

Appendix

Table of Contents

Table of Contents	1
Program License	3
Use Cases	3
Personas	5
Scenarios	6
Admin books a room:	6
Admin cancel a reservation:	6
Canceling own reservation:	7
System Sequence Diagrams	7
Reserve room by Admin	7
Delete reservation by Client	8
Delete reservation by Admin	8
Sequence Diagram	9
Startup	9
Section 1	10
Section 2	10
Section 3	11
Section 4	12
Section 5	12
Group Contract	13
Business Case	15
Operation Contracts	21
BPMN	22
Admin	23
Manual	24
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
Database Triggers	38
Reservations Table	38
Rooms Table	40
Users Table	42
Code	44
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

Use case name	Client deletes own reservation
Use case description	A client wishes to delete a reservation from the system
Actor	Client (Students & Teachers)
Pre-conditions	Client has reservation
Basic Flow (Success Scenario)	<ol style="list-style-type: none">1. Client enters reservation system2. Client requests the system to display his reservations list3. Client selects which reservation he wants to delete4. Client requests for reservation to be deleted5. System checks reservation queue to see if any of the reservations can have a room6. Client receives notifications that cancellation was successful
Post-conditions	Reservation(s) are removed from repository, Rooms and Users lists and database.
Alternate Flow	None significant

Use case name	Admin reserves a room
----------------------	-----------------------



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

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

Basic Flow (Success Scenario)	<ol style="list-style-type: none">1. Admin enters reservation management system2. OPT: Inputs criteria (from date to date, username) to filter list3. Admin selects which reservation he wants to delete4. Admin requests deleting the reservation5. Receives confirmation notification that room is deleted
Post-conditions	Reservation has been deleted
Alternate Flow	None significant

Use case name	Notification System
Use case description	System is sending a reminder 15 minutes before the client's reservation is beginning.
Actor	System
Pre-conditions	System must be running
Basic Flow (Success Scenario)	<ol style="list-style-type: none">1. System gets upcoming reservations2. System sends reminder for upcoming reservations3. System checks again in one minute
Post-conditions	Reminder notification was sent
Alternate Flow	AF1: No upcoming reservations 1.System checks again in one minute

Personas

 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.
 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:

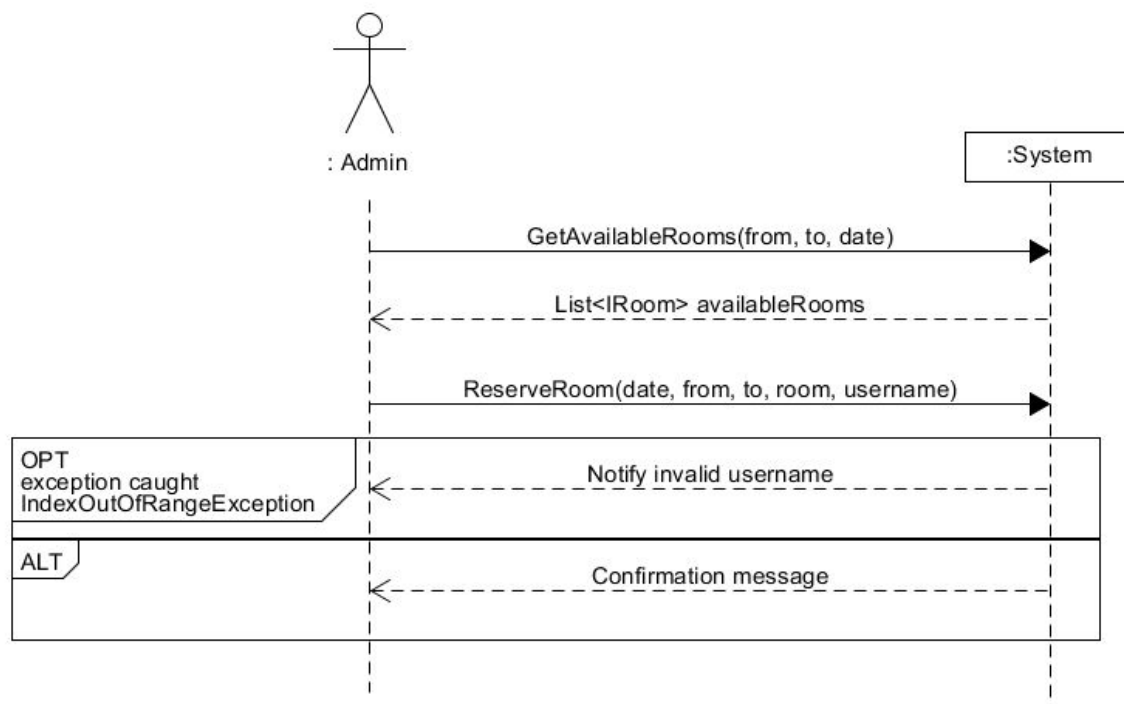
Leo was arranging a meeting between some teachers, but something has happened and he no longer needs his booked meeting room.

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".

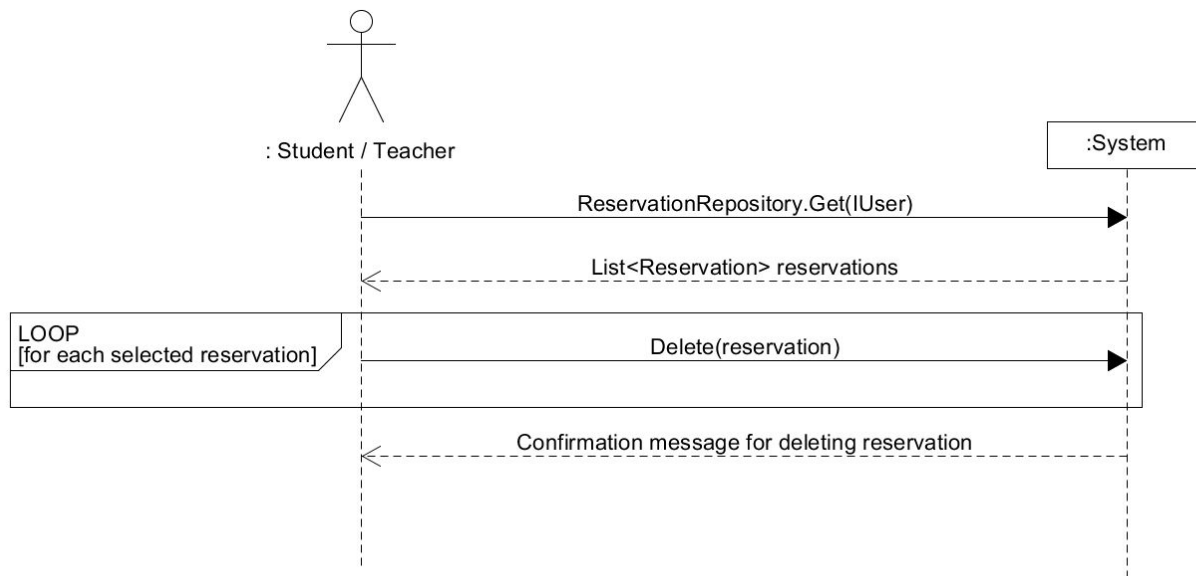
A message ensures Leo that his reservation has been cancelled.

System Sequence Diagrams

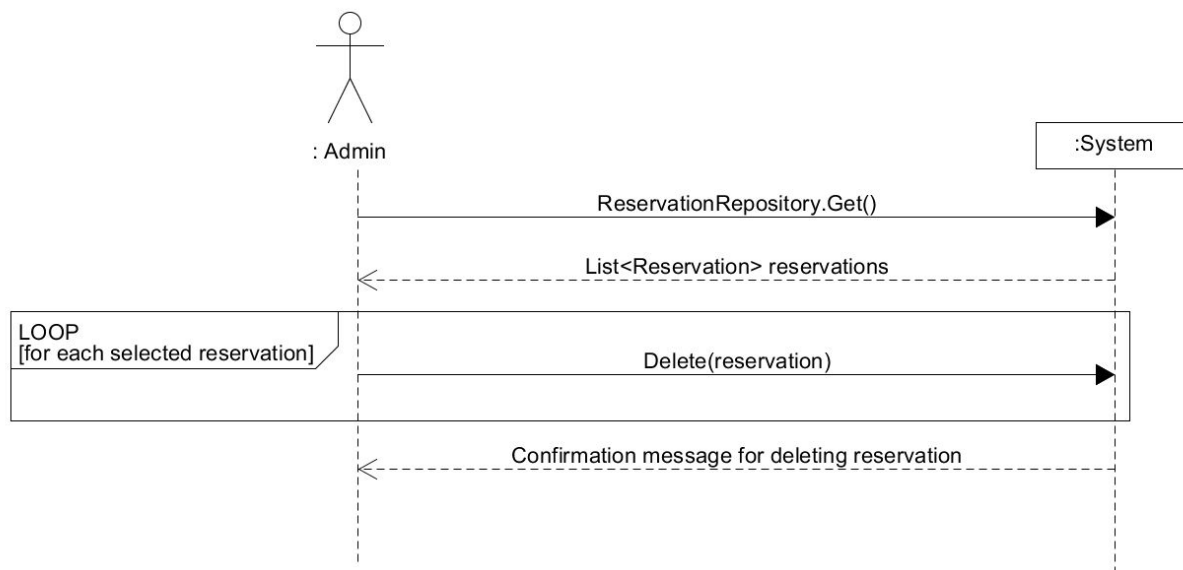
Reserve room by Admin



Delete reservation by Client

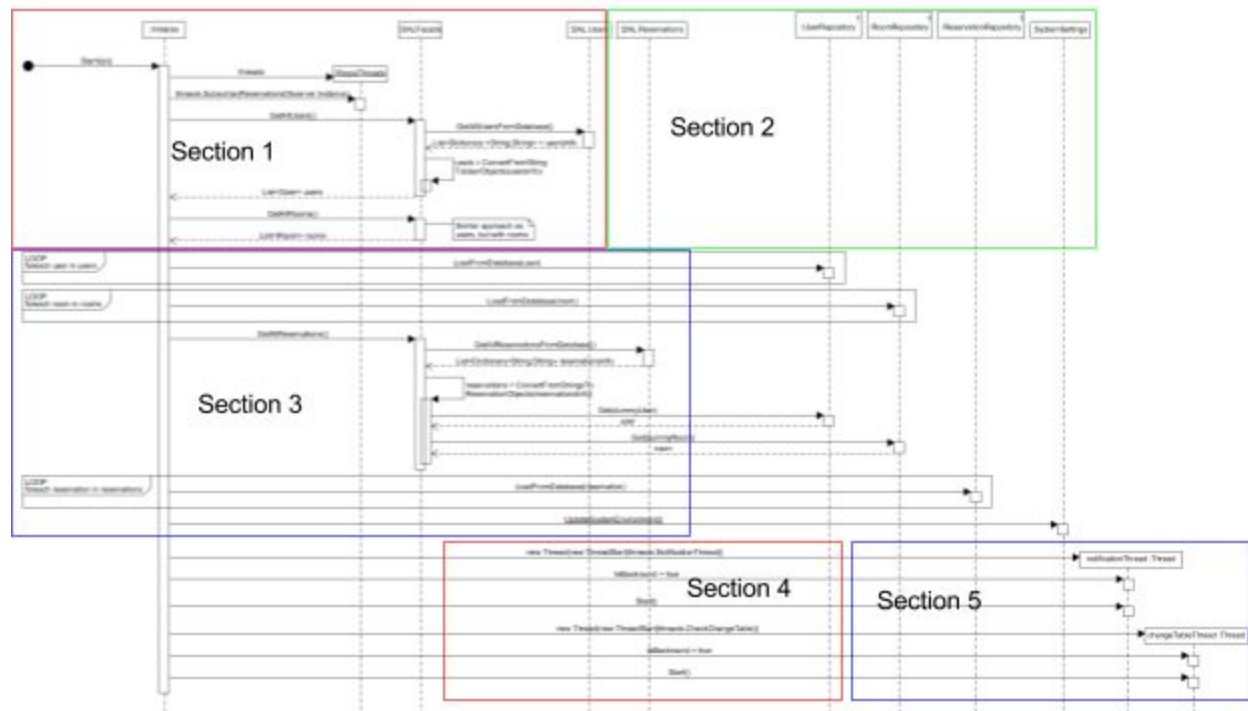


Delete reservation by Admin

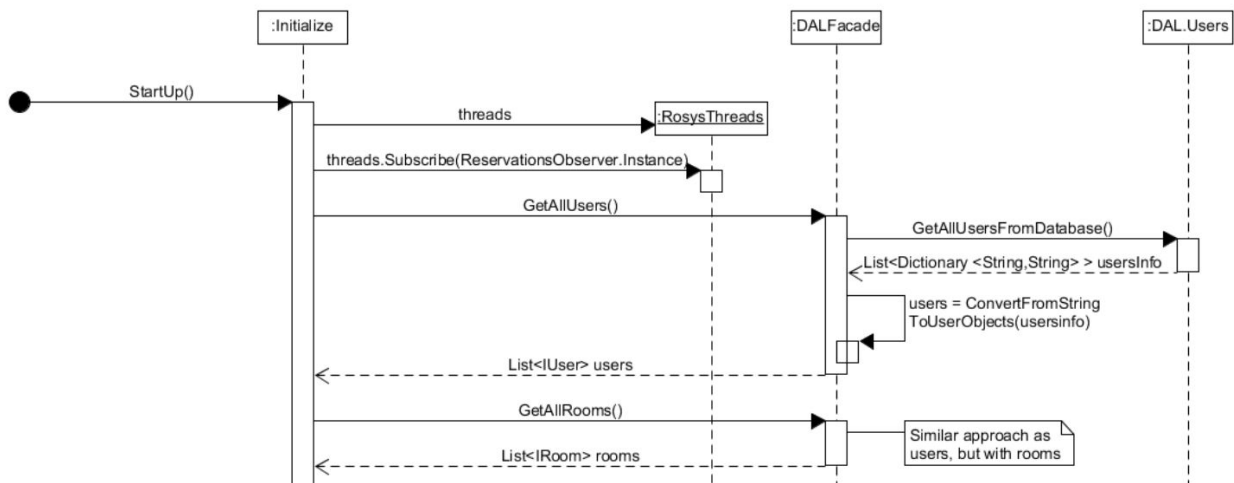


Sequence Diagram

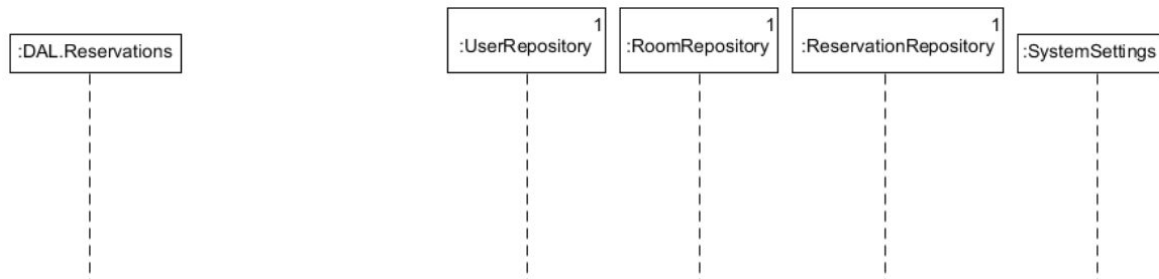
Startup



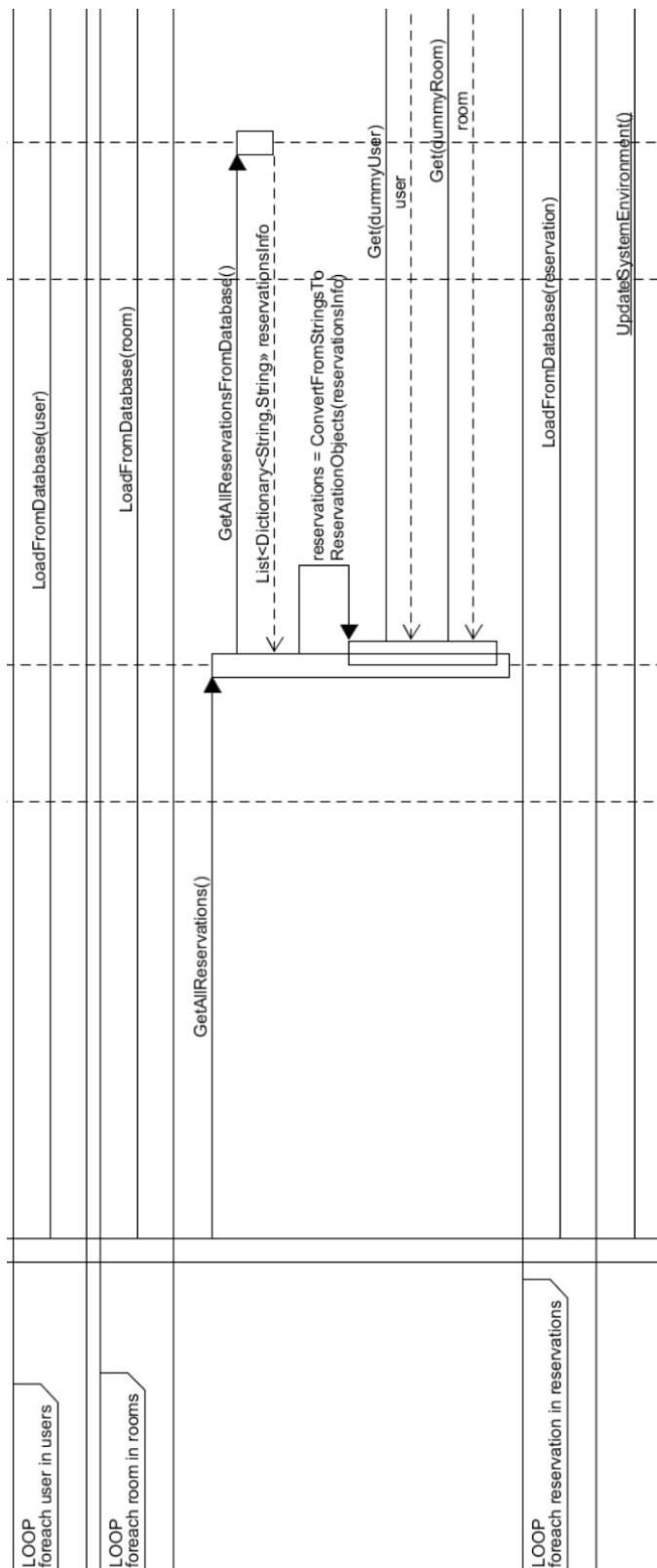
Section 1



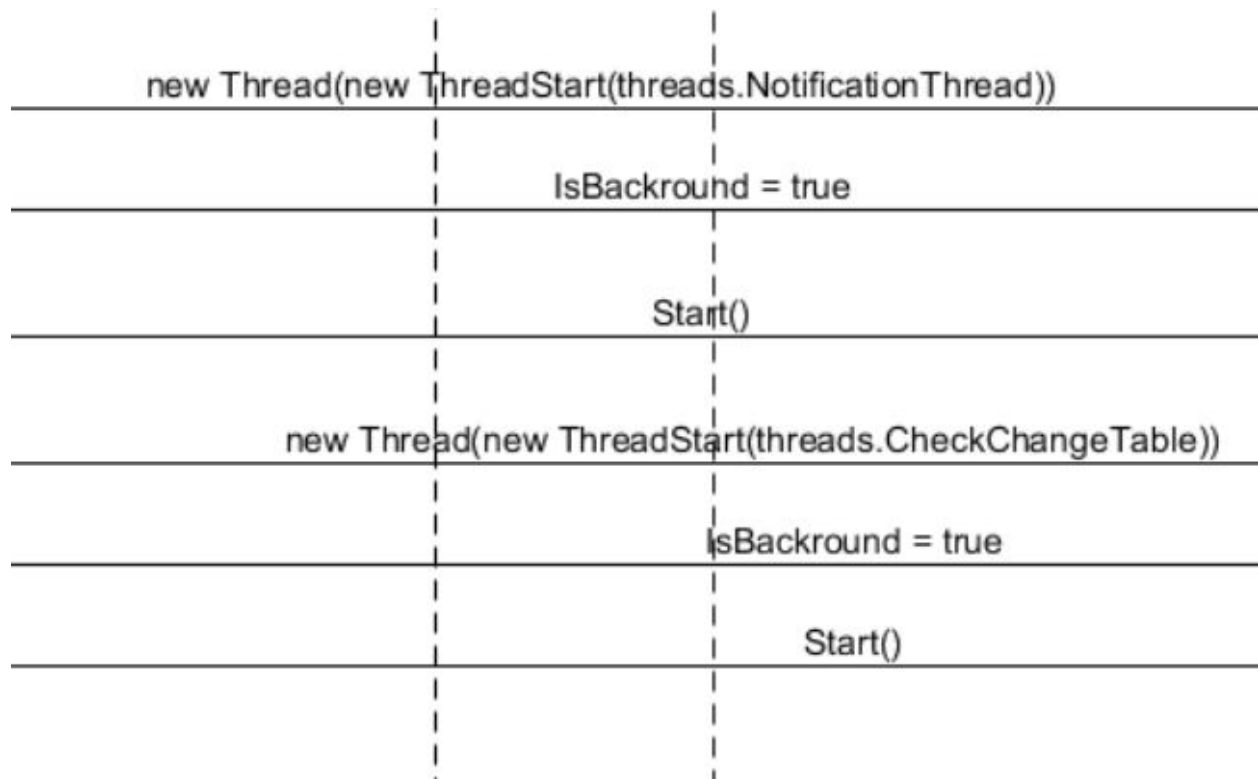
Section 2



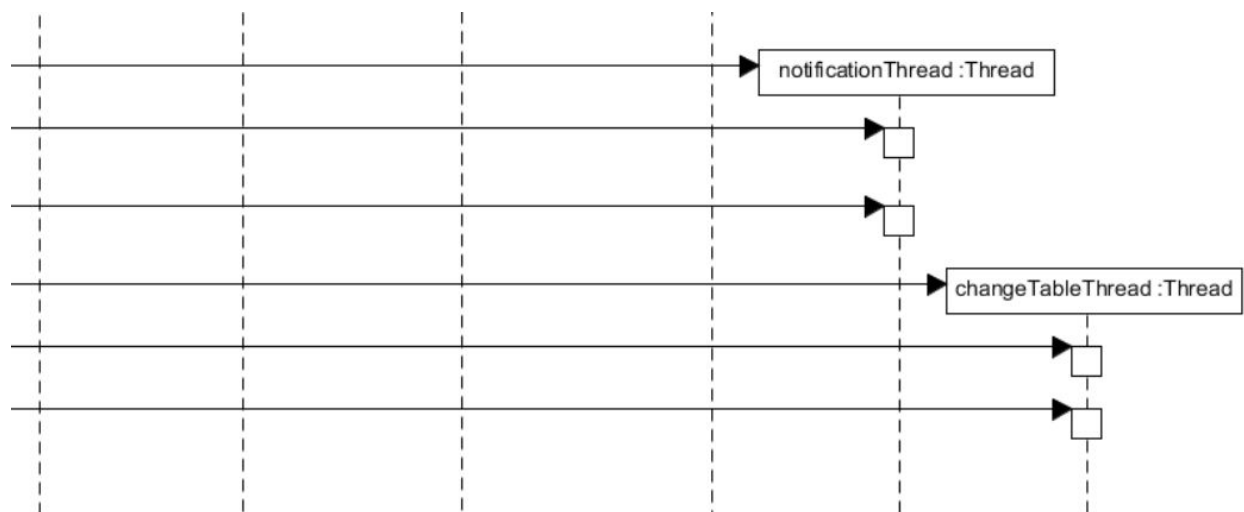
Section 3



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

¹Business 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

¹ 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

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:

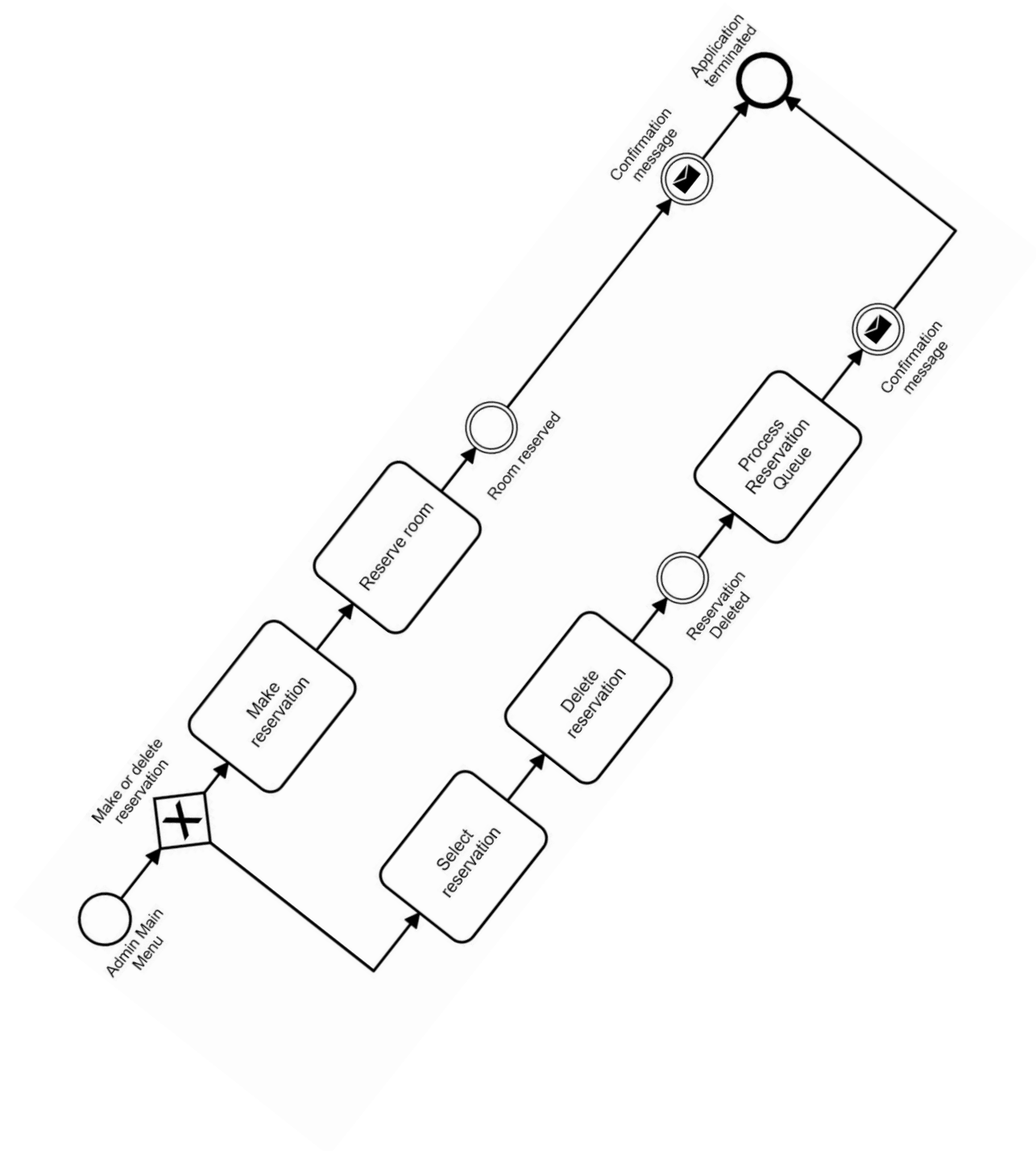
- Reservation Exists

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.

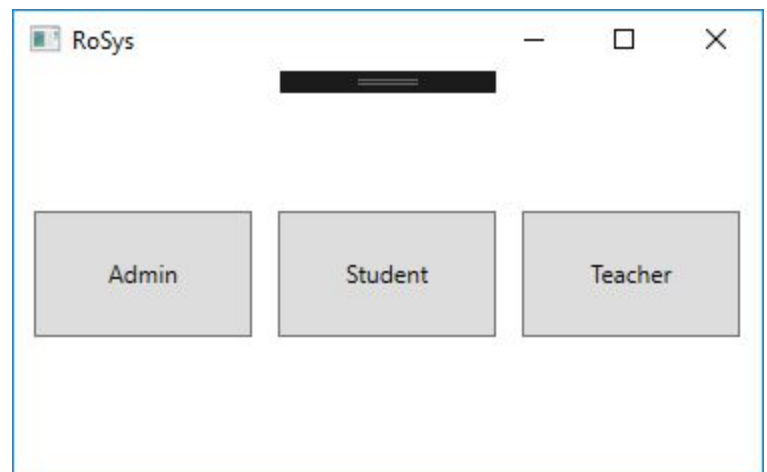
Our focus with the Core is the ability to integrate into other systems, so we made sure to use interfaces for some of the main resources with the program.

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

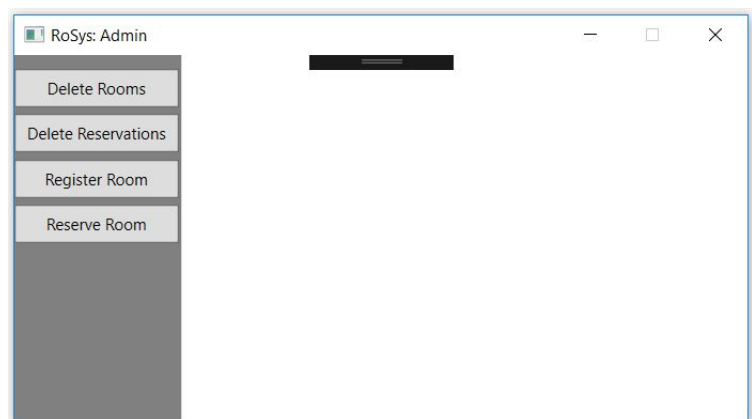
When running the program, there are, initially, 3 options, as seen in Picture 1. This is essentially our “login”, where selection is made of which type of user will interact with the program.

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

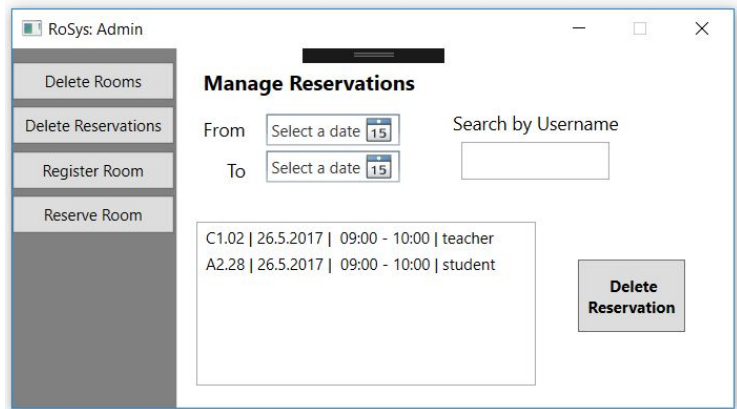
Picture 2.



Picture 3.

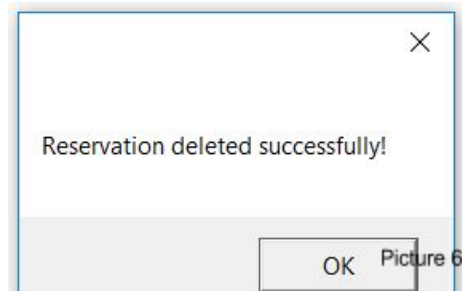
room from the program and database.

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.



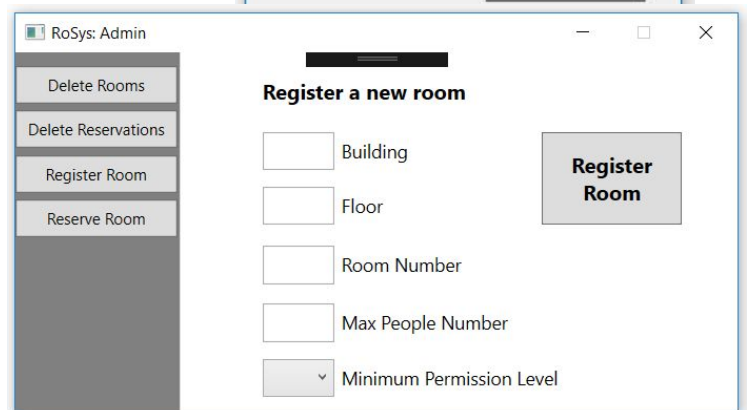
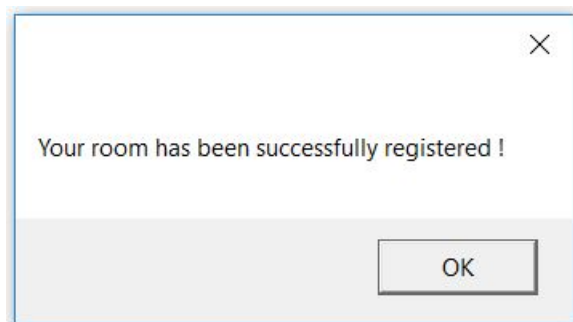
Picture 4.

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).



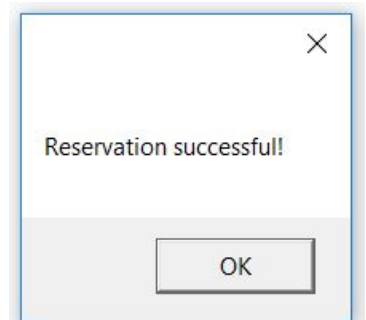
Picture 6.

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

Picture 8.

needs to exist in the database and repository and the

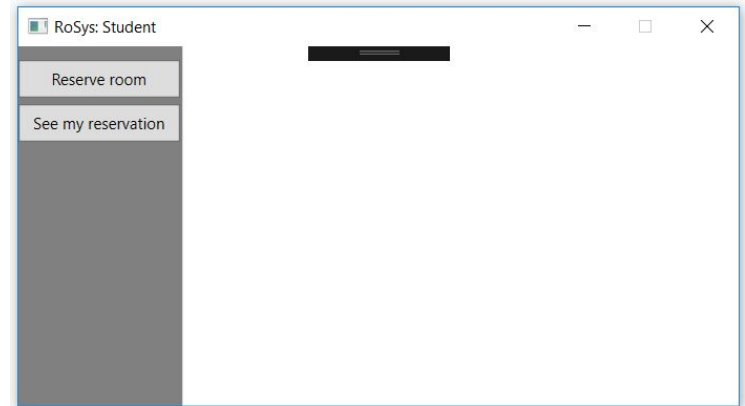


text, “Enter a username”, must be cleared before the admin can input the username. The admin also needs to select which room he wants the reservation to be in. If everything is done right, then a message will appear, like in Picture 9.

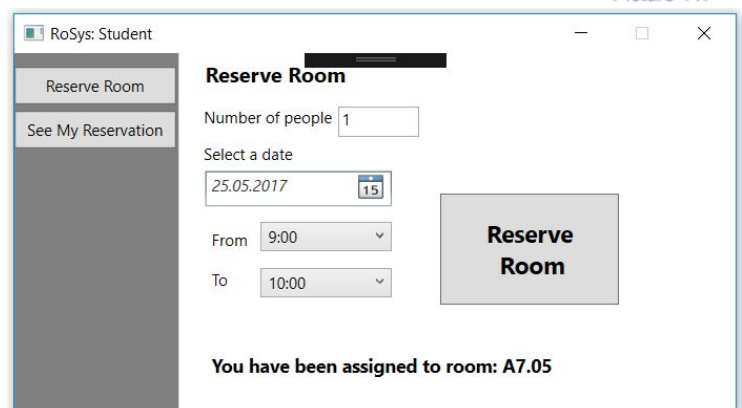
By pressing the “Student” button in the menu, Picture 1, the “RoSys: Student” window pops up, Picture 10.

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.

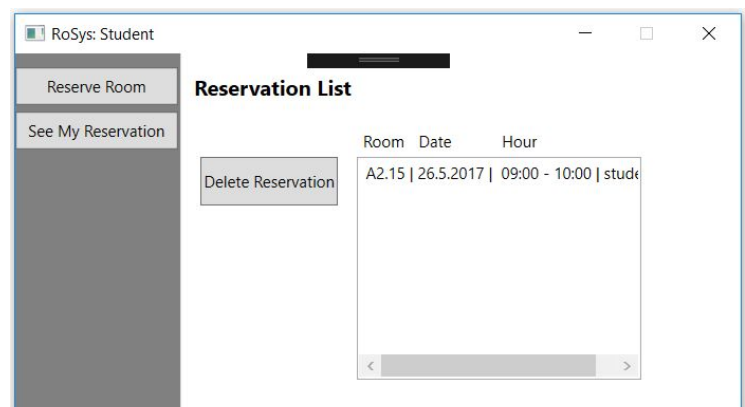
You already have a room booked, at this time.
Picture 13.

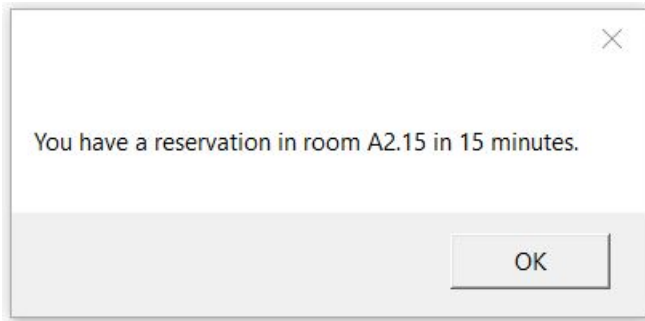
No rooms available

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.

```