
jona	j@eal.dk	Teacher	2	###	###	B	0	12	8	Teacher
matt	m@eal.dk	Admin	3	###	###	B	0	12	15	Teacher
roxa	r@eal.dk	Student	4	###	###	A	2	49	4	Student

From & To are date-time stamps, e.g. “07/04/2017 09:30:00”, but we could not show this here, due to size constraints.

First we determined the functional dependencies:

Email -> (Username, PermissionLevel);  
 Username -> (Email, PermissionLevel);  
 (Username, PermissionLevel) -> Email;  
 (Building, FloorNr, Nr) -> (MaxPeople, MinPermissionLevel).

Then we determined the candidate keys from the unnormalized Reservation table. Since one room can only have one reservation at a specific time, we have this candidate key: (Building, FloorNr, Nr, DateTo, DateFrom).

Since we have determinants that are not the candidate key, we needed to put them into separate tables. The new tables are shown here:

Primary Key / Composite Key: Underlined.  
 USERS(Username, Email, PermissionLevel)  
 ROOMS(Building, FloorNr, Nr, MaxPeople, MinPermissionLevel)

We decided that it makes no sense for us to put PermissionLevel and Email in a different table since it would unnecessarily complicate the user part of our database.

## Final Database Schema

\* Primary Key  
 - Foreign Key

<b><i>Users</i></b>	
<b>Colum</b>	<b>Type</b>
* Username	NVarChar (Primary)
Email	NVarChar
PermissionLevel	int

<b><i>Rooms</i></b>	
<b>Column</b>	<b>Type</b>
* Building	Char (Primary)
* FloorNr	Int (Primary)

* Nr	Int (Primary)
MinPermissionLevel	int
MaxPeople	int

<b><i>Reservations</i></b>	
<b>Column</b>	<b>Type</b>
* ID	Int (Primary / Surrogate)
- Username	NVarChar (Foreign / Users)
- Building	Char
- FloorNr	int
- Nr	int
From	DateTime2
To	DateTime2
People	int

<b><i>Change</i></b>	
<b>Column</b>	<b>Type</b>
* ID	int
Command	int
TableName	NVarChar
PrimaryKey	NVarChar

## The Change table

Tracking changes to the database was a fairly easy and straightforward process, after we discovered the power of triggers. For each table, we created three triggers, one for insert, one for update and one for delete. The pattern for each is the same. Our full SQL for the triggers can be found in our Appendix; page 38.



```

1. CREATE TRIGGER trgInsertReservations ON Reservations
2. FOR INSERT AS
3.     declare @command    int;
4.     declare @table      NVarChar(max);
5.     declare @PK         NVarChar(max);
6.
7.     declare @ID         NVarChar(max);
8.     declare @DateTo     NVarChar(max);
9.     declare @DateFrom   NVarChar(max);
10.    declare @Username    NVarChar(max);
11.
12.    set @command = 0;
13.    set @table = 'Reservations';
14.    select @ID = i.ID from inserted i;
15.    select @DateTo = i.DateTo from inserted i;
16.    select @DateFrom = i.DateFrom from inserted i;
17.    select @Username = i.Username from inserted i;
18.    select @PK = @ID + ';' + @DateTo + ';' + @DateFrom + ';' +
@Username;
19.
20.    INSERT INTO Change(Command, TableName, PrimaryKey) VALUES
21.    (@command, @table, @PK);
22.
23.    PRINT 'Trigger: Insert Reservation'
24. GO

```

The way they all work is simple. Each type of query is given an integer representation. We chose to put insert as 0, update as 1 and delete as 2. Each time a query is run on the tables: Rooms, Users or Reservations, a trigger is also executed, storing information about which command it was (line 12), which table (line 13), and the primary key of the row that was “changed”.

In our program, we have a thread running that reads this change table and processes these changes into our program.

We started encountering various issues with this approach, mainly the issue of our program saving changes to the database. Shortly after having implemented the change table, we started seeing objects being duplicated in our program. This was because our own queries were also executing the triggers, which gave us quite a headache to deal with.

Initially, we tried to put an Identifier key in the change table that would be set depending on if the query was by us, or external. This proved to be just as big a challenge, as we could not always properly update this identifier key and after almost dropping the feature, we found out we could enable / disable the triggers on the go. We updated our stored procedures to simply disable the respective triggers before the query, and re-enable them again afterwards. Because this approach was successful, we were able to finish off this feature.

We also had a problem with the database wanting to change all keys for triggers to integers for some reason. We were unable to find a solution for it other than to change everything to NVarchar.

## Abstracting the Connection

When we started working with the Database, one of the things we realised was how often we had to open and close the connection. We wanted to create multiple files for the various domain resources (users, rooms & reservations) to keep them separate.

This initially meant we had some duplicate code for the opening and closing of the database. We decided to put these in an abstract class, which the individual classes would then implement. We decided to make the Database class abstract, as it has no use by itself, without other code expanding on it.

```

1. using System.Data.SqlClient;
2. using System.Threading;
3.
4. namespace DAL
5. {
6.     public abstract class Database
7.     {
8.         private static string _connInfo = DatabaseConn.ConnString;
9.         private static SqlConnection _conn;
10.        protected static object locked = new object();
11.
12.        protected SqlConnection OpenConnection()
13.        {
14.            Monitor.Enter(locked);
15.            if (_conn == null)
16.            {
17.                _conn = new SqlConnection(_connInfo);
18.            }
19.
20.            _conn.Open();
21.            return _conn;
22.        }
23.
24.        protected void CloseConnection()
25.        {
26.            _conn.Close();
27.            Monitor.Exit(locked);
28.        }
29.    }
30. }
```

Late in the project, we also implemented a factory for the connection string, which we described earlier. Having the open and close methods in this abstract class made implementing a variable connection string much easier.

## Conclusion & Evaluation

Here in the last few days of our project, we started reflecting on what we have done and exactly how we worked. The thoughts we wish to carry with us to the next project are described later.

We feel we did very well in this project. We worked together fairly well and have a decent split of competences that we felt often complimented each other. Naturally, we had our arguments and disagreements, but as long as they are of a constructive, objective nature, we believe this can be healthy for the group work.

In the beginning, we had some issues getting into the SCRUM and sprint habits, but after one or two weeks that came well into gear. Unfortunately, as we came towards the end, sprints became less important, and we simply focused on completing the use-cases, yet we feel we still followed the SCRUM methodology through the majority of the project span.

We ended up finishing a lot more code than we expected, unfortunately, we did stop using TDD at some point, since we started focusing on finishing features rather than SOLID code. Our project owner was very pleased to receive a functioning program.

After the third week, we started doing some daily standup meetings, not that we felt it was necessary, since we would always be together, talking about issues as they arose, but it was a great way to start the day.

We used GitHub pretty well, but we had some issues with merge conflicts, due to some improper branching, other than that we usually made decent commit messages and committed often enough.

## Project Future

During the last meeting with our project owner, we talked a little about the future plans he has for our project. He hopes to have a system implemented by the start of next semester, which will be based largely on our project. The current plan is to integrate it into a website that the school is already operating.

If we ourselves had more time, we would have explored more data binding options for the UI, which would help facilitate updating the User Interface in accordance with our repositories.

We would also like to make our code more SOLID, mainly by abstracting more from the DAL to allow it to be changed out more easily. In addition, we would like to improve on our program by adhering more to the Single Responsibility Principle. Lastly, we would like to employ more dependency injection to allow for easier mocking of the other parts of the code.

We would also like to add the ability to accept or decline the reservation from the 15-minute notification.

## Next Project

For the next project, we want to employ a much stricter TDD process throughout the project. We need to start mocking earlier and plan out the test cases. We should avoid dropping the sprints as this allowed us to get much more disorganised and put less pressure on us to complete our tasks.

As mentioned, we had some issues with merging, so we need to do more research on that, when to branch out and how to properly merge branches.

Regarding sprint's, we need to make the tasks smaller and more detailed. We often ended up with generalized tasks for coding and test making.

Another problem we encountered was our lack of notes, not only from the meetings with our project owner, but also our internal discussions. As we were writing the report, we had trouble remembering the reasoning behind our decisions.