

# **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.

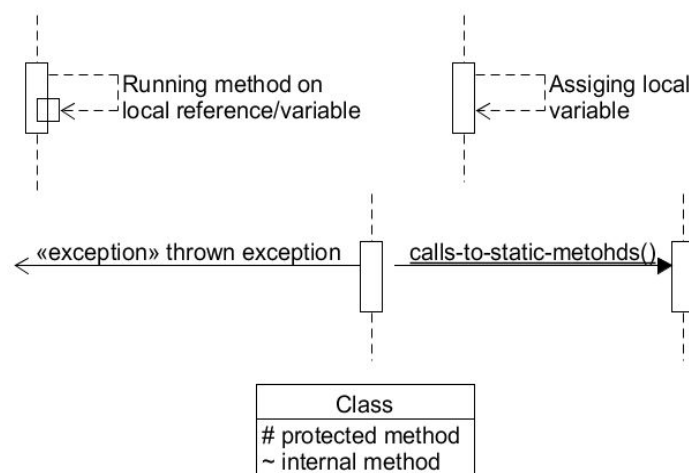
The word “Client” refers to both students and teachers.

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><i><b>Channels</b></i></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><i><b>Cost Structures</b></i></p>  |  | <p><i><b>Revenue</b></i></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.

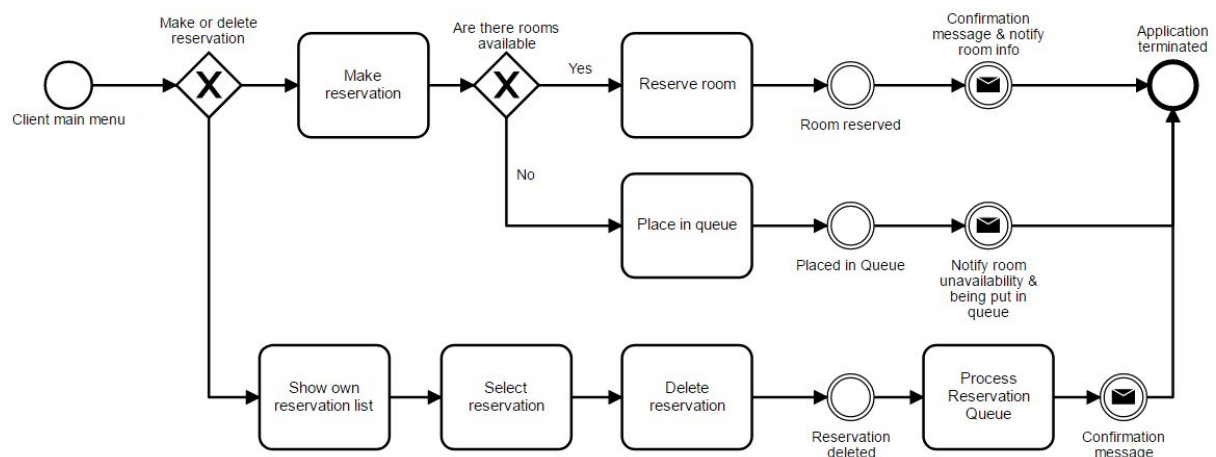
**Interface Requirements** – It must interact with any system the project owner wants to use it with.

**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.

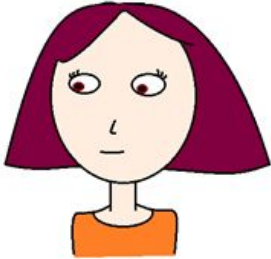

As the name suggests, the use case “Client reserves a room” is responsible for the process of students and teachers reserving a project/meeting room.

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<br/>(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="587 291 959 779">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="984 291 1347 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="587 813 959 1115">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="984 813 1347 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

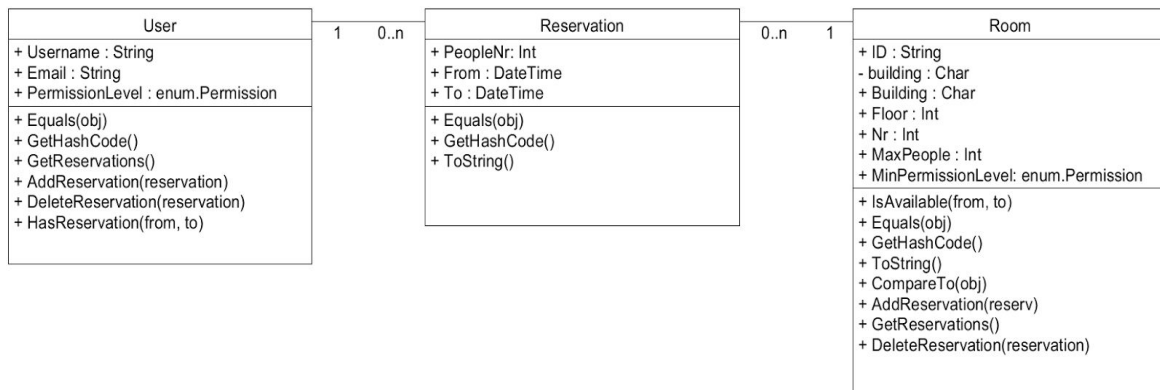
### Domain Model

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.

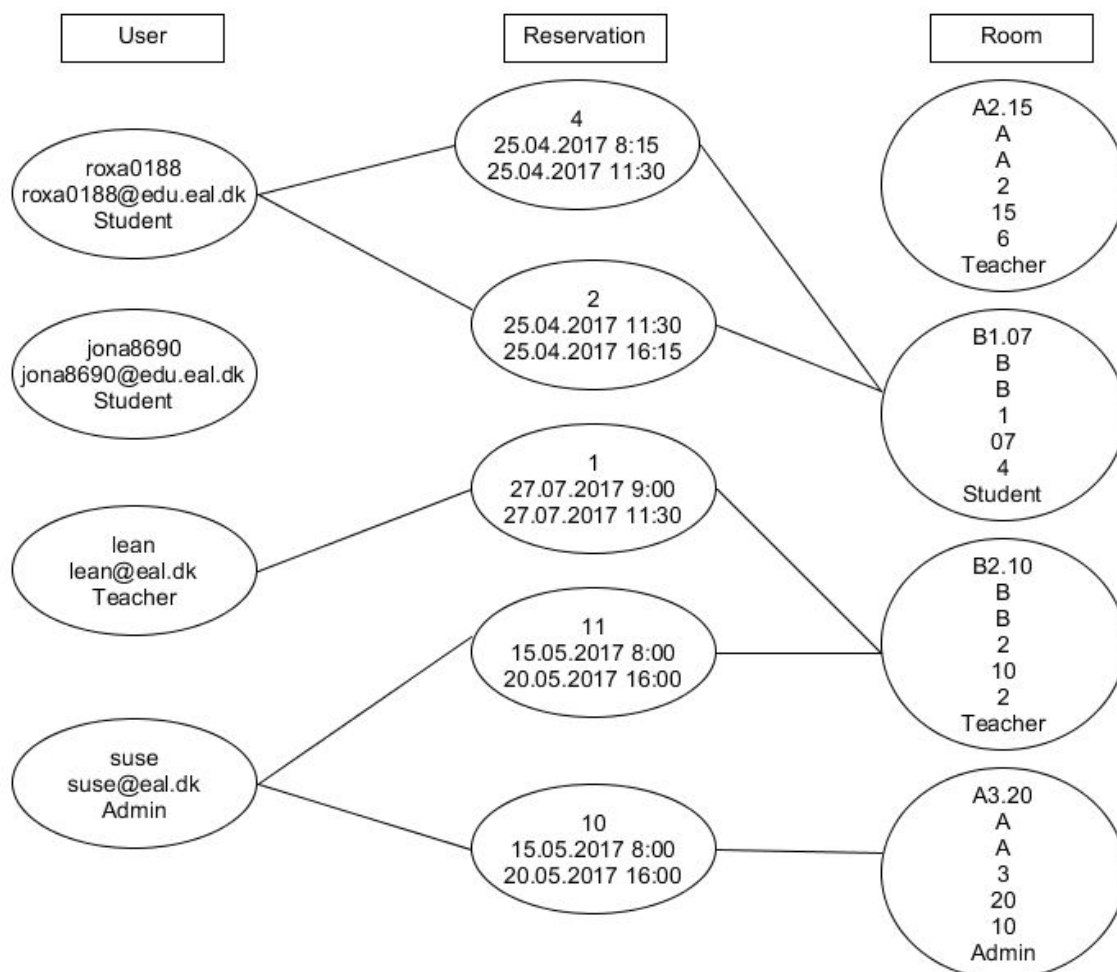
The Reservation is linking both Room and User together, and it has DateTimes that show the period of time in which the reservation will be active as well as the number of people that will be using the room.

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.



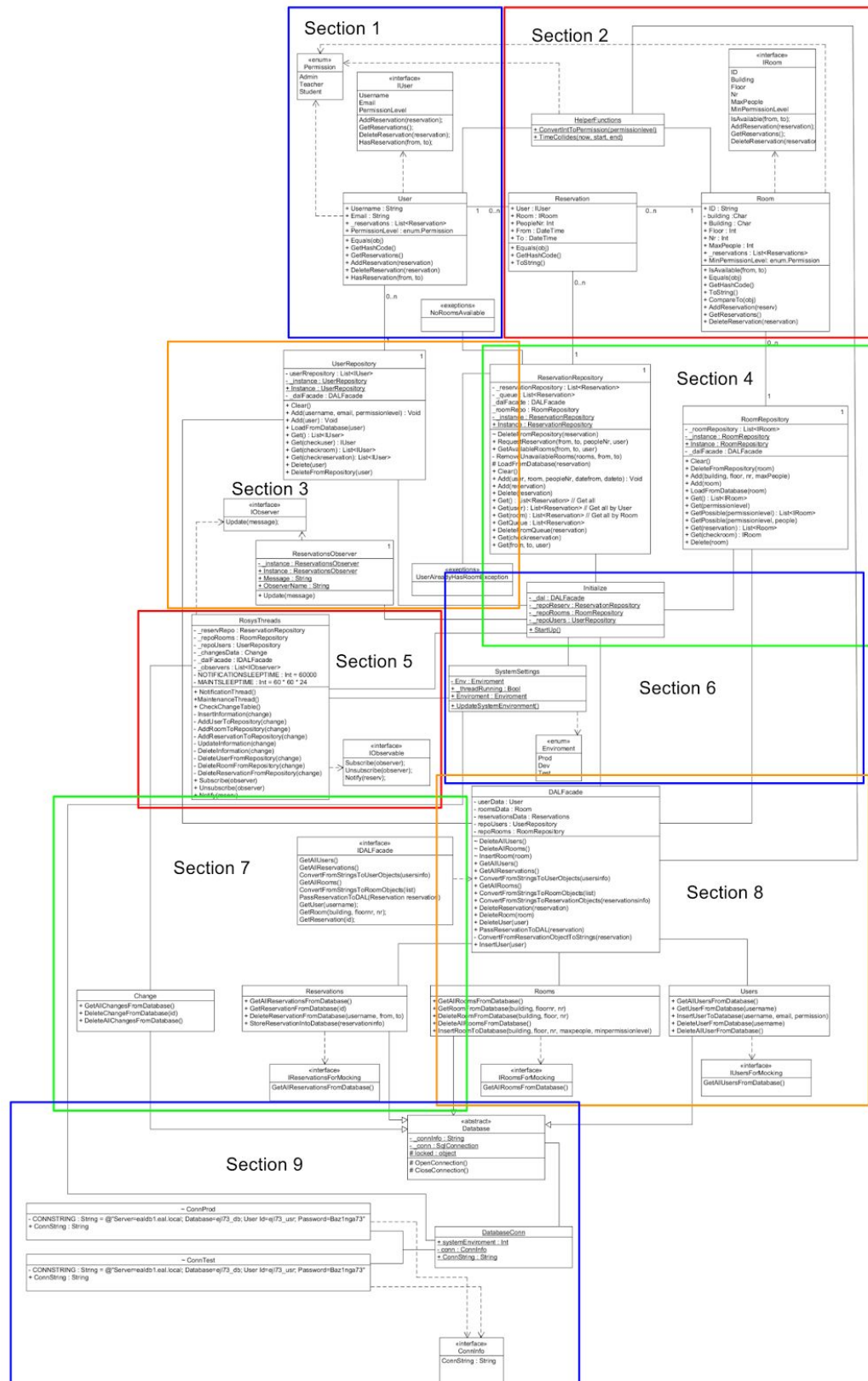


## Design Class Diagram

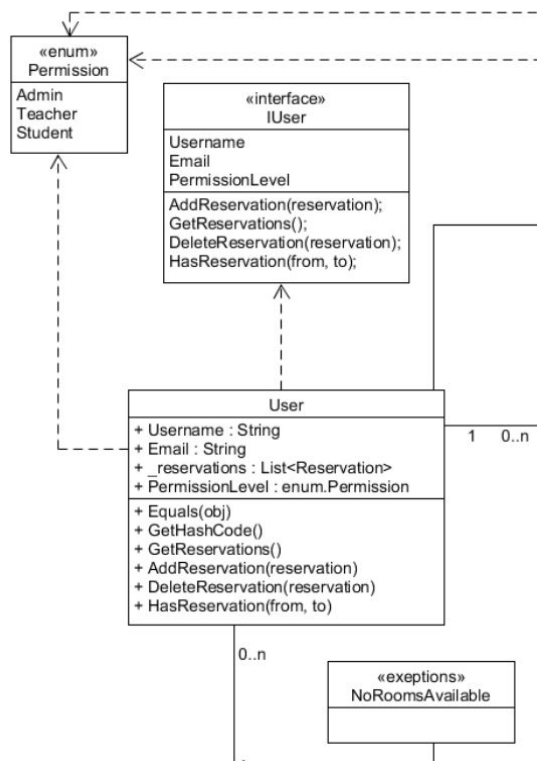
Before starting on the code, we put together a small DCD to help us visualise how the various classes would be interacting. When starting a new use case in the sprint, one of the first tasks would be updating the DCD, and then when creating methods refer to the DCD to ensure our naming scheme was kept.

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

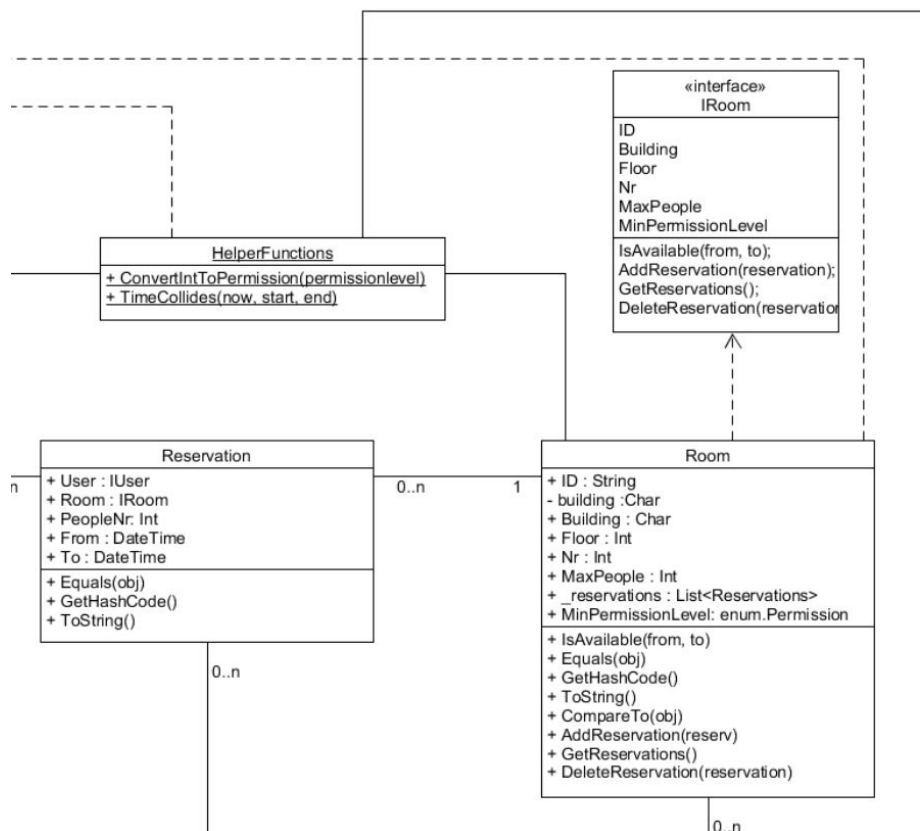
## Overview



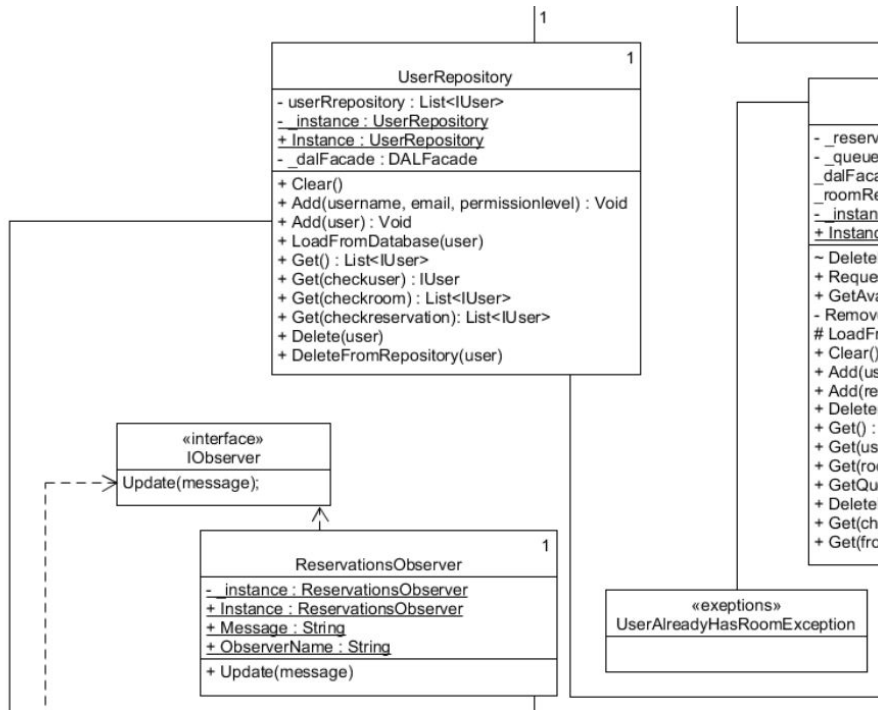
## Section 1



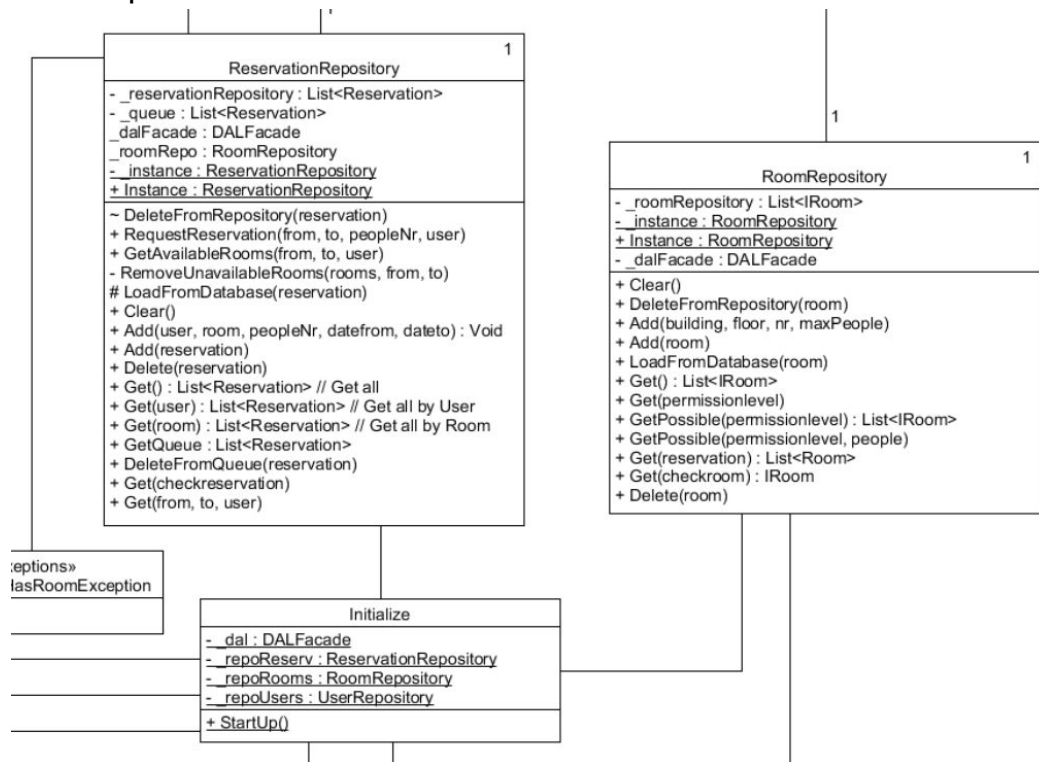
## Section 2



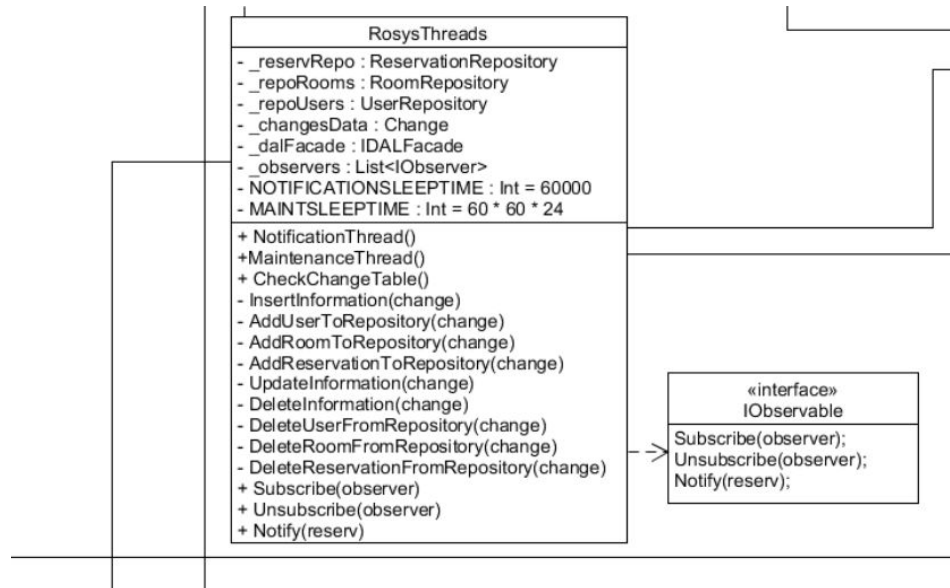
## Section 3



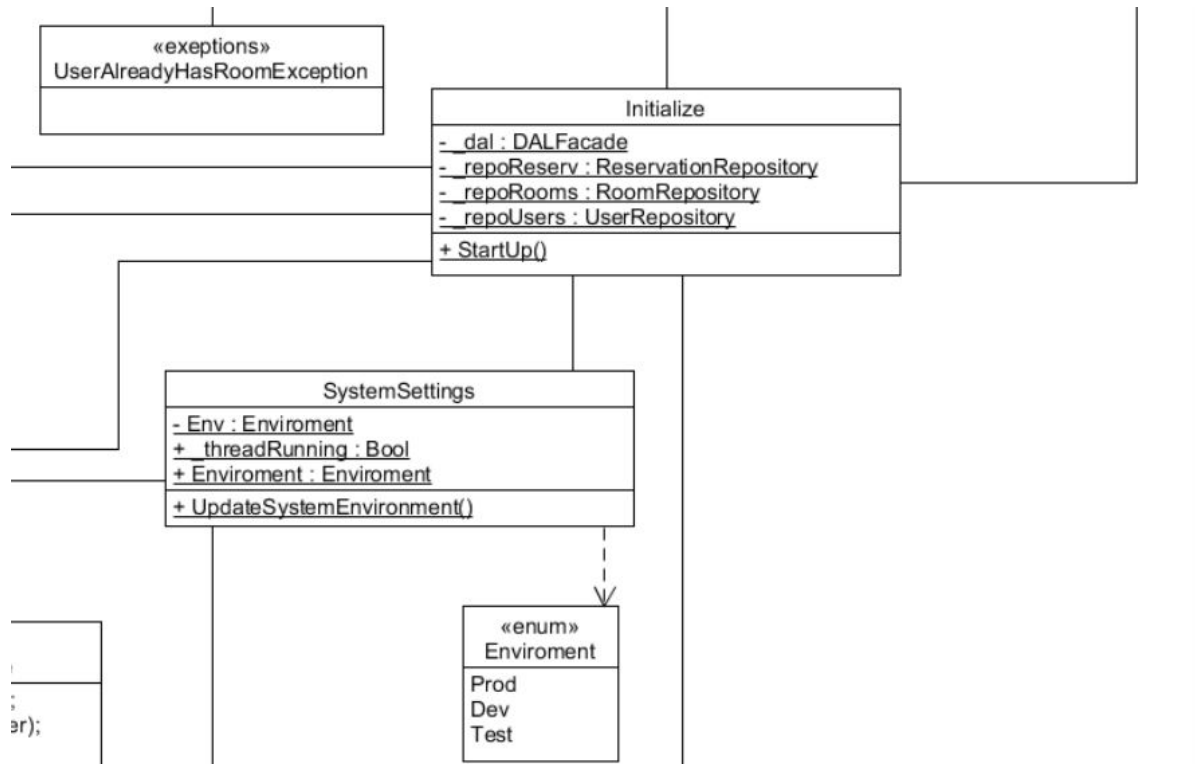
## Section 4



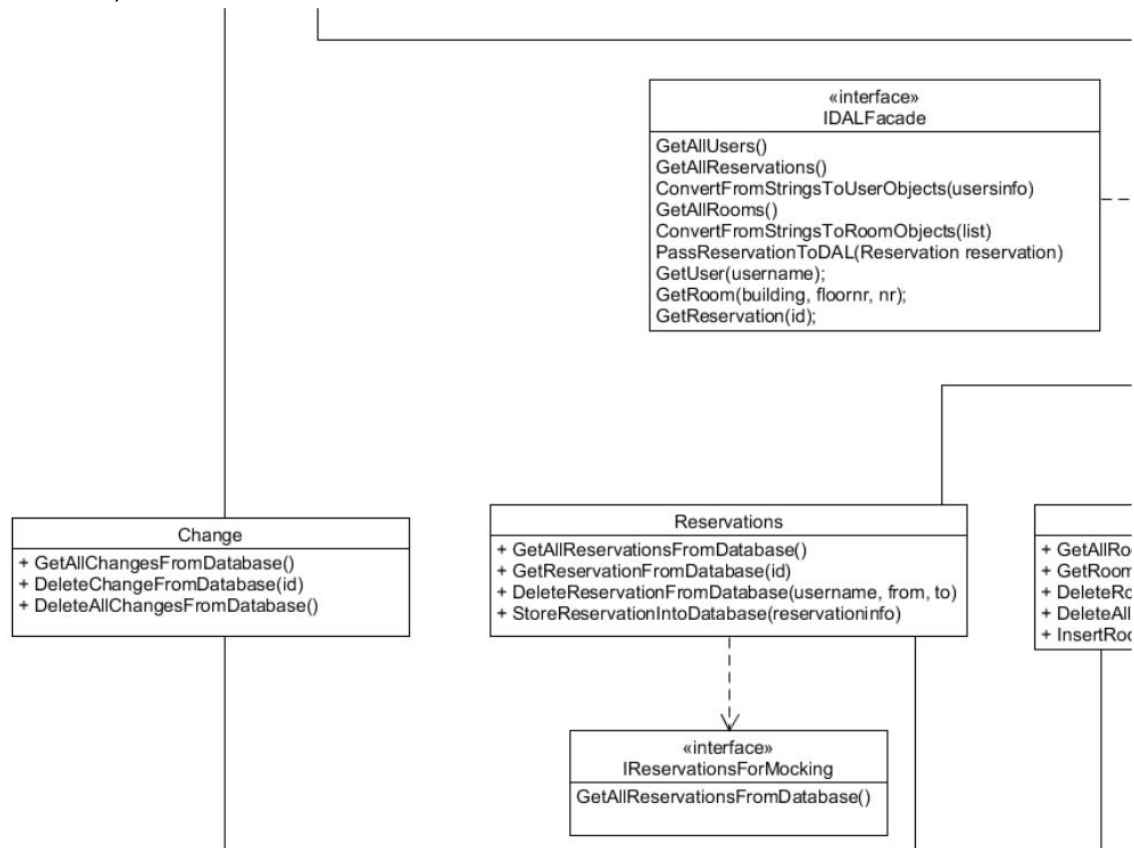
## Section 5



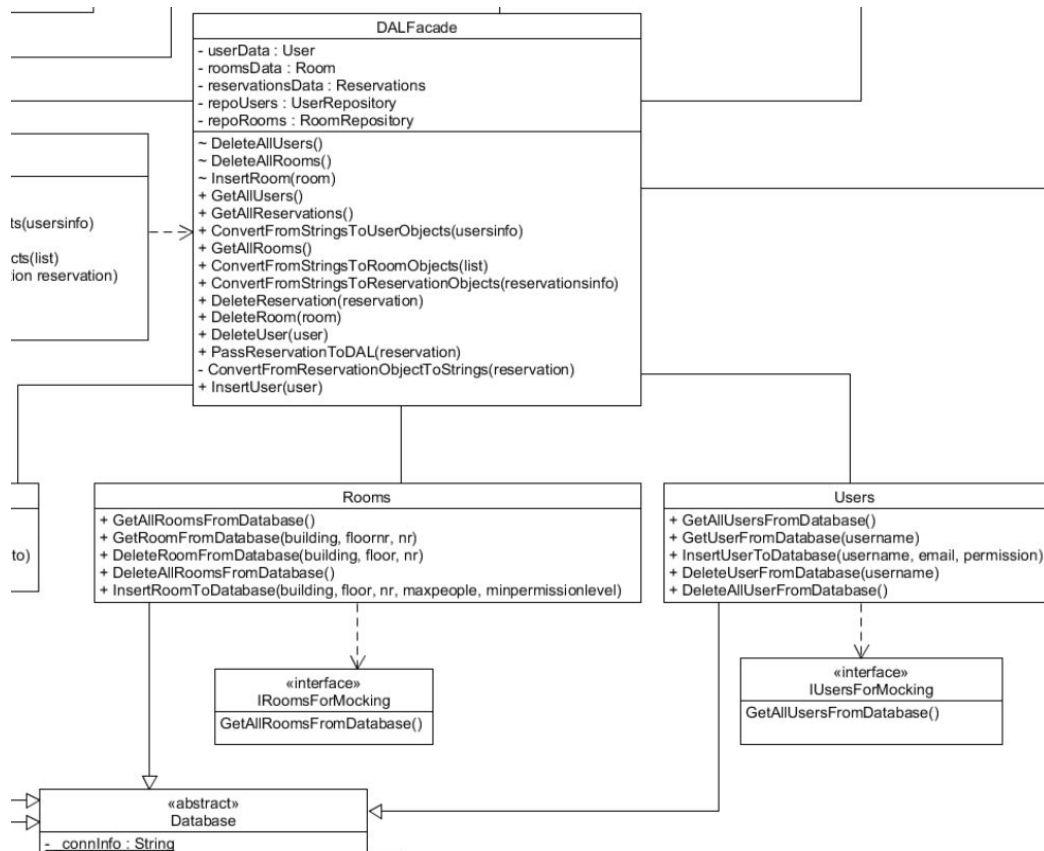
## Section 6



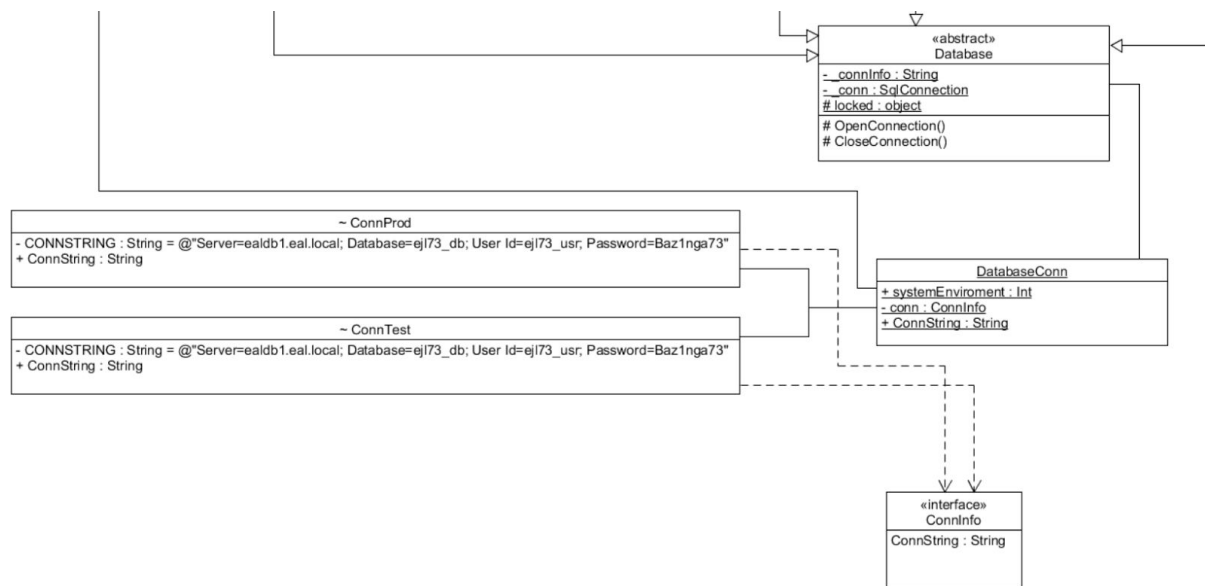
## Section 7



## Section 8



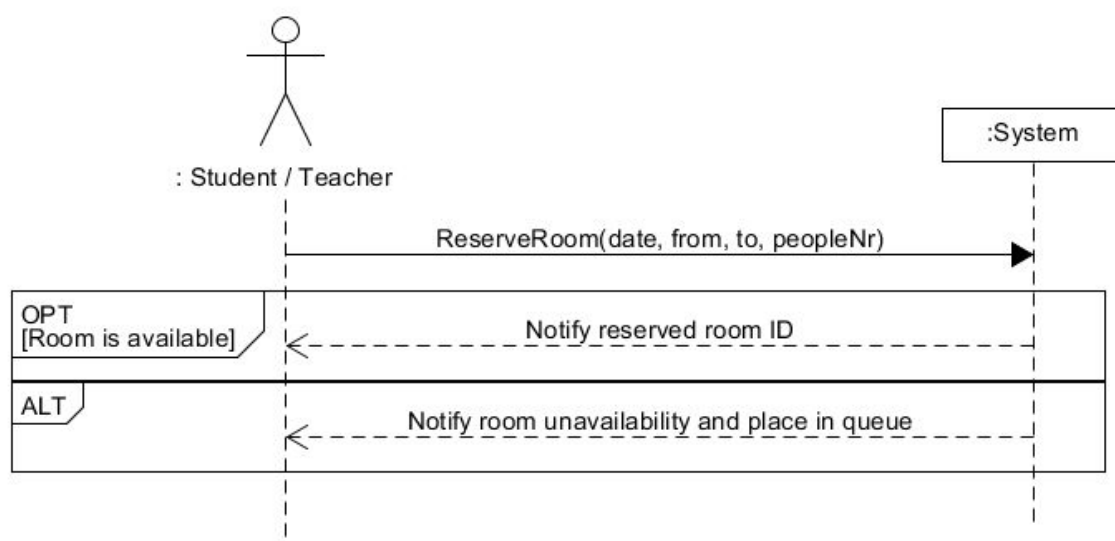
## Section 9



## 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

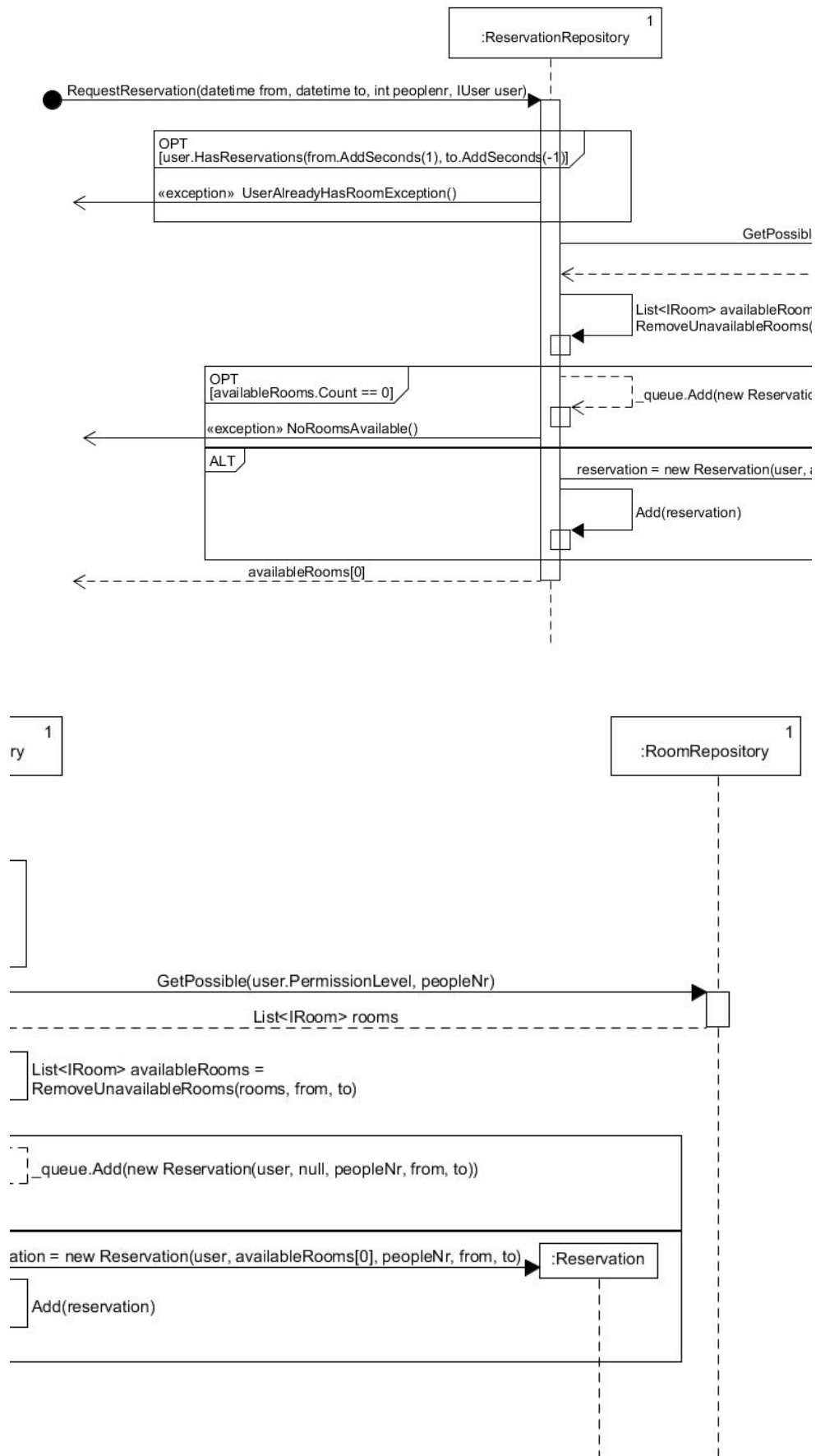
Postconditions:

- 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) &&
timeCheck.Hour.Equals(DateTime.Now.Hour) &&
timeCheck.Minute.Equals(DateTime.Now.Minute))
13.            {
14.                this.Notify(reserv);
15.            }
16.        }
17.        Thread.Sleep(NOTIFICATIONSLEEPTIME);
18.    }
19. }
```

## 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

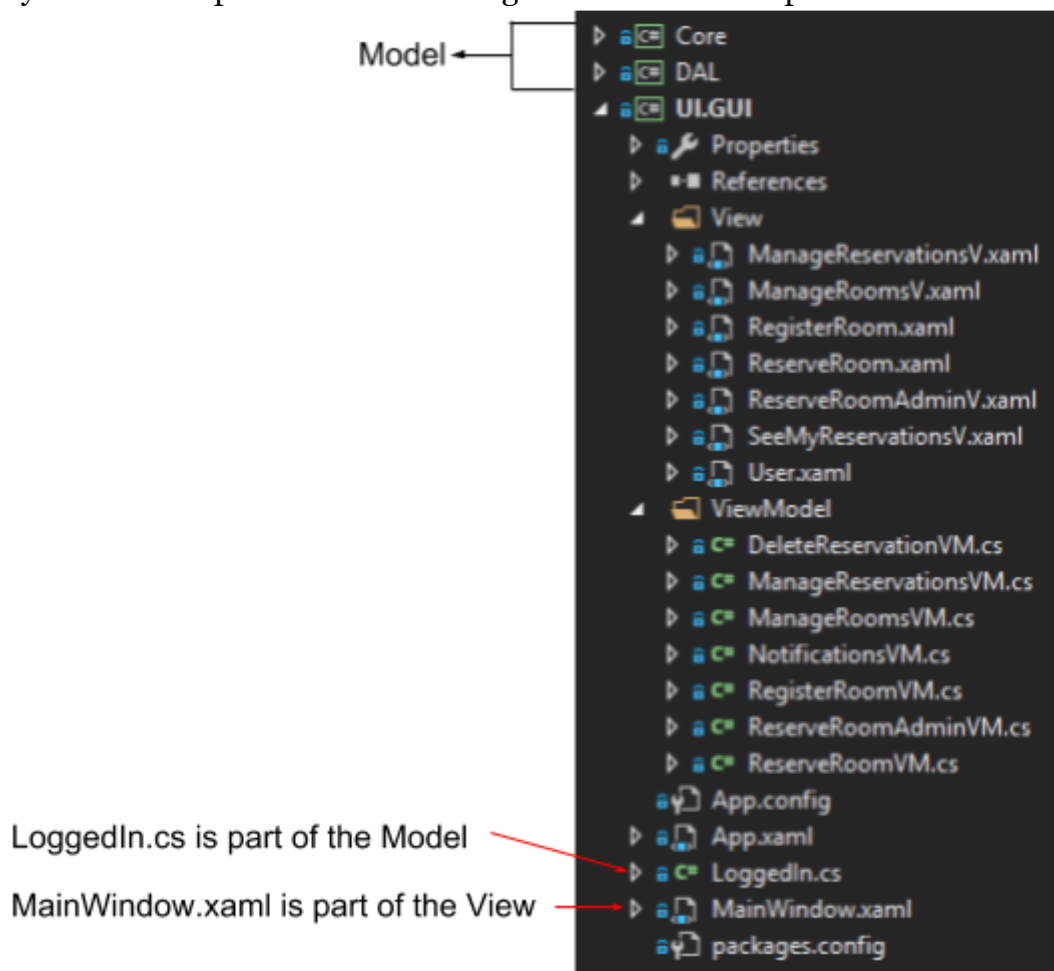
As part of our project, we have created a Graphical User Interface. The main purpose of this is to demonstrate our program's functionalities.

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.

# Appendix

## Table of Contents

|                                 |           |
|---------------------------------|-----------|
| <b>Table of Contents</b>        | <b>1</b>  |
| <b>Program License</b>          | <b>3</b>  |
| <b>Use Cases</b>                | <b>3</b>  |
| <b>Personas</b>                 | <b>5</b>  |
| <b>Scenarios</b>                | <b>6</b>  |
| Admin books a room:             | 6         |
| Admin cancel a reservation:     | 6         |
| Canceling own reservation:      | 7         |
| <b>System Sequence Diagrams</b> | <b>7</b>  |
| Reserve room by Admin           | 7         |
| Delete reservation by Client    | 8         |
| Delete reservation by Admin     | 8         |
| <b>Sequence Diagram</b>         | <b>9</b>  |
| Startup                         | 9         |
| Section 1                       | 10        |
| Section 2                       | 10        |
| Section 3                       | 11        |
| Section 4                       | 12        |
| Section 5                       | 12        |
| <b>Group Contract</b>           | <b>13</b> |
| <b>Business Case</b>            | <b>15</b> |
| <b>Operation Contracts</b>      | <b>21</b> |
| <b>BPMN</b>                     | <b>22</b> |
| Admin                           | 23        |
| <b>Manual</b>                   | <b>24</b> |
| Introduction                    | 24        |
| The User Interface              | 24        |
| Initial Run                     | 24        |

|                             |           |
|-----------------------------|-----------|
| Code                        | 28        |
| Core: DAL Facade            | 28        |
| Core: HelperFunctions       | 29        |
| Core: Initialize            | 29        |
| Core: Reservation           | 29        |
| Core: ReservationRepository | 29        |
| Core: Reservations Observer | 31        |
| Core: Room                  | 31        |
| Core: RoomRepository        | 31        |
| Core: RosysThreads          | 32        |
| Core: SystemSettings        | 33        |
| Core: User                  | 34        |
| Core: UserRepository        | 34        |
| UI.GUI: LoggedIn            | 34        |
| DAL: Change                 | 35        |
| DAL: Database               | 35        |
| DAL: DatabaseConn           | 35        |
| DAL: ConnProd               | 35        |
| DAL: ConnTest               | 35        |
| DAL: Reservations           | 35        |
| DAL: Rooms                  | 36        |
| DAL: Users                  | 36        |
| <b>Database Triggers</b>    | <b>38</b> |
| Reservations Table          | 38        |
| Rooms Table                 | 40        |
| Users Table                 | 42        |
| <b>Code</b>                 | <b>44</b> |
| Core                        | 44        |
| ReservationRepository       | 44        |
| DAL                         | 50        |
| Database                    | 50        |
| Tests > Unit Tests          | 51        |
| Client Create Reservation   | 51        |

## Program License

We decided upon using the MIT License, since it seems to have the best option in case someone wishes to continue working upon our project, they can do so, even for commercial purposes.

The MIT license is an “AS IS” license, so we cannot be held responsible for how our software is used by others, and how bugs in our software might affect them.

## Use Cases

|  |   |
|--|---|
| <b>Use case name</b>                     | Client deletes own reservation  |
| <b>Use case description</b>              | A client wishes to delete a reservation from the system   |
| <b>Actor</b>                             | Client (Students & Teachers)  |
| <b>Pre-conditions</b>                    | Client has reservation  |
| <b>Basic Flow<br/>(Success Scenario)</b> | <ol style="list-style-type: none"><li>1. Client enters reservation system</li><li>2. Client requests the system to display his reservations list</li><li>3. Client selects which reservation he wants to delete</li><li>4. Client requests for reservation to be deleted</li><li>5. System checks reservation queue to see if any of the reservations can have a room</li><li>6. Client receives notifications that cancellation was successful</li></ol> |
| <b>Post-conditions</b>                   | Reservation(s) are removed from repository, Rooms and Users lists and database.   |
| <b>Alternate Flow</b>                    | None significant  |

|                      |                       |
|----------------------|-----------------------|
| <b>Use case name</b> | Admin reserves a room |
|----------------------|-----------------------|

|  |   |
|--|---|
| <b>Use case description</b>              | Admin needs to reserve a room.<br>Admins may need specific rooms for classes and such.  |
| <b>Actor</b>                             | Admin (Reception, administration, e.t.c)  |
| <b>Pre-conditions</b>                    | Available rooms must be displayed in the list box   |
| <b>Basic Flow<br/>(Success Scenario)</b> | <ol style="list-style-type: none"> <li>1. Admin enters the reservation management system</li> <li>2. Admin requests to reserve a room</li> <li>3. Admin selects the date and time for reservation</li> <li>4. System updates the list of available rooms</li> <li>5. Admin types username for the reservation</li> <li>6. Admin selects room for reservation</li> <li>7. Admin requests reservation</li> <li>8. System assigns a room</li> <li>9. Receives confirmation notification that room is reserved</li> </ol> |
| <b>Post-conditions</b>                   | Reservation has been created  |
| <b>Alternate Flow</b>                    | None significant  |



|                             |   |
|-----------------------------|---|
| <b>Use case name</b>        | Admin deletes a reservation             |
| <b>Use case description</b> | An Admin needs to delete a reservation  |
| <b>Actor</b>                | Admin (Reception, administration, etc.) |
| <b>Pre-conditions</b>       | Reservation exists                      |



|  |  |
|--|--|
| <b>Basic Flow<br/>(Success Scenario)</b> | <ol style="list-style-type: none"> <li>1. Admin enters reservation management system</li> <li>2. OPT: Inputs criteria (from date to date, username) to filter list</li> <li>3. Admin selects which reservation he wants to delete</li> <li>4. Admin requests deleting the reservation</li> <li>5. Receives confirmation notification that room is deleted</li> </ol> |
| <b>Post-conditions</b>                   | Reservation has been deleted   |
| <b>Alternate Flow</b>                    | None significant   |

|  |  |
|--|--|
| <b>Use case name</b>                     | Notification System  |
| <b>Use case description</b>              | System is sending a reminder 15 minutes before the client's reservation is beginning.  |
| <b>Actor</b>                             | System   |
| <b>Pre-conditions</b>                    | System must be running   |
| <b>Basic Flow<br/>(Success Scenario)</b> | <ol style="list-style-type: none"> <li>1. System gets upcoming reservations</li> <li>2. System sends reminder for upcoming reservations</li> <li>3. System checks again in one minute</li> </ol> |
| <b>Post-conditions</b>                   | Reminder notification was sent   |
| <b>Alternate Flow</b>                    | AF1: No upcoming reservations<br>1. System checks again in one minute  |

## Personas

|  |  |   |
|--|--|---|
| <br>Moira | Moira is 58 years old and works for Erhvervsakademiet Lillebælt. She is good with computers. She likes to help people, and she works in information at the Odense campus. Moira has a degree in economics. | Moira, as admin of the system, can book any room in the school. Her job is also to cancel unused reservations and book classrooms for lessons and lectures. |
| <br>Leo   | 47-year-old Leo is a teacher in Lillebaelt Academy. He likes news and politics. He has a goldfish named Eko. 10 years ago, he had a nose operation but he doesn't like to talk about it.                   | As a teacher, he can reserve teacher rooms, such as meeting rooms, project rooms and others.  |

## Scenarios

### Admin books a room:

Moria has been tasked with booking a meeting room for some teachers.

She enters the booking system, types in when the booking needs to be, and selects which room the booking should be valid for.

She hits the button to book the room, and a message appears that confirms the booking.

### Admin cancel a reservation:

Moria is setting at the front information desk when a student approaches.

Despite the new booking system having launched, someone has booked a project room and accepted the booking by the room, yet has still left it empty.

Moria enters the admin side of the reservation system. She finds the room which is empty and marks it. She then clicks the "Delete Reservation" button.

Moria informs the student that the standing reservation has been cancelled, and asks the student to perform the new reservation themselves.

## Canceling own reservation:

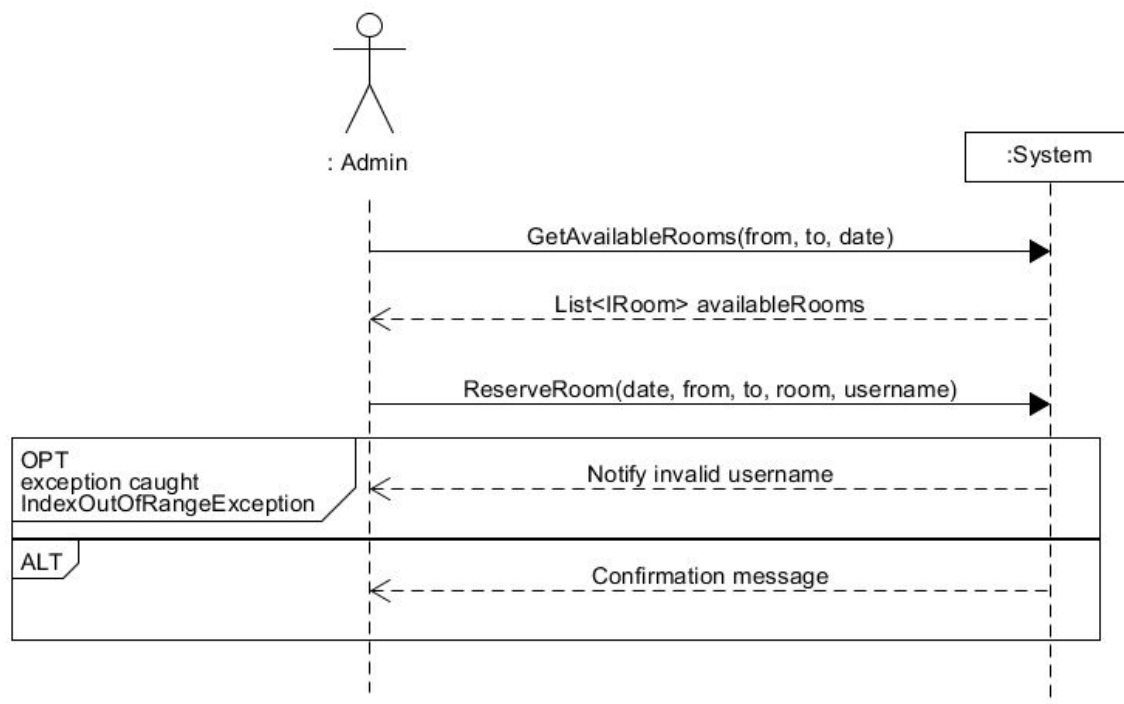
Leo was arranging a meeting between some teachers, but something has happened and he no longer needs his booked meeting room.

Leo opens up the booking system and enters the section to list own reservations. He marks the reservation for the room he no longer needs, and clicks "Delete Reservation".

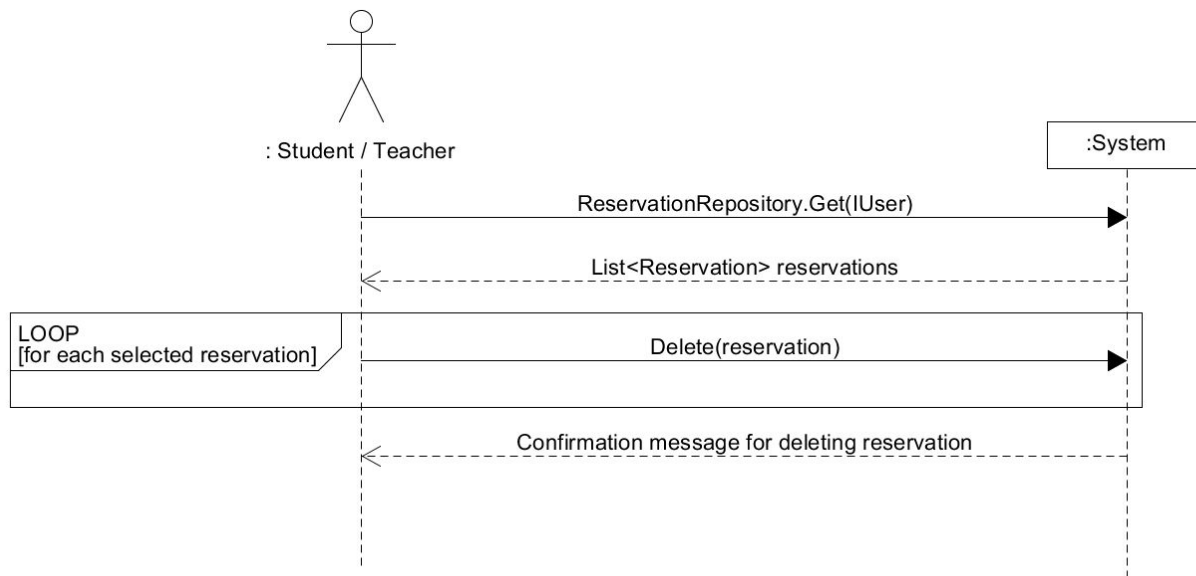
A message ensures Leo that his reservation has been cancelled.

## System Sequence Diagrams

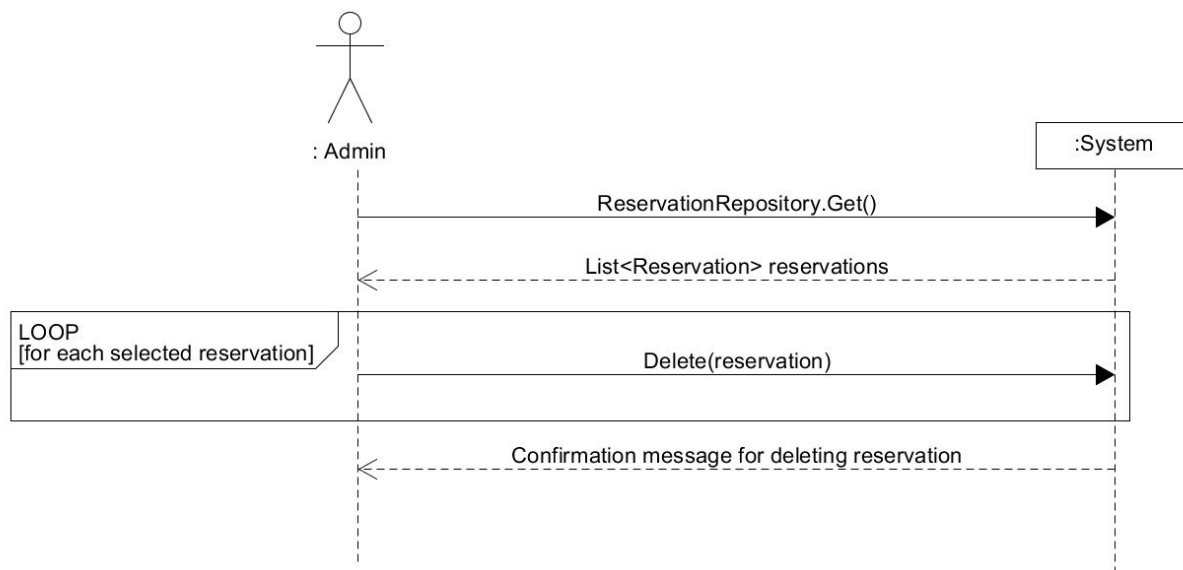
### Reserve room by Admin



## Delete reservation by Client

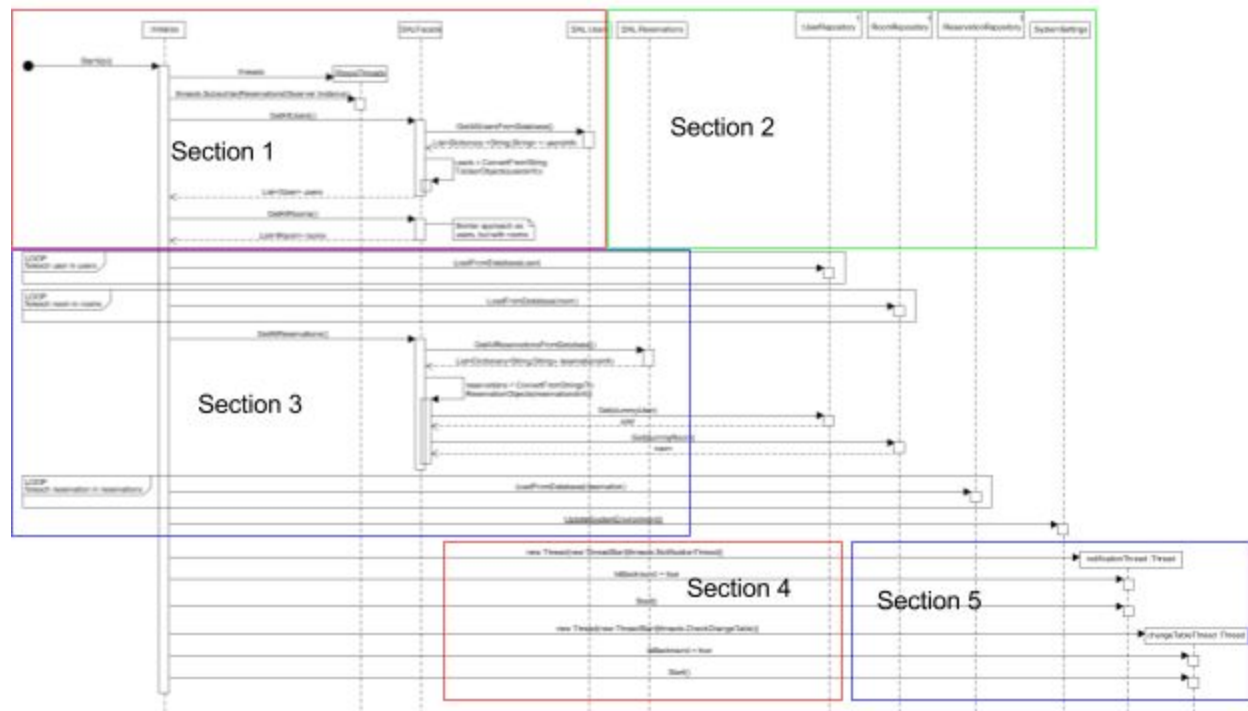


## Delete reservation by Admin

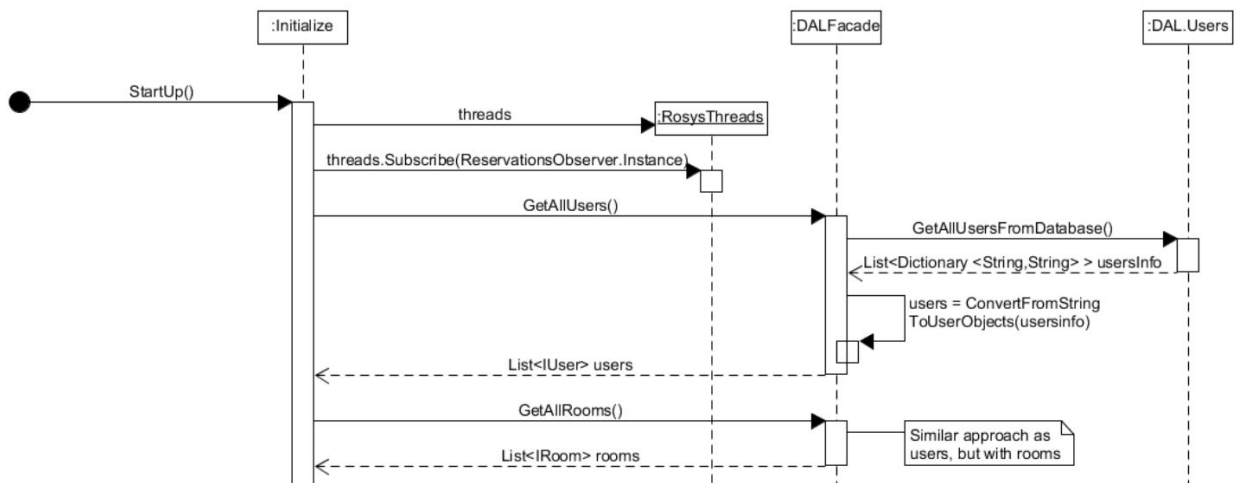


## Sequence Diagram

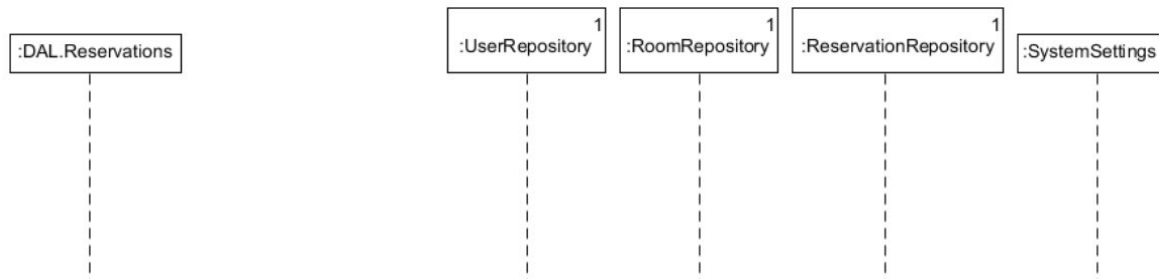
## Startup



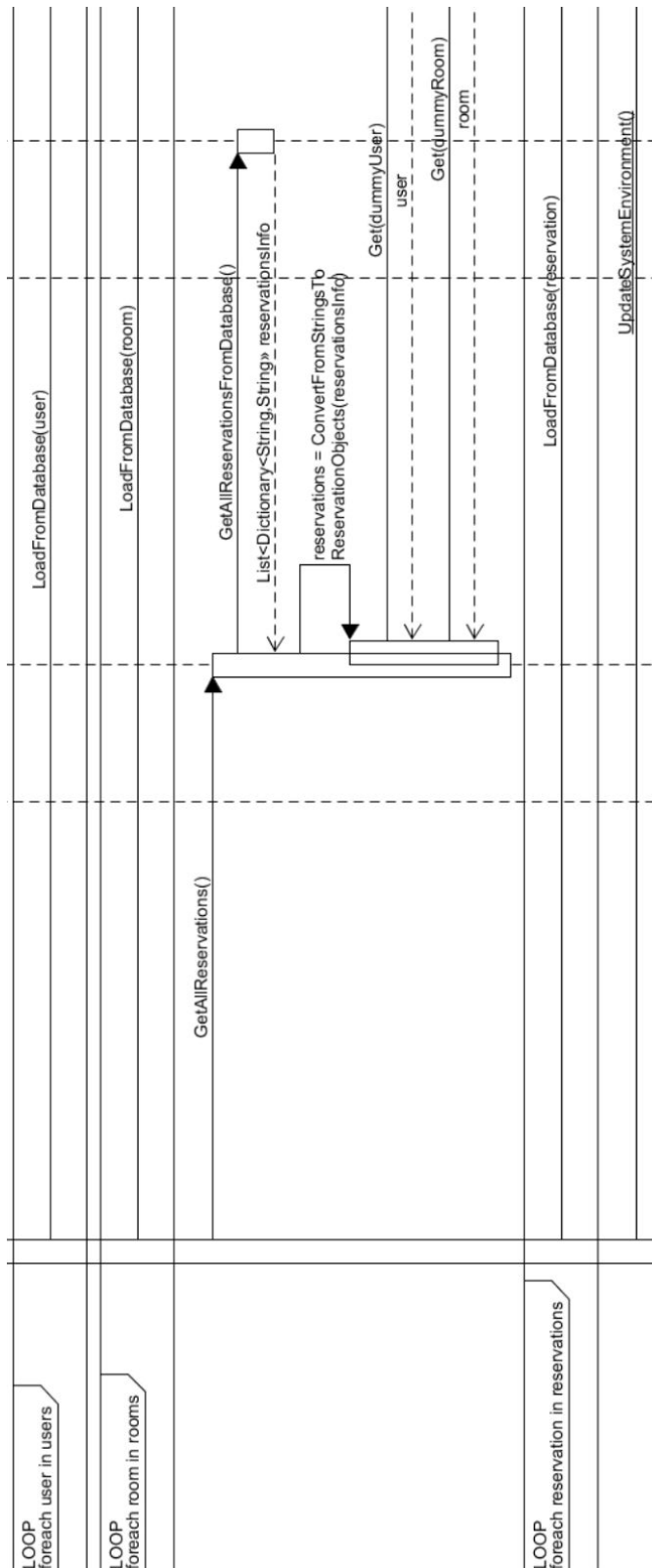
## Section 1



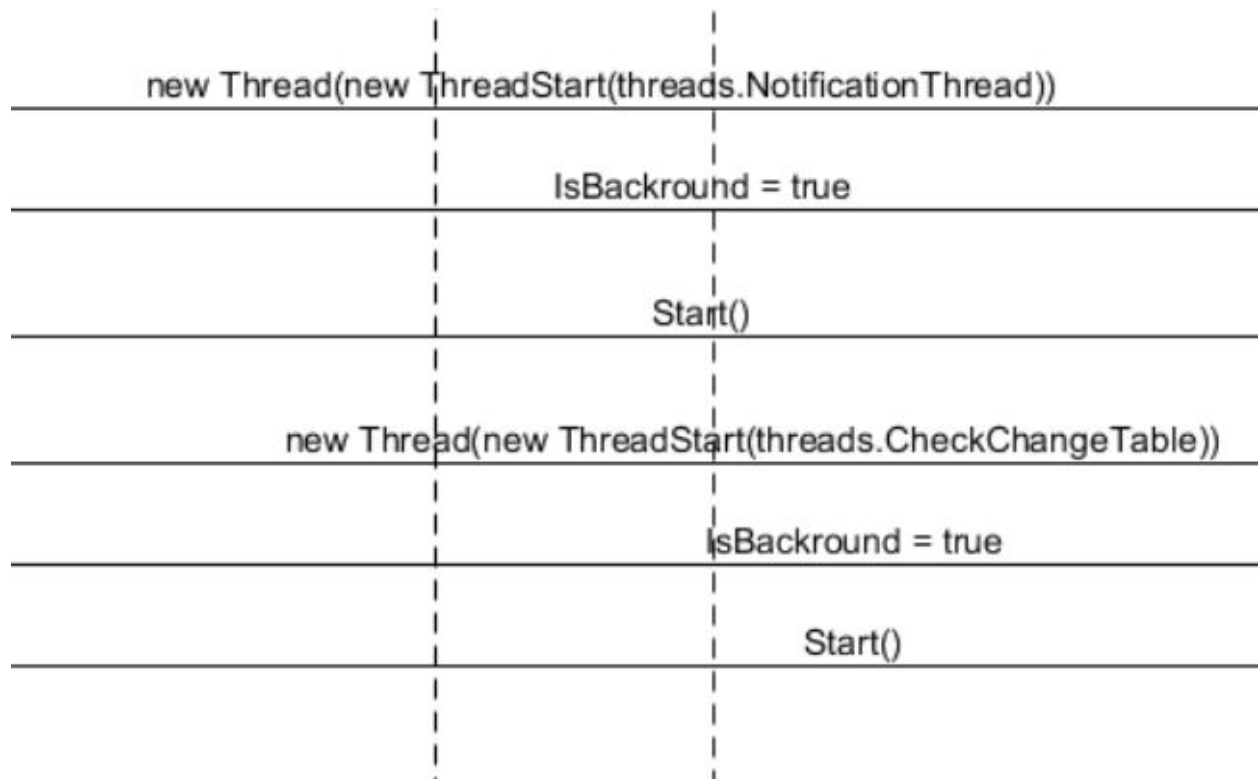
## Section 2



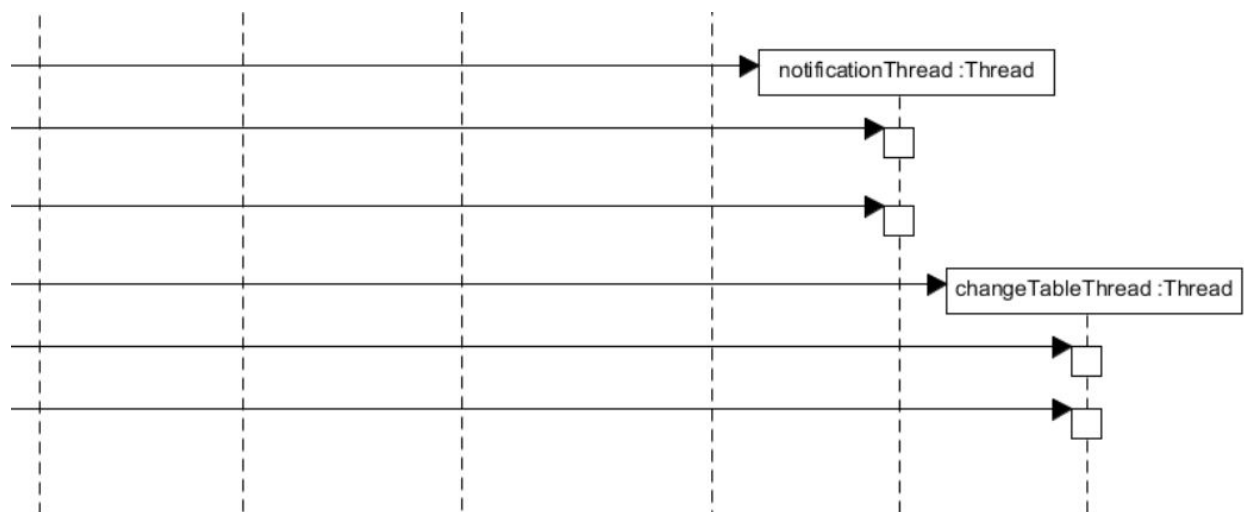
## Section 3



## Section 4



## Section 5





# Group Contract

## General

This is a team project, not an I project. If in doubt, ask. No stupid ideas. Be honest with each other.

Grammar

Project spelling/grammar is to be American based.

No use of contractions such as: isn't, wasn't, can't and others.

## Conflict Handling

Should issues arise professionally in the group, e.g. disagreements over a procedure, a majority win vote should decide it.

Should someone break the contract, a penalty jar is set up. The penalty will be 5 kr.

Matthew will keep the money until the group decides to use it.

A log shall be kept over penalties, in the GitHub folder. The group will vote on whether the penalty will be applied or not, a majority vote is needed to decide.

In case of disagreement, a die is rolled to decide what we do.

## Attendance and Meetings.

You are expected to show up every day. If unable to attend, inform the group via the common text chat.

If unable to attend, you are still expected to attempt to be in the daily scrum meeting via skype or another similar platform.

We will use a Slack group and events to facilitate organization. If you cannot attend an event, please be honest about why not. Always respond to events ASAP. DO NOT RESPOND MAYBE.

We meet at 10:00 am, Monday-Friday; 01:30 pm Sunday[1].

Starting with a stand-up meeting. If late to the stand-up meeting, penalty applies.

## Scrum Usage

The scrum master will change every sprint, as with working pairs.

At the end of each sprint we will do a retrospective to reflect on our work

We will begin each day with a sit-down meeting

Sprints will be 1 week in length and will begin and end on wednesday.

## Developer Tools

Organization of time, place, and specials meetings: Facebook and Events

Organization of project files, code, and version control: Git w/ GitHub and Google Drive.

22 Organization of tasks, project backlog etc: Meister Task

*Exam Project*

*Group 1 - Rooms Reservation System (RoSys)*

## **Naming Conventions**

Methods must describe what they do

Global variables are named with an underscore at the beginning, unless they are constant.

Constant variables are all upper case.

Variables always start with lowercase and methods always start with uppercase, and then camelcase is used after.

Parameters of methods are all lower case.

Variables should describe what they are.

Properties named like methods.

## **Architecture**

User interface should be MVVM.

Main class library should be called Core.

Unit and integration tests will be used, TDD.

File and Database management will be done by Data Access Layer.

The code should follow the SOLID principles.

## **Quality Assurance**

Completing tasks in the QA section of the scrum board is the top priority

You cannot check tasks that you created (all grammar is checked by Matthew)

You must follow the given criteria when checking tasks

The scrum master will assign group members to check tasks

## **How you are expected to use Git / GitHub**

Only change one method per commit.

Expected to make decent comments.

Sync whenever you leave your computer and whenever you feel it is necessary.

Each task should be dealt with in separate branches

After QA, a pull request to master should be made, then the scrum master reviews, merges, and closes the branch

## **Logging**

There should be an entry in the log every day that work is done.

It is the common responsibility to ensure the log is written.

During meetings, a secretary is assigned to write a summary of the meetings.

The scrum master will choose the secretary

## **Voting**

All votes are equal

You cannot abstain

### **Amending the contract**

To amend the contract, a 75% vote is needed to approve of the amendment.

### **Team Pac-Man members:**

Hedviga Arta Geriņa; hedv0149@edu.eal.dk

Matthew Peterson; matt2694@edu.eal.dk

Roxana Ion; roxa0188@edu.eal.dk

Jonas Laursen; jona8690@edu.eal.dk

[1] Jonas except from meeting Sunday due to work

## **<sup>1</sup>Business Case**

### **Executive Summary**

A better way to have rooms available for reservation is needed. We will create a better reservation system at little to no cost to Erhvervsakademiet Lillebælt. There is little risk, but because we are second semester students we do not have the knowledge or expertise most professionals do, but the cost benefits are worth the risk.

### **Financial Appraisal**

The project will have little to no cost for the school because we are completing this project as a school project and thus will not be requiring monetary compensation for the work we do on this specific project. If the school would like to have our group do extra work for them beyond what is in the scope of this project, we may seek monetary compensation. The school may need to pay to have its employees trained to use our solution and to implement the solution.

The purpose of the project is not focused on monetary gain because public schools are not trying to make a profit in the same way that a business is. The school gets money from the government, not the students. The school does, however, need students to continue operating.

Our system will help to make the school a more attractive option for incoming students because it will give it a better reputation as a place to work. Also, the students will be more able to work using the project rooms and this could have a positive effect, albeit small, on the performance of students.

### **Problem Description**

---

<sup>1</sup> The format for this business case was taken from this website:

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

Project rooms are often reserved multiple weeks ahead (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 students who went to the reception.

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

## **Business Objective**

Our goal is to have a better way to reserve rooms in the school. We want it to be very simple for anyone to reserve a room and to get the room they need. We also want to solve the problem of rooms being reserved by people who are not using them.

We will need access to various school resources such as: users, system for displaying information on the tablets and information about rooms. This will allow students to find the rooms they need to work. This is good for the school because it will improve their reputation and bring more students.

## **Benefits and Limitations**

The benefits of this project include: happier students, more efficient students, remain competitive and improve customer service (with staff and students). We will not be focusing on working with the meeting rooms or administration rooms, our focus will be on the project rooms. Since we are second semester students we may not have the knowledge to complete every part that would be required to make a completed project. Also, we may not have time to complete everything.

## **Option Identification and Selection**

Problem: Too many unused but reserved rooms

Solution 1: We will make a new reservation system for the school to use. It will have CRUD operations for reservations, users and rooms. It will also have students get notifications about their reservation and have them confirm the reservation when they get to the room.

*Exam Project*

*Group 1 - Rooms Reservation System (RoSys)*

Solution 2: Have a staff member walk around to all the different rooms to see if the rooms are in use, and if a room is not in use but is reserved, then the staff member will delete that reservation.

Solution 3: We do nothing.

Solution 1 is a relatively cheap solution to the problem. Also, it will cancel the reservations much faster than solution 2 because it does not require a person to walk around to the room. Solution 2 will require an employee to be paid to walk around the school, and this is just a waste of time and money when an IT solution would be much cheaper in the long run. Solution 3 is the cheapest, but it doesn't solve the problem at all. Therefore, based on this, solution 1 is the best solution for the problem.

**Main Solution:** We will make a new reservation system for the school to use. It will have CRUD operations for reservations, users and rooms. It will also have students get notifications about their reservation and have them confirm the reservation when they get to the room.

### Scope, Impact and Interdependencies

The project will affect all the users that reserve rooms and the reservation system for the school. It has interdependencies with the database of users and rooms. It is possible that it will have other dependencies beyond how we create it.

### Outline Plan

We will be working in one week sprints using the scrum methodology. We will continue doing this until the end of the project on June 2nd.

### Risk Assessment

| Incident                          | Impact | Vulnerability | Threat   |
|-----------------------------------|--------|---------------|--|
| Unauthorized access to admin user | High   | High          | This would be very bad because it would basically make the system useless. It is highly vulnerable as well because there are many computer |

|   |        |            |   |
|---|--------|------------|---|
|   |        |            | science students in the school who may be able to do this.  |
| Memory leak, exposing reservations                    | Low    | High       | This would not be very bad because it will not have any effect on any reservations. It is highly vulnerable though because this is a school with many computer science students who may be able to do this. |
| Natural Disasters                                     | High   | Low        | The chance of any sort of natural disaster happening in Denmark and damaging our system is extremely low, but if it did happen it would be very bad.  |
| An admin user abuses their power to damage the system | Medium | Medium/Low | Since the admin users have a lot of power they could easily do it, but the chance that an admin would want to do this is low. It could cause an annoyance to many users, but it would not have              |

|  |        |     |   |
|--|--------|-----|---|
|  |        |     | dire consequences.  |
| An admin accidentally damages the system | Medium | Low | Since the admin users have a lot of power they could easily do it, but the chance that they could accidentally do it is very low. It could cause an annoyance to many users, but it would not have dire consequences. |
| Power to school cut off                  | Medium | Low | This would not have any lasting effects, but the system would not work while the power is out. The chance of this happening is low.   |

## Project Approach

We will work using the scrum methodology. Our sprints are one week in length and will begin and end on Wednesday. We will begin each day with an informal sit-down meeting where we discuss what we need help with and what we have done and will do. Also, at the end of each sprint the group will have a retrospective to discuss how things worked for that sprint and if there is any way we could improve the sprint for next time. We will rotate the scrum master for each sprint so that each group member can get experience with that position. Since we are working on this for our school project we will not be having anyone else doing any of the work for us. We will be getting guidance and assistance from teachers and other students.

## Project Governance

*Exam Project*  
*Group 1 - Rooms Reservation System (RoSys)*

Scrum Master:

Rotating between group members

Group Members:

Matthew Anthony Peterson

Roxana Ion

Hedviga Arta Gerina

Jonas Amstrup Laursen

Stakeholders:

Jens Mejer Pedersen – Rector

Asger Rabølle Nielsen – Pro-rector/Director of Education

Mette Greisen – Director of Education

Leif Bojesen – Director of Education

Ole Hjort Willatzen – Director of Finance and Resources

Torben Lindegaard Hansen – Director of Research and Development

Rikke Kjærsgaard Budden – Head of Study Administration

Marie Falk Nyboe – Head of Study Guidance and Career Centre

Peter Topp Jensen – IT Manager

All students and staff at EAL

All future students and staff

Future employers of students

## Progress Reporting

|   |   |
|---|---|
| KPI                                     | Less Empty Project Rooms  |
| Why measure?                            | This will show if the system is fixing the problem we set out to solve by doing this project.   |
| How measure?                            | Count the amount of project rooms that are reserved and being used before we implement our system and the amount of project rooms that are reserved and being used after we implement our system. Then, take that value and divide it by the total number of project rooms available for reservation. |
| Who is responsible for the measurement? | Group 1   |
| Expected date for measuring             | During the last week of use for the old system around 13:00, and again one month after the implementation of our system around 13:00.   |

### *Exam Project*

#### *Group 1 - Rooms Reservation System (RoSys)*



|   |   |
|---|---|
| Expected values measured  | The percentage reserved and used project rooms should be higher after we implement our system |
| Measure   |   |
| Plan of action in case the measure is outside the range of the expected measure | Remake or eliminate the system.   |
| Responsible for action  | Group or IT at school.  |

## Operation Contracts

### Contract C02: Admin ReserveRoom

Operation:

- ReserveRoom(date : string, from : string, to : string, room : IRoom, username : string)

Cross References:

- SSD: SSD\_Reserve
- UC: Admin reserves a 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

### Contract C03: Delete

Operation:

- ReservationRepository.Delete(reservation : Reservation)

Cross Reference:

- SSD: SSD\_Delete
- UC: Client cancel own reservation; Admin cancels a reservation

Preconditions:

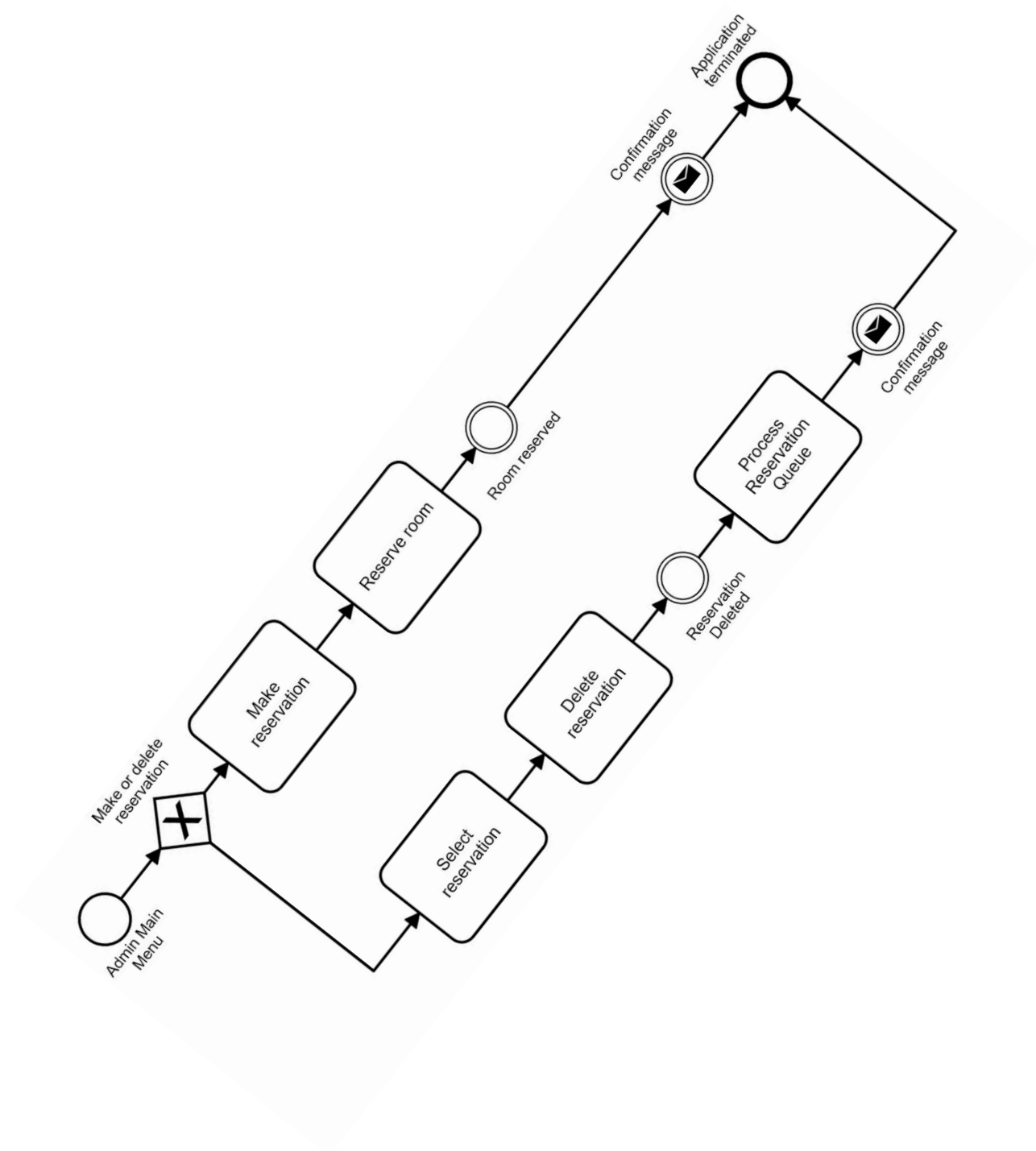
- Reservation Exists

Postconditions:

- A Reservation was deleted

# BPMN

## Admin



# Manual

## Introduction

This manual is a guide that should help clarify some of the ideas we used throughout the program. It will detail how some parts of the program works.

An important thing to keep in mind about our program is that some parts, such as the user interface, were made purely to demonstrate the program's functionality. The main part of the program was the “Core” project, which is meant to serve as a backend server to any/multiple user exposed interfaces.

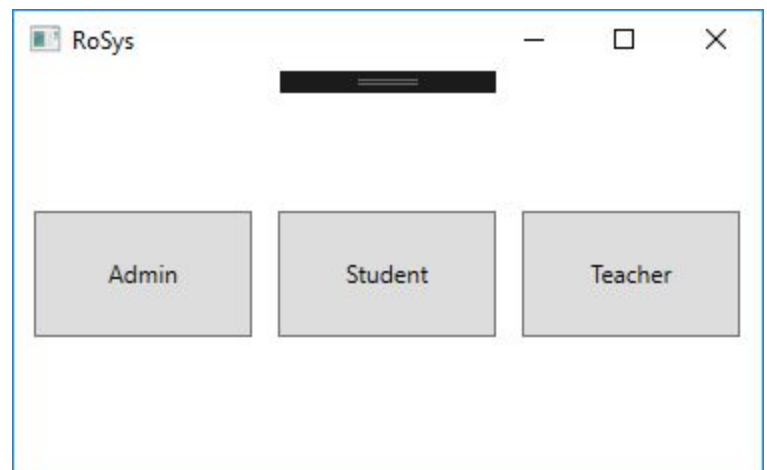
Our focus with the Core is the ability to integrate into other systems, so we made sure to use interfaces for some of the main resources with the program.

This focus on being able to integrate the program, as well as wishing to demonstrate the programs functions on its own, was a greater challenge than expected.

## The User Interface

**Remember: The User interface was made purely to demonstrate features of our program**

Picture 1.



## Initial Run

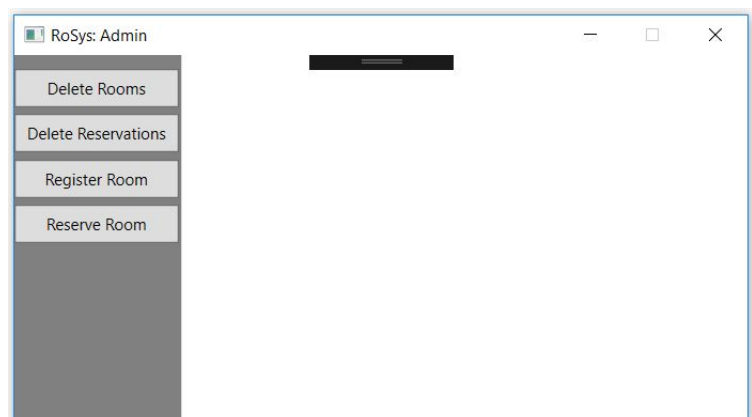
When running the program, there are, initially, 3 options, as seen in Picture 1. This is essentially our “login”, where selection is made of which type of user will interact with the program.

The different types of users have different functionality. Student and Teacher generally have the same features, but Teacher has access to rooms that Student does not.

Admin has some management features, like deleting rooms and reservations, registering new rooms, selecting specific rooms for a reservation and access to a wider variety of rooms to reserve.

By pressing the “Delete Rooms” button, in Picture 2, a list of rooms appears, from which a selection can be made, and then can be deleted, Picture 3. This will delete Rooms from the repository and database. Also, it will delete all reservations for that

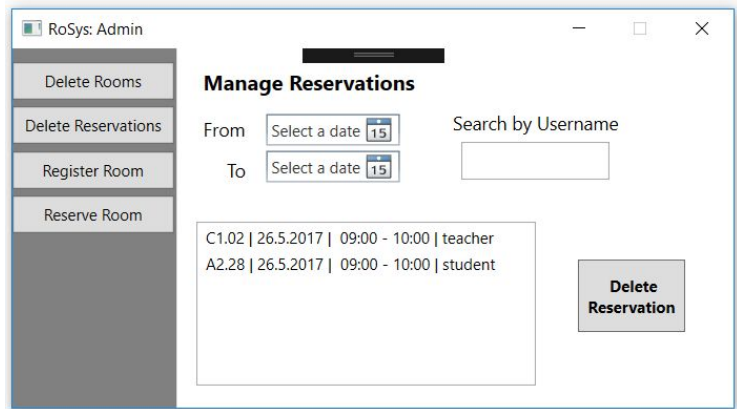
Picture 2.



Picture 3.

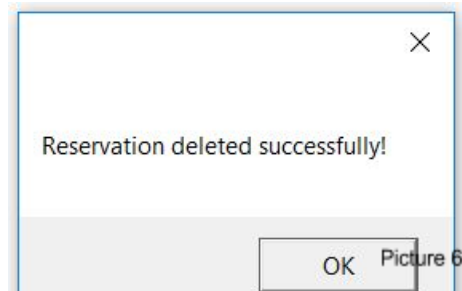
room from the program and database.

When the “Delete Reservations” button is pressed, a page, “Manage Reservations”, appears, Picture 4. If both select dates are empty as well as “Search by Username” the list shows all reservations. If only “Search by Username” is filled, then the list shows all reservations for that specific user. If only one “Select a date” field is filled with a specific date, then the list shows either all reservations from that date or until that date, depending on which date was filled in – from or to. If both “Select a date” fields are filled, then the list shows all reservations for that period. If only one “Select a date” field is filled with a specific date and “Search by Username” is filled, then the list shows either that users reservations from that date or until that date, depending on which date was filled in, the same as previously. And if all fields are filled in, then the list displays all reservations for that specific user in that specific period.



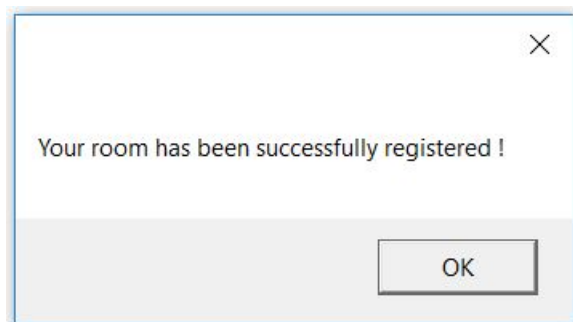
Picture 4.

If an admin selects a reservation and presses the “Delete Reservation” button that reservation will be removed from the repository and database and a message will appear, Picture 5. To close the message box, the admin needs to press the “OK” button.

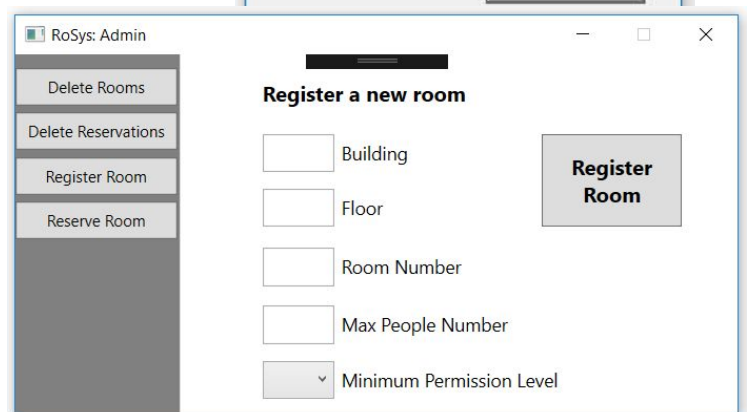


Picture 5.

To register a new room in RoSys, an admin needs to press the “Register Room” button, Picture 6. Before pressing the “Register Room” button in the “Register a new room” page, all fields must be filled, otherwise the program will send a pop-up message. In the “Building” field, only one letter can be inputted. If all fields are filled and the “Register Room” button is pressed, a message will popup (Picture 7).



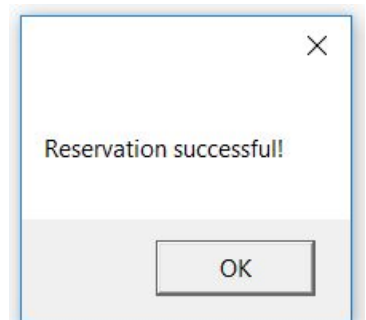
Picture 8.



Picture 7. To close the message box, the admin needs to press the “OK” button.

Admins can also reserve rooms. All fields must be filled. Date and times are automatically set to today’s date and 9:00-10:00, respectively (Picture 8). The user

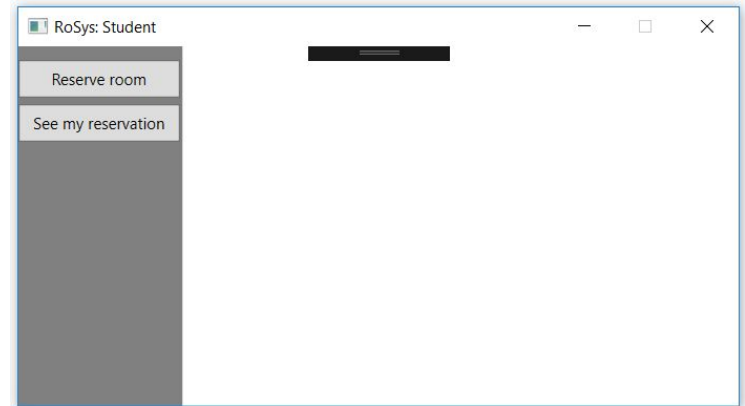
needs to exist in the database and repository and the



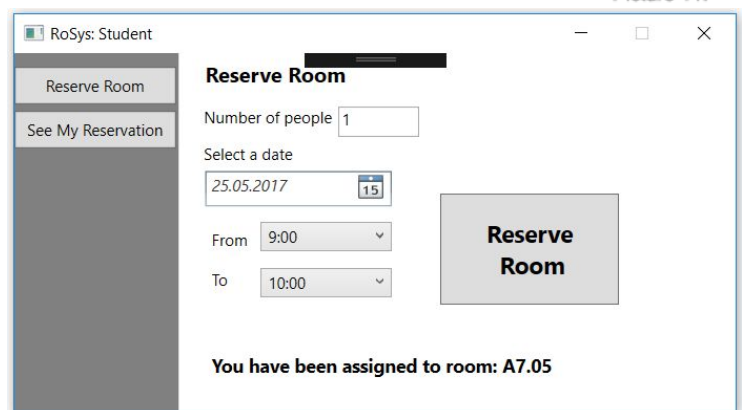
text, “Enter a username”, must be cleared before the admin can input the username. The admin also needs to select which room he wants the reservation to be in. If everything is done right, then a message will appear, like in Picture 9.

By pressing the “Student” button in the menu, Picture 1, the “RoSys: Student” window pops up, Picture 10.

The “Reserve Room” button opens a page with four parameters to reserve a room: “Number of people”, “Select a date”, “From” and “To” (Picture 11). The number of people that will be in the room needs to be typed in the textbox. The “Select a date” box is automatically set to today’s date if not changed manually by clicking on calendar icon and selecting a specific date. The “From” and “To” drop downs are made to select the time that the reservation will start and the time it will end. If any of the fields are still empty, then a message should be shown, but it has not been implemented yet. If a student user tries to reserve another room at the same time and date, the room will not be reserved and a message will be displayed, Picture 12. If there are available rooms with matching parameters as inputted, then a message saying that the user has been assigned to a room, as shown at the bottom of Picture 11, if there are no available rooms, then a message saying that there are no available rooms will appear, Picture 13.



Picture 11.



Picture 12.

To see the reservation the student user has made, he or she needs to press the button, “See My Reservation”, Picture 14. One list item shows one reservation. A reservation is shown as the room it is in, the date, the time and the user who made the reservation. If the user no longer needs the reservation, then he or she can select a reservation and press the button, “Delete Reservation”. This will delete the reservation from the repository and database.

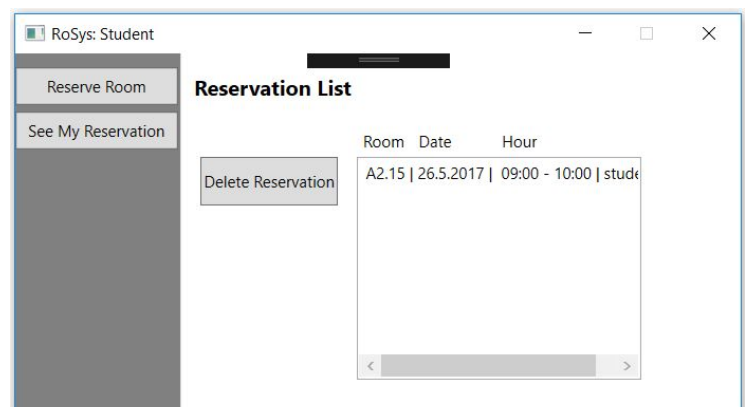
**You already have a room booked, at this time.**  
Picture 13.

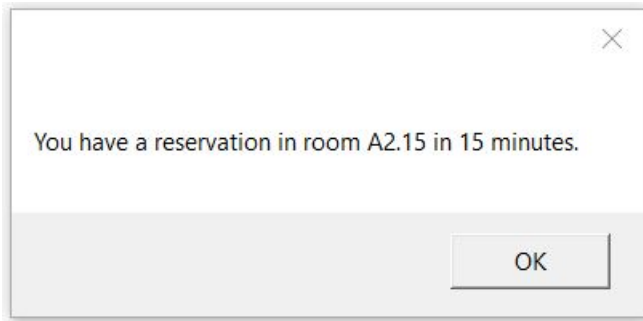
**No rooms available**

Picture 14.

As we mentioned earlier, the teacher user has the same functions as the student user, but teachers can reserve rooms with teacher permission level, which

Picture 15.





student users are not able to reserve.

The system also notifies the user if they have a reservation in 15 minutes, Picture 15. By pressing button “OK” message box will be closed.

## Code

This section will cover some of the features of the code.

### Core: DAL Facade

This class is like a bridge between our DAL and Core. It also converts things from database strings into objects and vice versa.

#### **DeleteAllUsers, DeleteAllRooms**

Calls a method in DAL.Users or DAL.Rooms to delete all users or rooms from the database.

#### **InsertRoom, InsertUser, PassReservationToDAL**

Calls a method in DAL.Rooms, DAL.Users or DAL.Reservations to insert a room, user or reservation to the database.

#### **GetUser, GetRoom, GetReservation**

These three methods are the ones that collect data from the DAL and then return a User, Room or Reservation object to its caller.

#### **GetAllUsers, GetAllReservations, GetAllRooms**

These three methods are like the previous ones, but they return lists of all User, Room or Reservation objects from the database.

#### **ConvertFromStringsToUserObjects**

This method creates User objects from dictionaries of strings. Before creating the object, it converts the PermissionLevel string into an integer, so in Helper Functions this can be converted into permission level. Then it creates a new User and adds it to the list and returns the list.

#### **ConvertFromStringsToRoomObjects**

This method creates Room objects from dictionaries of strings. Before creating the object, it converts the permission level string into an integer, so in Helper Functions this can be converted into permission level. It also converts floorNr, nr and peopleNr from string to integer and building from a string to char. Then it creates a new Room and adds it to the list and returns the list.

#### **ConvertFromStringsToReservationObjects**

This method creates Reservation objects from dictionaries of strings. Before creating the object, it creates a new User using the username from the database and calls the Get(IUser) in the UserRepository to get the User from the repository. It also creates a new Room using the building, floorNr and nr from the database and calls the Get(IRoom) in the RoomRepository to get the Room from the repository. Then it creates a new Reservation and adds it to the list and returns the list.



### **DeleteReservation, DeleteRoom, DeleteUser**

Calls a method in DAL.Reservations, DAL.Rooms or DAL.Usersto delete a reservation, room or user from the database.

### **ConvertFromReservationObjectToStrings**

This method does the opposite of ConvertFromStringsToReservationObjects. It converts all the properties of the reservation to strings and adds them to a dictionary of strings.

## **Core: HelperFunctions**

Used to keep various generic functions that are used throughout the core project.

### **ConvertIntToPermission**

The method uses a switch to convert an int to a Permission type (0 = Student, 1 = Teacher, 2 = Admin).

### **TimeCollides**

Checks if a specific DateTime value lies between two other DateTime values.

## **Core: Initialize**

### **StartUp**

Initializes the program. All loading from the database is done here. Threads for notifications and checking the Change table start running here.

## **Core: Reservation**

### **Equals**

Reservations are considered equal if they both have the same User, From and To values. The comparison of those three are done using each ones respective Equals method.

### **GetHashCode**

The hash code of a reservation is the hash code of its User, From and To added together.

### **To String**

The ToString method returns the Room ID/In Queue (depending on if it is the queue or not) followed by the date of the reservation, the time of the reservation and finally, the username associated with the reservation.

## **Core: ReservationRepository**

The reservation repository is a singleton. It has list of reservations.

### **RequestReservation**

First, it checks if the user already has a reservation in the specified timeperiod. Then, it gets all rooms that have a permission level that is lower than or equal to the users permission level and have a higher peopleNr than the reservation specified.

RemoveUnavailableReservations is called to return a list of rooms that do not have reservations in the specified timeperiod. If the list of available rooms is empty, then an exception is thrown and the reservation is added to the queue. If there are rooms in the list, then it creates a new Reservation, adds it to the repository and returns the room that the reservation will be in.

### **GetAvailableRooms**

Calls the GetPossible method in the RoomRepository to get the rooms the provided user has access to. RemoveUnavailableReservations is called to return a list of rooms that do not have reservations in the specified timeperiod. Then, the list is returned.

### **RemoveUnavailableRooms**

Receives a list of rooms and two DateTime values and calls the IsAvailable method in each room and sends the two DateTime values. If IsAvailable returns true the room is added to the list. Then, the list is returned.

### **DeleteFromQueue**

Removes the reservation from the queue.

### **GetQueue**

Returns the list of reservations in the queue.

### **LoadFromDatabase**

Adds the reservation to the repository from the database at startup. It also adds the reservation to the Room and User in the reservation.

### **DeleteFromRepository**

Remove the reservation from the repository and not the database. It also removes the reservation from the Room and User in the reservation.

### **Clear**

Removes all reservations from the repository and database.

### **Add**

Adds the reservation to the repository and database. It also adds the reservation to the Room and User in the reservation.

### **Delete**

Removes the reservation from the repository and database. It also removes the reservation from the Room and User in the reservation. Then, it checks if there are any

reservations in the queue that could replace the deleted reservation. Then a message alerts the user that they now have a reservation and the reservation is removed from the queue.

#### **Get**

Returns the reservation(s) based on the parameters that were sent.

### **Core: Reservations Observer**

#### **Update**

Sets the message property that is used for the notification.

### **Core: Room**

#### **IsAvailable**

Checks if the room has a reservation between the DateTime value parameters, if not, true is returned, if it does, false is returned.

#### **Equals**

Rooms are considered equal if they both have the same ID. The comparison of the IDs is done using the ID's Equals method.

#### **GetHashCode**

Returns the hash code of the room's ID.

#### **ToString**

Returns the room's ID.

#### **CompareTo**

Returns 1 if the other object's MaxPeople is lower than this object's MaxPeople, 0 if the MaxPeople are the same and -1 if the other object's MaxPeople is greater than this object's MaxPeople.

#### **AddReservation**

Adds the reservation to the list of reservations in the Room object.

#### **GetReservations**

Returns the list of reservations in the Room object.

#### **DeleteReservation**

Removes a reservation from the list of reservations in the Room object.

### **Core: RoomRepository**

#### **Clear**

Removes all rooms from the repository and database. It also calls the Clear method in

the `ReservationRepository`.

#### **Add**

Adds the room to the repository and the database.

#### **LoadFromDatabase**

Adds the room to the repository from the database at startup.

#### **DeleteFromRepository**

Removes the room from the repository.

#### **Get**

Returns the room(s) based on the parameters that were sent.

#### **GetPossible**

Returns a list of rooms based on the parameters sent.

#### **Delete**

Removes the room from the repository and database. It also removes any reservation inside that room.

### **Core: RosysThreads**

Implements the `IObservable` interface.

#### **NotificationThread**

Notifies the observer that there is a reservation in 15 minutes. It checks the reservations every minute.

#### **MaintenanceThread**

Removes reservations that have passed from the repository and the queue.

#### **CheckChangeTable**

Gets information from the Change table and determines which command was used in the database and calls the respective method.

#### **InsertInformation**

Determines which table the insert command was used on and calls the respective method.

#### **AddUserToRepository**

Calls the `DALFacade` to get the user that was added from the database and add it to the repository.

#### **AddRoomToRepository**

Calls the `DALFacade` to get the room that was added from the database and add it to

the repository.

#### **AddReservationToRepository**

Calls the DALFacade to get the reservation that was added from the database and add it to the repository.

#### **UpdateInformation**

Not Implemented.

#### **DeleteInformation**

Determines which table the delete command was used on and calls the respective method.

#### **DeleteUserFromRepository**

First, it creates a new user using the primary key value in the change table. Then, it calls the Get(IUser) method and deletes the returned user from the repository.

#### **DeleteRoomFromRepository**

First, it creates a new room using the primary key value in the change table. Then, it calls the Get(IRoom) method and deletes the returned room from the repository.

#### **DeleteReservationFromRepository**

First, it creates a new room and user using the primary key value in the change table. Then, it creates a new reservation using the username, dateTo and dateFrom. Then, it calls the Get(Reservation) method and deletes the returned reservation from the repository.

#### **Subscribe**

Adds the observer to the list of observers.

#### **Unsubscribe**

Removes the observer from the list of observers.

#### **Notify**

Calls the Update method in each Observer and sends the message to be outputted.

### **Core: SystemSettings**

The System Settings class is currently used to define the “environment” the program runs in.

We set up 3 levels of environment, but we only use 2: Development and Production currently have no differences, but Test effects which database we connect to.

#### **Update System Environment**

This method updates the Environment variable.

## Core: User

### Equals

Users are considered equal if they both have the same ID. The comparison of the Usernames is done using the Username's Equals method.

### GetHashCode

Returns the hash code for the user's Username.

### GetReservation

Returns the list of reservations in the User object.

### AddReservation

Adds the reservation to the list of reservations in the User object.

### DeleteReservation

Removes a reservation from the list of reservations in the User object.

### HasReservation

Checks if the User has a reservation in the specified timeperiod.

## Core: UserRepository

### Clear

Removes all users from the repository and database. It also calls the Clear method in the ReservationRepository.

### Add

Adds the user to the repository and the database.

### LoadFromDatabase

Adds the user to the repository from the database at startup.

### DeleteFromRepository

Removes the user from the repository.

### Get

Returns the user(s) based on the parameters that were sent.

### Delete

Removes the user from the repository and database. It also removes any reservation inside that user.

## UI.GUI: LoggedIn

Is the class that stores which user is currently logged in. It is used by the GUI to keep track of

which user was initially selected.

#### **DAL: Change**

##### **GetAllChangesFromDatabase**

Runs the stored procedure, SP\_GetAllChanges, in the database. Then, it adds all the values of one row to a dictionary with the key being the column name. Next, the dictionary is added to a list. Finally, the list is returned.

##### **DeleteChangeFromDatabase**

Runs the stored procedure, SP\_DeleteChange, in the database using the id parameter.

##### **DeleteAllChangesFromDatabase**

Runs the stored procedure, SP\_DeleteAllChanges, in the database.

#### **DAL: Database**

##### **OpenConnection**

Creates an SqlConnection and calls the Open method.

##### **CloseConnection**

Calls the Close method for the SqlConnection.

#### **DAL: DatabaseConn**

A factory class to switch which connection string is used.

#### **DAL: ConnProd**

Makes the connection string be to the production database.

#### **DAL: ConnTest**

Makes the connection string be to the production database.

#### **DAL: Reservations**

##### **GetAllReservationsFromDatabase**

Runs the stored procedure, SP\_GetAllReservations, in the database. Then, it adds all the values of one row to a dictionary with the key being the column name. Next, the dictionary is added to a list. Finally, the list is returned.

##### **GetReservationFromDatabase**

Runs the stored procedure, SP\_GetReservation, in the database using the id as the parameter. Then, it adds all the values of one row to a dictionary with the key being the column name. Next, the dictionary is added to a list. Finally, the list is returned.

##### **DeleteReservationFromDatabase**

Runs the stored procedure, SP\_DeleteReservation, in the database using the

username, from and to parameters.

#### **StoreReservationIntoDatabase**

Runs the stored procedure, SP\_StoreReservationIntoDatabase, in the database using the properties of a reservation as parameters.

### **DAL: Rooms**

#### **GetAllRoomsFromDatabase**

Runs the stored procedure, SP\_GetAllRooms, in the database. Then, it adds all the values of one row to a dictionary with the key being the column name. Next, the dictionary is added to a list. Finally, the list is returned.

#### **GetRoomFromDatabase**

Runs the stored procedure, SP\_GetRoom, in the database using the building, floorNr and nr as parameters. Then, it adds all the values of one row to a dictionary with the key being the column name. Next, the dictionary is added to a list. Finally, the list is returned.

#### **DeleteRoomFromDatabase**

Runs the stored procedure, SP\_DeleteRoom, in the database using the building, floorNr and nr as parameters.

#### **DeleteAllRoomsFromDatabase**

Runs the stored procedure, SP\_DeleteAllRooms, in the database.

#### **InsertRoomToDatabase**

Runs the stored procedure, SP\_InsertRoom, in the database using the properties of a room as parameters.

### **DAL: Users**

#### **GetAllUsersFromDatabase**

Runs the stored procedure, SP\_GetAllUsers, in the database. Then, it adds all the values of one row to a dictionary with the key being the column name. Next, the dictionary is added to a list. Finally, the list is returned.

#### **GetUserFromDatabase**

Runs the stored procedure, SP\_GetUser, in the database using the username as the parameter. Then, it adds all the values of one row to a dictionary with the key being the column name. Next, the dictionary is added to a list. Finally, the list is returned.

#### **InsertUserToDatabase**

Runs the stored procedure, SP\_InsertUser, in the database using the properties of a user as parameters.

#### **DeleteUserFromDatabase**



Runs the stored procedure, SP\_DeleteUser, in the database using the username as the parameter.

#### **DeleteAllUserFromDatabase**

Runs the stored procedure, SP\_DeleteAllUsers, in the database.

# Database Triggers

## Reservations Table

```
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
25.
26. CREATE TRIGGER trgUpdateReservations ON Reservations
27. FOR UPDATE AS
28.     declare @command int;
29.     declare @table    NVarChar(max) ;
30.     declare @PK       NVarChar(max) ;
31.
32.     declare @ID       NVarChar(max) ;
33.     declare @DateTo   NVarChar(max) ;
34.     declare @DateFrom NVarChar(max) ;
35.     declare @Username  NVarChar(max) ;
36.
37.    set @command = 1;
38.    set @table = 'Reservations';
39.    select @ID = i.ID from inserted i;
40.    select @DateTo = i.DateTo from inserted i;
41.    select @DateFrom = i.DateFrom from inserted i;
42.    select @Username = i.Username from inserted i;
43.    select @PK = @ID + ';' + @DateTo + ';' + @DateFrom + ';' +
@Username;
44.
45.    INSERT INTO Change(Command, TableName, PrimaryKey) VALUES
```

```
46.      (@command, @table, @PK);
47.
48.      PRINT 'Trigger: Update Reservation'
49. GO
50.
51. CREATE TRIGGER trgDeleteReservations on Reservations
52. FOR DELETE AS
53.     declare @command int;
54.     declare @table    NVarChar(max);
55.     declare @PK       NVarChar(max);
56.
57.     declare @ID       NVarChar(max);
58.     declare @DateTo   NVarChar(max);
59.     declare @DateFrom NVarChar(max);
60.     declare @Username  NVarChar(max);
61.
62.     set @command = 2;
63.     set @table = 'Reservations';
64.     select @ID = i.ID from deleted i;
65.     select @DateTo = i.DateTo from deleted i;
66.     select @DateFrom = i.DateFrom from deleted i;
67.     select @Username = i.Username from deleted i;
68.     select @PK = @ID + ';' + @DateTo + ';' + @DateFrom + ';' +
@Username;
69.
70.     INSERT INTO Change (Command, TableName, PrimaryKey) VALUES
71.     (@command, @table, @PK);
72.
73.     PRINT 'Trigger: Delete Reservation'
74. GO
```

## Rooms Table

```
1. CREATE TRIGGER trgInsertRooms ON Rooms
2. FOR INSERT AS
3.     declare @command int;
4.     declare @table NVarChar(max);
5.     declare @PK NVarChar(max);
6.
7.     declare @Building NVarChar(max);
8.     declare @FloorNr NVarChar(max);
9.     declare @Nr NVarChar(max);
10.
11.     set @command = 0;
12.     set @table = 'Rooms';
13.     select @Building = i.Building from inserted i;
14.     select @FloorNr = i.FloorNr from inserted i;
15.     select @Nr = i.Nr from inserted i;
16.     select @PK = @Building + ';' + @FloorNr + ';' + @Nr;
17.
18.     INSERT INTO Change(Command, TableName, PrimaryKey) VALUES
19.     (@command, @table, @PK);
20.
21.     PRINT 'Trigger: Insert Room'
22. GO
23.
24. CREATE TRIGGER trgUpdateRooms ON Rooms
25. FOR UPDATE AS
26.     declare @command int;
27.     declare @table NVarChar(max);
28.     declare @PK NVarChar(max);
29.
30.     declare @Building Char;
31.     declare @FloorNr int;
32.     declare @Nr int;
33.
34.     set @command = 1;
35.     set @table = 'Rooms';
36.     select @Building = i.Building from inserted i;
37.     select @FloorNr = i.FloorNr from inserted i;
38.     select @Nr = i.Nr from inserted i;
39.     select @PK = @Building + ';' + @FloorNr + ';' + @Nr;
40.
41.     INSERT INTO Change(Command, TableName, PrimaryKey) VALUES
42.     (@command, @table, @PK);
43.
44.     PRINT 'Trigger: Update Rooms'
45. GO
46.
47. CREATE TRIGGER trgDeleteRooms on Rooms
```

```
48. FOR DELETE AS
49.     declare @command int;
50.     declare @table    NVarChar(max) ;
51.     declare @PK       NVarChar(max) ;
52.
53.     declare @Building Char;
54.     declare @FloorNr  NVarChar(max) ;
55.     declare @Nr       NVarChar(max) ;
56.
57.     set @command = 2;
58.     set @table = 'Rooms';
59.     select @Building = i.Building from deleted i;
60.     select @FloorNr = i.FloorNr from deleted i;
61.     select @Nr = i.Nr from deleted i;
62.     set @PK = @Building + ';' + @FloorNr + ';' + @Nr;
63.
64.     INSERT INTO Change(Command, TableName, PrimaryKey) VALUES
65.     (@command, @table, @PK);
66.
67.     PRINT 'Trigger: Delete ' + @table
68. GO
```

## Users Table

```
1. CREATE TRIGGER trgInsertUser ON Users
2. FOR INSERT AS
3.     declare @command int;
4.     declare @table    NVarChar(max) ;
5.     declare @PK       NVarChar(max) ;
6.
7.     set @command = 0;
8.     set @table = 'Users';
9.     select @PK = i.Username from inserted i;
10.
11.     INSERT INTO Change(Command, TableName, PrimaryKey) VALUES
12.     (@command, @table, @PK);
13.
14.     PRINT 'Trigger: Insert User'
15. GO
16.
17. CREATE TRIGGER trgUpdateUser ON Users
18. FOR UPDATE AS
19.     declare @command int;
20.     declare @table    NVarChar(max) ;
21.     declare @PK       NVarChar(max) ;
22.
23.     set @command = 1;
24.     set @table = 'Users';
25.     select @PK = i.Username from inserted i;
26.
27.     INSERT INTO Change(Command, TableName, PrimaryKey) VALUES
28.     (@command, @table, @PK);
29.
30.     PRINT 'Trigger: Update User'
31. GO
32.
33. CREATE TRIGGER trgDeleteUser on Users
34. FOR DELETE AS
35.     declare @command int;
36.     declare @table    NVarChar(max) ;
37.     declare @PK       NVarChar(max) ;
38.
39.     set @command = 2;
40.     set @table = 'Users';
41.     select @PK = i.Username from deleted i;
42.
43.     INSERT INTO Change(Command, TableName, PrimaryKey) VALUES
44.     (@command, @table, @PK);
45.
46.     PRINT 'Trigger: Delete User'
47. GO
```

Code

Core

ReservationRepository



```
1. using Core.Exceptions;
2. using Core.Interfaces;
3. using System;
4. using System.Collections.Generic;
5.
6. namespace Core
7. {
8.     public class ReservationRepository
9.     {
10.         private DALFacade _dalFacade = new DALFacade();
11.         private List<Reservation> _reservationRepository = new
List<Reservation>();
12.         private List<Reservation> _queue = new List<Reservation>();
13.         private RoomRepository _roomRepo = RoomRepository.Instance;
14.
15.         private static ReservationRepository _instance = new
ReservationRepository();
16.         public static ReservationRepository Instance { get { return _instance; }
}
17.
18.         private ReservationRepository() { }
19.
20.         public IRoom RequestReservation(DateTime from, DateTime to, int peoplenr,
IUser user)
21.         {
22.
23.             if (user.HasReservation(from.AddSeconds(1), to.AddSeconds(-1)))
24.             {
25.                 throw new UserAlreadyHasRoomException();
26.             }
27.
28.             List<IRoom> rooms = _roomRepo.GetPossible(user.PermissionLevel,
peoplenr);
29.             List<IRoom> availableRooms = RemoveUnavailableRooms(rooms, from,
to);
30.
31.             if (availableRooms.Count == 0)
32.             {
33.                 _queue.Add(new Reservation(user, null, peoplenr, from, to));
34.                 throw new NoRoomsAvailableException();
35.             }
36.             else
37.             {
38.                 Reservation reservation = new Reservation(user, availableRooms[0],
peoplenr, from, to);
39.                 this.Add(reservation);
40.                 return availableRooms[0];
41.             }
42.
43.         }
```

```

44.
45.     public List<IRoom> GetAvailableRooms(DateTime from, DateTime to, IUser
      user)
46.     {
47.         List<IRoom> rooms = _roomRepo.GetPossible(user.PermissionLevel);
48.         List<IRoom> availableRooms = RemoveUnavailableRooms(rooms, from,
      to);
49.         return availableRooms;
50.     }
51.
52.     private List<IRoom> RemoveUnavailableRooms(List<IRoom> rooms, DateTime
      from, DateTime to)
53.     {
54.         List<IRoom> availableRooms = new List<IRoom>();
55.
56.         foreach (IRoom room in rooms)
57.         {
58.             bool roomAvailable = room.IsAvailable(from, to);
59.             if (roomAvailable == true)
60.             {
61.                 availableRooms.Add(room);
62.             }
63.         }
64.         return availableRooms;
65.     }
66.
67.     internal void DeleteFromQueue(Reservation res)
68.     {
69.         _queue.Remove(res);
70.     }
71.
72.     internal List<Reservation> GetQueue()
73.     {
74.         return _queue;
75.     }
76.
77.     internal void LoadFromDatabase(Reservation reservation)
78.     {
79.         _reservationRepository.Add(reservation);
80.         reservation.Room.AddReservation(reservation);
81.         reservation.User.AddReservation(reservation);
82.     }
83.
84.     internal void DeleteFromRepository(Reservation reservation)
85.     {
86.         _reservationRepository.Remove(reservation);
87.         reservation.Room.DeleteReservation(reservation);
88.         reservation.User.DeleteReservation(reservation);
89.     }
90.

```

```
91.     public void Clear()
92.     {
93.         foreach (Reservation reservation in _reservationRepository)
94.         {
95.             _dalFacade.DeleteReservation(reservation);
96.         }
97.         _reservationRepository.Clear();
98.     }
99.
100.    public void Add(Reservation reservation)
101.    {
102.        _reservationRepository.Add(reservation);
103.        reservation.Room.AddReservation(reservation);
104.        reservation.User.AddReservation(reservation);
105.        _dalFacade.PassReservationToDAL(reservation);
106.    }
107.
108.    public void Delete(Reservation reservation)
109.    {
110.        _reservationRepository.Remove(reservation);
111.        reservation.Room.DeleteReservation(reservation);
112.        reservation.User.DeleteReservation(reservation);
113.        _dalFacade.DeleteReservation(reservation);
114.
115.        CheckReservationQueue();
116.    }
117.
118.    private void CheckReservationQueue()
119.    {
120.        // Check the queue, see if anyone fits the criterias...
121.        List<Reservation> newRegistrations = new List<Reservation>();
122.        foreach (Reservation res in _queue)
123.        {
124.            List<IRoom> rooms =
125.                _roomRepo.GetPossible(res.User.PermissionLevel, res.PeopleNr);
126.            List<IRoom> availableRooms = RemoveUnavailableRooms(rooms,
127.                res.From, res.To);
128.
129.            if (availableRooms.Count > 0)
130.            {
131.                res.Room = availableRooms[0];
132.                this.Add(res);
133.                newRegistrations.Add(res);
134.            }
135.
136.            // remove the new registrations from the queue, and send
137.            notifications
138.            foreach (Reservation res in newRegistrations)
139.            {
```

```

138.         ReservationsObserver.Instance.Message = "Dear " +
        res.User.Username + "\nYou have recived a reservation in room: " + res.Room.ID +
        "\nSee your reservations for more info.";
139.         _queue.Remove(res);
140.     }
141. }
142.
143.     public void Add(IUser user, IRoom room, int peoplenr, DateTime datefrom,
        DateTime dateto)
144.     {
145.         Reservation reservation = new Reservation(user, room, peoplenr,
        datefrom, dateto);
146.         this.Add(reservation);
147.     }
148.
149.     public List<Reservation> Get()
150.     {
151.         return _reservationRepository;
152.     }
153.
154.     public List<Reservation> Get(IUser user)
155.     {
156.         List<Reservation> reservationsByUser = new List<Reservation>();
157.
158.         foreach (Reservation reservation in _reservationRepository)
159.         {
160.             if (reservation.User.Equals(user))
161.             {
162.                 reservationsByUser.Add(reservation);
163.             }
164.         }
165.         return reservationsByUser;
166.     }
167.
168.     public List<Reservation> Get(IRoom room)
169.     {
170.         List<Reservation> reservationsByRoom = new List<Reservation>();
171.
172.         foreach (Reservation reservation in _reservationRepository)
173.         {
174.             if (reservation.Room.Equals(room))
175.             {
176.                 reservationsByRoom.Add(reservation);
177.             }
178.         }
179.         return reservationsByRoom;
180.     }
181.
182.     public Reservation Get(Reservation checkreservation)
183.     {

```

```
184.         Reservation result = null;
185.
186.         foreach (Reservation reservation in _reservationRepository)
187.         {
188.             if (reservation.Equals(checkreservation))
189.             {
190.                 result = reservation;
191.             }
192.         }
193.         return result;
194.     }
195.
196.     public List<Reservation> Get(DateTime? from, DateTime? to, IUser user)
197.     {
198.         List<Reservation> reservations = new List<Reservation>();
199.         List<Reservation> allReservations = this.Get();
200.
201.         if (from != null)
202.         {
203.             foreach (Reservation reservation in allReservations)
204.             {
205.                 if (from < reservation.From)
206.                 {
207.                     reservations.Add(reservation);
208.                 }
209.             }
210.             allReservations = reservations;
211.             reservations = new List<Reservation>();
212.         }
213.
214.         if (to != null)
215.         {
216.             foreach (Reservation reservation in allReservations)
217.             {
218.                 if (to > reservation.To)
219.                 {
220.                     reservations.Add(reservation);
221.                 }
222.             }
223.             allReservations = reservations;
224.             reservations = new List<Reservation>();
225.         }
226.
227.         if (user != null)
228.         {
229.             foreach (Reservation reservation in allReservations)
230.             {
231.                 if (user.Equals(reservation.User))
232.                 {
233.                     reservations.Add(reservation);
```

```
234.         }
235.     }
236.     allReservations = reservations;
237.     reservations = new List<Reservation>();
238. }
239.     return allReservations;
240. }
241. }
242. }
243.
244.
```

## DAL

### Database

```
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. }
```

## Tests > Unit Tests

### Client Create Reservation

```

1. using Core.Interfaces;
2. using Microsoft.VisualStudio.TestTools.UnitTesting;
3. using System;
4. using System.Collections.Generic;
5.
6. namespace Core.UnitTest
7. {
8.     [TestClass]
9.     public class ClientCreateReservation
10.    {
11.
12.        IRoom _room1;
13.        IRoom _room2;
14.        IRoom _room3;
15.        //IRoom _room4;
16.
17.        IUser _student;
18.        //IUser _teacher;
19.        //IUser _admin;
20.
21.        RoomRepository _repoRoom = RoomRepository.Instance; // Singletons
22.        ReservationRepository _repoReserv = ReservationRepository.Instance;
23.        UserRepository _repoUser = UserRepository.Instance;
24.
25.        [ClassInitialize]
26.        public static void ClassInit(TestContext testContext)
27.        {
28.            SystemSettings.Environment = Environment.Test;
29.        }
30.
31.        [ClassCleanup]
32.        public static void ClassClean()
33.        {
34.            SystemSettings.Environment = Environment.Prod;
35.        }
36.
37.        [TestInitialize]
38.        public void TestsInitialize()
39.        {
40.            _repoRoom.Clear();
41.            _repoReserv.Clear();
42.            _repoUser.Clear();
43.
44.            _room1 = new Room('A', 1, 2, 4, Permission.Student);
45.            _room2 = new Room('A', 1, 99, 8, Permission.Student);
46.            _room3 = new Room('B', 1, 45, 2, Permission.Student);
47.
48.            _repoRoom.Add(_room1);
49.            _repoRoom.Add(_room2);
50.            _repoRoom.Add(_room3);

```



```

51.
52.         _student = new User("roxa0198", "roxa0188@edu.eal.dk",
Permission.Student);
53.
54.         _repoUser.Add(_student);
55.     }
56.
57.     [TestMethod]
58.     public void SortRoomsByMaxPeopleIntoFILOStack()
59.     {
60.         List<IRoom> StackRooms = _repoRoom.GetPossible(Permission.Student,
4);
61.
62.         Assert.AreEqual(StackRooms[0], _room1);
63.         Assert.AreEqual(StackRooms[1], _room2);
64.     }
65.
66.     [TestMethod]
67.     [ExpectedException(typeof(ArgumentOutOfRangeException))]
68.     public void SortRoomsByMaxPeopleIntoFILOStackOutOfRooms()
69.     {
70.         List<IRoom> Rooms = _repoRoom.GetPossible(Permission.Student, 4);
71.
72.         IRoom test = Rooms[2];
73.     }
74.
75.     [TestMethod]
76.     public void RoomIsAvailable()
77.     {
78.         DateTime from = new DateTime(2017, 05, 01, 13, 0, 0);
79.         DateTime to = new DateTime(2017, 05, 01, 14, 0, 0);
80.         bool roomAvailable = _room1.IsAvailable(from, to);
81.         Assert.IsTrue(roomAvailable);
82.     }
83.     [TestMethod]
84.     public void RoomIsNotAvailable()
85.     {
86.         DateTime from = new DateTime(2017, 05, 01, 13, 0, 0);
87.         DateTime to = new DateTime(2017, 05, 01, 14, 0, 0);
88.         Reservation res = new Reservation(_student, _room1, 3, from, to);
89.         _repoReserv.Add(res);
90.         bool roomAvailable = _room1.IsAvailable(from, to);
91.         Assert.IsFalse(roomAvailable);
92.     }
93. }
94. }
95.
96.

```