Hotel Management System

Group 5

Christian Hougaard Pedersen (315269)

Nina Wrona (315202)

Justina Ieva Bukinaitė (315199)

Karolis Sadeckas (315225)

Supervisor: Steffen Vissing Andersen

VIA University College

56.571

Software Engineering

2. semester

01-06-2022

Table of content

1. Abstract

Beavers Hotel contracted Group 5 to create a management system, to be used in their daily operations. The requirements of the system entailed the ability to manage rooms, guests and bookings. Following that, the system should function as a booking platform for the guests of the hotel.

The system is implemented as a client server system, with the server containing a database to ensure persistence of data. The program is coded using Java, and the graphical user interface is created using JavaFX.

The database is implemented using PostgreSQL and allows user-access due to the implementation of a *JDBC* database driver.

Throughout the implementation of the system, several specific design patterns were used, in order to prevent code redundancy and to ensure maintainability of the system.

The current product is well functioning and covers the critical and most high priority requirements. In the application the guests can create a new account and later on log in as well as book available amenities and view and edit their personal details. Both the hotel manager and receptionist have separate views, where different functionality can be achieved. For example, the Hotel Manager is able to add and edit the room data, while the receptionist can edit the bookings and guest personal details.

1 Introduction

The chosen project theme for the second semester project was focused on hotel management and booking systems. The central principle of the project was to create a hotel management and booking system for a hotel by name “Beavers Hotel”, having mainly the stakeholder’s representative, the product owner as a guide to the necessary decisions throughout the project period.

Currently the hotel by the name “Beavers Hotel” is using a basic Microsoft Office as well as pen and paper to manage their rooms, booking and guest data. Furthermore, the guests have no available platform to view the offered amenities and/or book a room. At present the guest is required to call to the hotel to book accommodations or/and request further information about the hotel, their booking or edit of the booking. Our aim for this project is to provide a working system to the hotel so that the guests would be able to book, view and possibly edit their bookings. Additionally, the management process would be achieved in a more structured fashion and overbookings would be avoided.

The goal is to create a client-server system, where both the administration of the hotel and the guests remain on the client side only separated by different accounts and the server that acts as data provider and manipulator that stores the relevant data in its database. Additionally, in this version of the system the implementation of the website is excluded due to time and expertise constraints, but it may be considered in the future of the project.

This report will cover the thought process of creating the mentioned product as well as the implementation and testing. Starting with the Analysis, it will cover stakeholders voiced requirements and the said requirements put into use case scenarios. That will be followed by the Design segment, where the consideration for systems architecture and design choices will be portrayed. The implementation section will go into detail of certain chosen patterns and why they were chosen and on top of that the tests will be illustrated and then explained in the results and discussions segment. Lastly the conclusion will be drawn to portray the accurate picture of the goals and results of the project.

2 Analysis

As mentioned in the introduction this section will cover in detail the requirements that were provided by the stakeholders. Additionally, to identify a clear path for the user using the system, brief as well as fully dressed use case descriptions will be included.

2.1 Overview of the required system

The stakeholders requested a new system that will include both a booking process for the guests and the management of the rooms and bookings made by the guests.

According to the stakeholders the system should have an option for the guest to create an account to book room(s), as well as edit their personal details and change the bookings if the guests require. The guest should be able to see all available rooms when they input the dates, alongside the prices and room details (number of beds and room type).

The stakeholder has expressed many requests for the management side of the system, where firstly both the hotel manager and the receptionists have separate from each other access to the system with different views and functionalities. The receptionist should be able to view the current and future bookings and their statuses as well as edit the bookings and guest personal details. Moreover, the receptionist should be able to check in the booking on the day of the guest's arrival as well as check out the booking when the guest leaves.

The hotel manager on the other hand should have access to the rooms' data and should be able to add/edit them. Alongside that, they should have a view of all bookings including the archived (the bookings that have been concluded) and canceled. Moreover, it should also be included for the hotel manager to be able to view guest details, although the editing functionality is only meant for the receptionist. Lastly, for convenience both hotel manager and the receptionists should be able to cancel the booking by a guest's request.

Furthermore, if time restraints will not be an issue, the hotel manager should be able to see certain statistics of previous bookings as well as the financial layout of the hotel. Additionally, filtering in the guest overview and booking overview may be added to have quicker access to specific data and that may be also included when booking a room. On top of that, guests should be able to leave a review and view the previous opinions of preceding guests.

2.2 Requirements

Following the given overview of what is needed for the future system, the requirements for the system were laid out in a product backlog form as user stories to clearly understand each requirement and reasons for having it. Also, each requirement was given a priority type (Critical, high and low) in order to focus on the most needed aspects for the system and if those are achieved in time, the lower in priority list can be taken and implemented. Alongside that, an estimated number of hours was considered for each requirement, in order to have a better understanding of how long it may take to achieve the goal.

2.2.1 Functional Requirements

1. As a guest I want to book available rooms, in order to ensure that I can stay in a hotel at a given time interval, in a specific room.
2. As a hotel manager, I want to be able to view the room number, type, number of beds in order to have an overview of the hotel services.
3. As a hotel manager, I want to be able to remove the hotel room, in order to make changes, if the room is being renovated.
4. As a hotel manager, I want to be able to change the hotel room details, in order to make changes, if there is a new feature added to a room.

5. As a hotel manager, I want to be able to add a hotel room, in order to have all rooms that are available for booking in the system.
6. As a receptionist I want to be able to check in the customer, in order to change the status of the booking.
7. As a receptionist I want to have access to the customer's first and last name, phone number and an email of customers that booked a room, in order to make sure that they are not fake.
8. As a receptionist I want to be able to change the personal information of hotel's customers, in order to keep all the information up to date.
9. As a receptionist I want to be able to change the booking information, in order to better accommodate our customers' wishes.
10. As a hotel manager I want to be able to see all the bookings made in the system in order to keep track of room usage.
11. As a hotel manager I want to be able to see all the clients registered in the system in order to keep track of people making the bookings.
12. As a hotel manager I want to be able to cancel the booking made by the hotel's customers, due to e. g. water damage.
13. As a guest I want to see information about my booking, in order to check the details that I forgot about.
14. As a guest I want to be able to log in to the system to see all my bookings in order to keep my information more secure.
15. As a guest I want to be able to see the details of all my bookings in order to check the reliability of my information.
16. As a guest I want to be able to cancel, in order to inform the hotel that my plans have changed.
17. As a guest I want to be able to make changes to personal information in my booking, in order to inform the hotel that my email or phone number has changed.
18. As a guest I want to know the price of the room I am booking, in order to know the cost in advance
19. As a guest I want to be able to search the rooms, by filtering the price, the people count and type, in order to find the best suited room for my needs.

20. As a guest I want to be able to book a conference space, in order to have business meetings.
21. As a receptionist, I want to be able to login in order to ensure that I am the only one able to access the receptionist part of the system.
22. As a Hotel Manager, I want to be able to login in order to ensure that I am the only one able to access the Hotel Manager part of the system.
23. As a guest I want to be able to search the rooms, by filtering balcony availability and size, in order to find the best suited room for my needs.
24. As a guest I want to be able to search the rooms, by filtering the location, in order to find the best suited room for my needs.
25. As a guest I want to be able to book a conference space, in order to have business meetings.
26. As a hotel manager, I want to be able to enter the bills and the expenses, in order to have financial details saved in the system.
27. As a hotel manager, I want the customers to be able to cancel the booking up to two weeks before the stay, in order to rent out the room to a different customer(s).
28. As a hotel manager I want to be able to get the past data about customers and rooms, in order to analyze the collected data.
29. As a guest I want to be able to request deletion of the archived information about my booking, in order to protect my personal information.
30. As a hotel manager, I want to be able to track the monthly budgeting (Hotel income and expenses), in order to make sure that all financial details are correct.
31. As a hotel manager I want to be able to see the statistics of the booking history, in order to find a room for improvement.
32. As a hotel manager I want to be able to differentiate the prices throughout the year depending on the season, in order to make more profit.
33. As a guest I want to be able to leave a review and a rating up to 5 stars, in order to share my experience from my stay.
34. As a guest I want to be able to see other customers’ reviews, in order to judge the reliability of the hotel.

35. As a guest I want to be able to send a complaint(s), in order to get help from the hotel manager.
36. As a hotel manager, I want to be able to receive the customers' complaints, in order to know which hotel room needs the requested service.
37. As a guest, I want to be able to choose a breakfast or All-inclusive option, in order to receive requested service.
38. As a hotel manager I want to be able to add discounts to package deals that include room booking alongside a conference space booking, in order to encourage other businesses to book conference rooms and regular rooms for their employees.
39. As a hotel manager I want to be able to give discounts to customers, in order to get a better rating review.
40. As a guest I want to be able to view current events in the hotel as well as in the city, in order to find activities to participate in during the stay.
41. As a guest I want to be able to see the features of the hotel (gym, pool, parking) in order to know what features are available for my use.
42. As a guest I want to be able to view the pictures of the available rooms in order to make sure I want to book them.
43. As a guest I want to be able to see prices for different age groups in order to calculate my payment.
44. As a guest I want to be able to access the contact section of the hotel owner as well as contact information for the hotel, in order to find their email and phone number.
45. As a guest I want to be able to switch the language between Danish and English
46. As a guest I want to be able to choose a type of bed (single or double).

2.2.2 Non-functional requirements

47. Only if the customer agrees to the terms and conditions the system will confirm the booking, to make sure that the customer follows the hotel rules as well as the GDPR legislation.
48. A customer can only change the details or cancel the booking 2 weeks prior to the reserved date of stay.
49. Customer's data will be deleted after 6 months, in order to follow the GDPR legislation.
50. The system will contain different prices for children, adults and seniors.
51. The system will output a map with the pin of the hotel location (It may be either a picture or interactive map).
52. The system will have links to flight bookings, car rentals, spa offers.
53. The booking system will have 30 rooms for booking.
54. The system will be styled using css.

2.2.3 Use case diagram

When the requirements were organized, the use case diagram was drawn in order to demonstrate how users of the system may interact with the program. The portrayal is very general, but it shows what actors will be participating in the system as well as the general access and functionality each actor will be able to achieve.

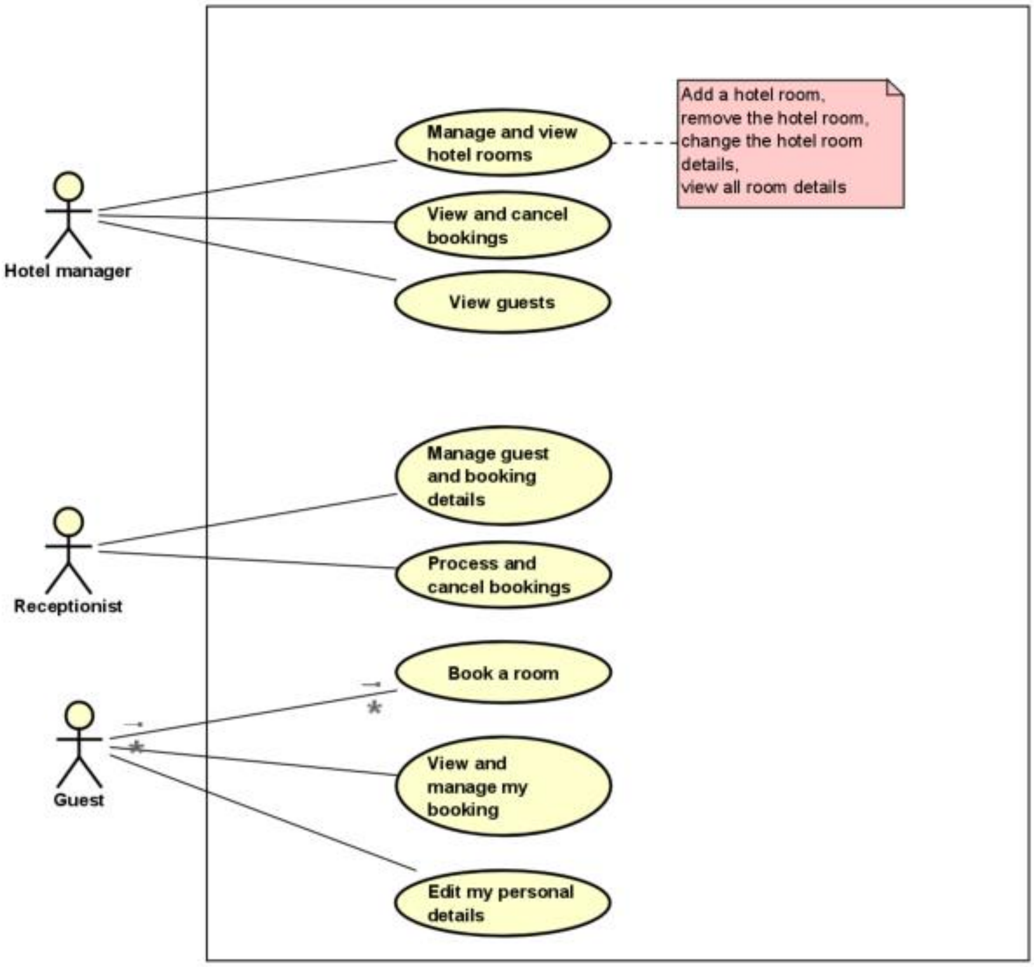

Hotel manager
Manage and view hotel rooms
Add a hotel room, remove the hotel room, change the hotel room details, view all room details
View and cancel bookings
View guests
Receptionist
Manage guest and booking details
Process and cancel bookings
Guest
Book a room
View and manage my booking
Edit my personal details
*
*

Figure 1 - Use Case Diagram

2.3 Use cases

After the use case diagram was created, in order to start visualizing the possible paths for the actors, brief format use case descriptions were created that very shortly cover how the actor will be interacting with the system on different occasions.

List of use cases:
Manage guest and booking details
Approve the booking
Book a room
View and manage my booking
View all bookings
View all guests
Cancel booking
Log in
Register

2.3.1 Use case descriptions (brief format)

- <u>Manage guest and booking details (as a receptionist):</u>
  The receptionist enters their account and by choosing any of the bookings that they can view, they can choose either to view booking details or guest details that are attached to the said booking. When/if the receptionist enters one of the detailed views, they are able to edit some of the given information as well as cancel the booking.

- <u>Approve the booking (as a receptionist):</u>

The receptionist chooses a booking and checks it in.

- Book a room (as a guest):

  Guests enter their account and go to book a room, where they enter the dates of their choosing and choose the available room.

- View and manage my booking (as a guest):

  Guests enter their account and go to bookings where they can view all their bookings.

- View all bookings (as the hotel manager):

  The hotel manager enters their account and goes to bookings, where all existing bookings are available to view.

- View all guests (as the hotel manager):

  The hotel manager enters their account and goes to the guest, where all existing guests and their details are available to view.

- Cancel booking (as the hotel manager):

  The hotel manager enters their account and goes to bookings. They choose a booking and cancel it.

- Log in (as a guest):

When the system is opened, the guest goes to log in and enters their username and password.

- Register (as a guest):

  When the guest wants to create a new account, they enter the registration and input the username and password of choice and their personal information.

After brief descriptions were created the use case tables were constructed for each use case and in the tables, it goes into detail what paths need to be taken to achieve a certain scenario as an actor.

USE CASE	**Manage guest and booking details**
Summary	The receptionist sees a list of all bookings. By selecting any booking, they can either select to see detailed guest information or booking information and edit them.
Actor	Receptionist
Precondition	
Postcondition	Booking details were successfully changed.
Base Sequence	2. Enter the system as a Receptionist. 3. System displays all bookings that were added to the system with detailed information: Booking ID, start date, end date, guest username, room ID, booking status. 4. The Receptionist selects a booking 5. The Receptionist selects to view detailed room information for specific room booking. 6. The system displays information. 7. The Receptionist changes the information. 8. The receptionist confirms changes. 9. The system validates filled information. 10. The system confirms changes.

USE CASE	**Manage guest and booking details**
Alternative Sequence	*At any time during step 1,3,5,7 Receptionist cancels 1. The use case ends. 8a. The system fails to validate information 1. The System shows an error and prompts for new information 2. The use case starts from step (4) 3a. Receptionist wants to edit guest information for a specific room booking. 1. The receptionist selects to view guest information for specific room booking. 2. Use case proceeds from step (5) 4a. Receptionist wants to cancel booking 1. The booking is canceled. 2. Use case proceeds from step 3.
Note	

USE CASE	**Approve the booking**
Summary	The receptionist searches for a specific booking and confirms a booking.
Actor	Receptionist
Precondition	Booking is created.
Postcondition	Booking is successfully confirmed.
Base Sequence	1. Enter the system as a Receptionist. 2. System displays all bookings that were added to the system with detailed information: Booking ID, start date, end date, guest username, room ID, booking status. 3. The Receptionist selects a specific room booking and makes a check-in. 4. System validates the information. 5. System displays changed information.

USE CASE	**Approve the booking**
Alternative Sequence	3a. Receptionist enters the bookings that have been checked in 1. Receptionist chooses the booking that needs to be checked in 2. The use case proceeds from step 4.
Note	For easier selection, possibility to select to show only bookings that are not approved yet.

USE CASE	**Book a room**
Summary	Guest opens the system, picks a room he wants to book and fills out the booking details.
Actor	Guest
Precondition	There must be available rooms in the hotel.
Postcondition	A room is booked.
Base Sequence	1. Enter the system as a guest. 2. The guest selects the start and end dates of the stay. 3. The system shows all available rooms for selected dates. 4. The guest chooses an available room. 5. The guest confirms booking with the selected room from a list. 6. The system creates a booking.
Alternative Sequence	*At any time during step 1,2,5 The Guest cancels 1. The use case ends
Note	

USE CASE	**View my booking**
Summary	Guest searches for his booking and views all information regarding his booking.
Actor	Guest

USE CASE	**View my booking**
Precondition	The guest must have at least one booking.
Postcondition	Information about booking successfully viewed.
Base Sequence	1. Enter the system as a guest. 2. Enter the booking overview. 3. See all active and not active bookings that were made in the past. 4. System shows all booking
Alternative Sequence	
Note	

USE CASE	**View all bookings**
Summary	Hotel manager sees a list of all bookings that were made in a hotel.
Actor	Hotel manager
Precondition	
Postcondition	Successfully viewed all the bookings made by a hotel.
Base Sequence	1. Enter the system as a Hotel manager. 2. View bookings. 3. System displays all bookings that were made in a system.
Alternative Sequence	
Note	Hotel manager can select to hide all canceled bookings.

USE CASE	**View all guests.**
Summary	Hotel manager sees a list of all guests that were registered in a hotel.
Actor	Hotel manager
Precondition	
Postcondition	Successfully viewed a list of all guests.
Base Sequence	1. Enter the system as a Hotel manager. 2. View guest list. 3. System displays all guests that were made in a system.
Alternative Sequence	
Note	

USE CASE	**Cancel a booking**
Summary	Hotel manager sees a list of all bookings and cancels selected bookings.
Actor	Hotel manager
Precondition	
Postcondition	Successfully canceled booking.
Base Sequence	1. Enter the system as a Hotel manager. 2. The system shows all bookings in a system. 3. The Hotel manager selects a specific booking and cancels it. 4. System confirms it.

USE CASE	**Cancel a booking**
Alternative Sequence	*At any time during 1, 3 Hotel manager cancels 4a. If the booking is already canceled, an error will be shown, redo from step 7. 1. The use case ends
Note	Hotel manager can select to hide all canceled bookings.

USE CASE	**Log in**
Summary	A user logs in to his account and can view booking and create new ones.
Actor	Guest/Hotel Manager/Receptionist
Precondition	The guest must have a username and password/account
Postcondition	Successfully logs in
Base Sequence	1. Enter Log in view for the guest 2. Enter Username and password 3. Log in
Alternative Sequence	1. Enter Log in view for the admin (hotel manager/receptionist) 2. Enter Username and password 3. Log in
Note	

USE CASE	**Register**
Summary	A guest creates a new account
Actor	Guest
Precondition	
Postcondition	Successfully creates new user/account
Base Sequence	1. Enter Register view 2. Enter username, password and other personal details 3. Register
Alternative Sequence	
Note	

2.4 System sequence diagram

To elaborate on the Use Case dealing with booking a room, a *System Sequence Diagram* has been created, as shown below:

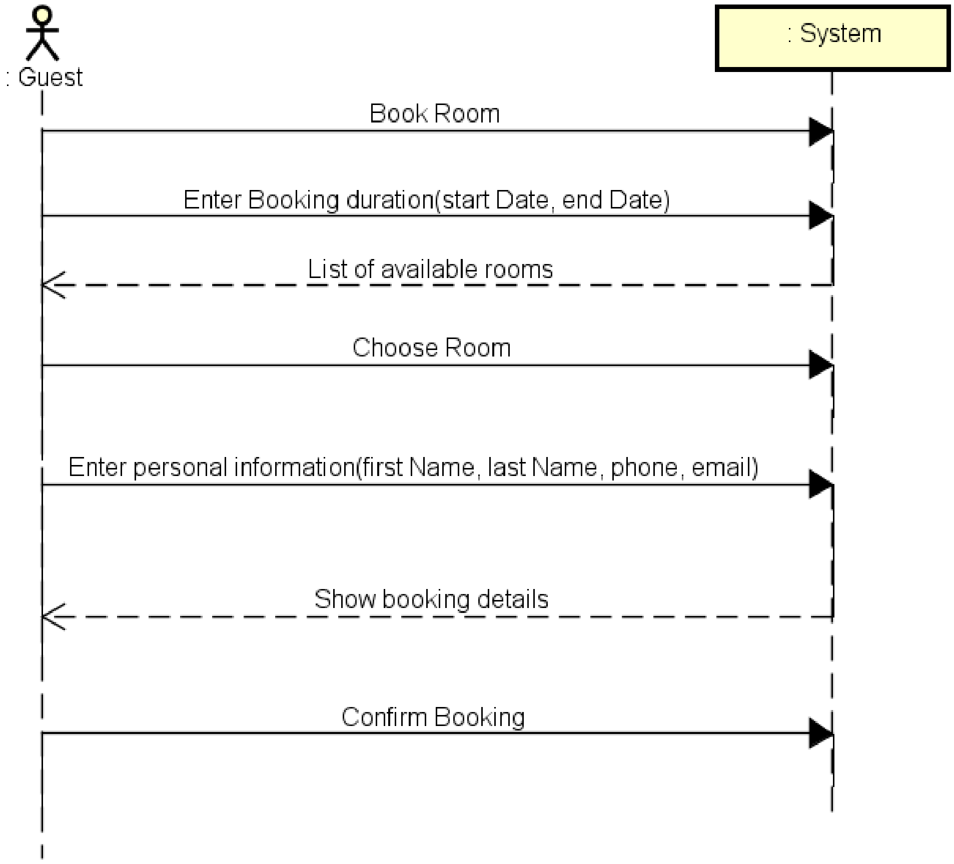

: Guest
: System
Book Room
Enter Booking duration(start Date, end Date)
List of available rooms
Choose Room
Enter personal information(first Name, last Name, phone, email)
Show booking details
Confirm Booking

Figure 2 - System Sequence Diagram

This diagram describes the flow of the operation of booking a room in the system by showing the relationship between user actions and the system responses.
The *Guest*-actor enters the start- and end-date of the booking, to which the system responds with a list of all rooms available for the given period. The next step is for the *Guest* to choose a room from the list, and the system will respond with the details of the booking to be created. The *Guest* then confirms the booking, and it is created in the system.

2.5 State machine diagram

In some cases, entities not only depend on the input, but also can be a subject of the preceding state. The history of the state can be best depicted by a state diagram.

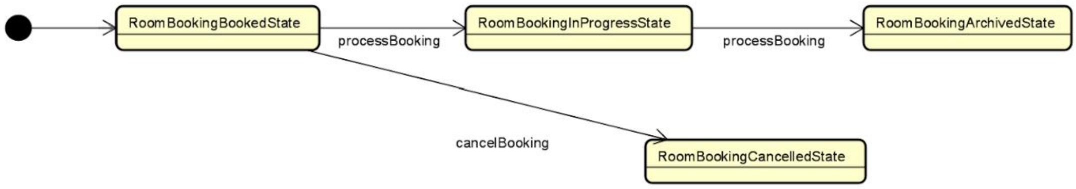


Figure 3 - State Machine Diagram

The current system has one case, where the entity is dependent on its previous state. Specifically, when the booking has been booked and maintains a “Booked” state, the state may be changed to “In progress” or it can be “Canceled”. On the other hand the state “Archived” may only be set if the previous state was “In Progress” and the entity with said state cannot be in “Canceled” state.

2.6 Domain model

After use case structuration and having in mind the system sequence diagram and state machine diagram, the domain model has been created. It helps to visualize the system and will be a starting point when the class diagram will have to be considered since it includes both behavior and data of the upcoming system.

The domain model contains three actors that interact differently with Hotel Room and Room Booking entities. For example, the Receptionist manages the Room Booking, which has a Hotel Room, therefore the receptionist has access to not only the bookings and can change them, but also can change the room number. Whereas the Guest entity has Room Booking, therefore the only functionality the actor can achieve is canceling. The Hotel Manager manages Hotel Room, therefore the actor can make changes to the Hotel Room entities, but as shown in the diagram it can only cancel the

Room Booking entities. All entities except Receptionist and Hotel Manager have their own properties, so they could be manipulated depending on which actor is changing it.

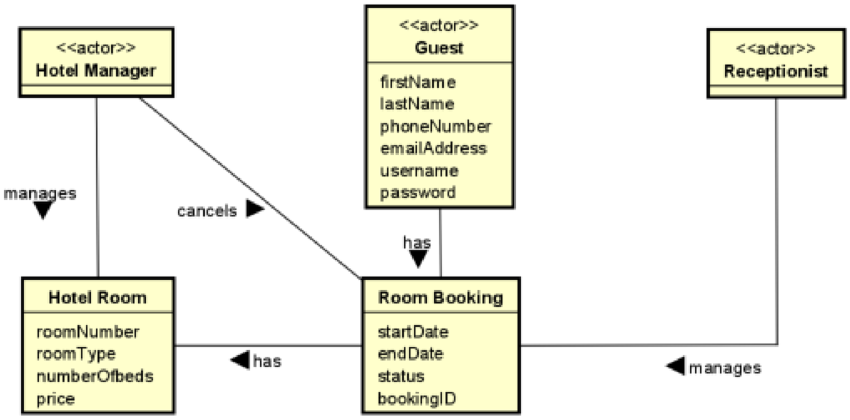


Figure 4 - Domain Model

3 Design

While the focus of the analysis phase was to map the problem domain, and by doing this, transforming the customers requirements into Use Cases to better grasp the transformation of these into a functional piece of software, the goal of the design phase is to plan out the system using the Domain Model.
As the source code for the system is written in Java, which is an Object-Oriented programming language, class diagrams are used to visualize the classes in the system, and the relationships between them.

3.1 Sockets

Per the customer's wishes, the system needs to be able to handle multiple users at once, all of them having access to the same data meaning that it needs to be a client/server system.
When designing the client/server part of the system, it was decided to use TCP sockets to handle the connection between the server and the clients. This approach was chosen for the control it gives regarding the data that is transported in comparison to for example RMI.
When using sockets, these will take care of the communication between the two notes by sending the requested data from the server to client-side. The sockets being an endpoint for sharing messages, it uses a combination of ports and IP addresses. For this case it will be mainly for exchanging room data as well as user information. That will be enforced by certain users having different access to the data and ways of manipulating it.

On our server side we have a HotelServer class in which we create an unbounded server socket and start listening for connections to be made for selected ports and after connection is received a separate thread is created for individual clients.

On the client side we have a HotelClient class in which we make connections to selected servers and make direct input and output streams to a server to have a way to transfer data.

Lastly, to avoid cases where both sides are waiting for a reply and none of them sending it, the data protocol is used for both server and client. For sending information we are using objects transferred to Json format. After receiving in client or server side we transfer back to an object and follow instructions encoded in the object what data must be received from a server.

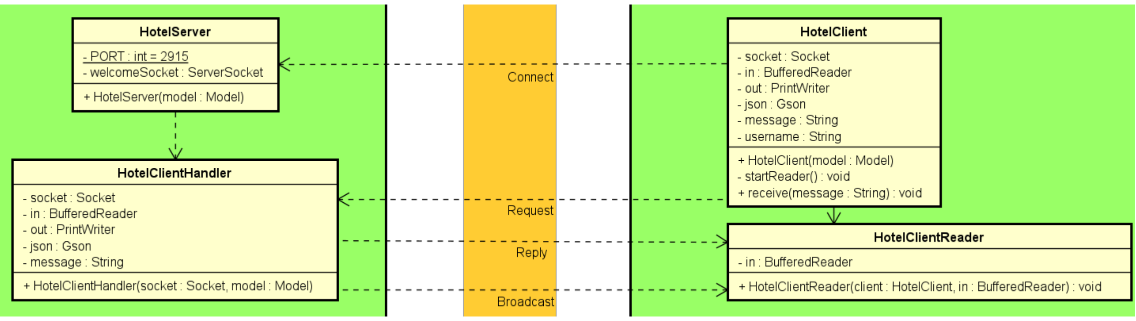


Figure 5 - Socket connection

3.2 Architecture

To structure the classes in the system, it was chosen to split the system into packages across both the server and the client.

3.2.1 Server

The server side is divided into 3 packages: network, model, and mediator.

3.2.1.1 Network package

The network package contains only one class, MyDatabase, which handles communication with the database.

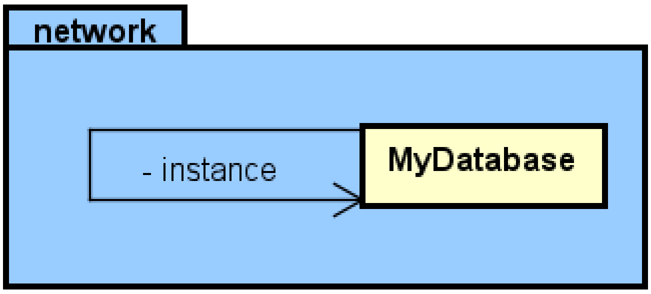


Figure 6 - network package

3.2.1.2 Model package

The model package contains the logic of the system, including the model interface, and the adapter for the database.

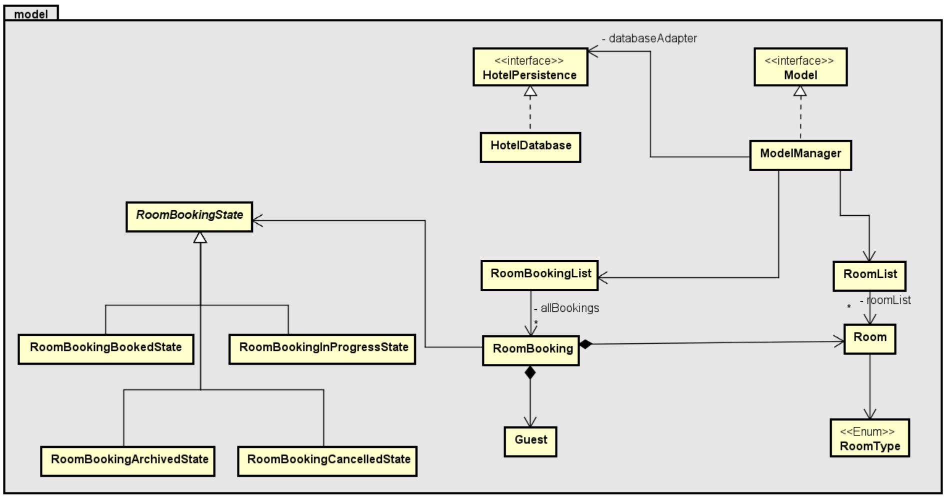


Figure 7 - Model package

3.2.1.3 Mediator package

The mediator package serves as the link between the server and the client and contains the HotelServer and HotelClientHandler classes which handles the communication with the client, as well as four transfer-classes, used to transport information to the client.

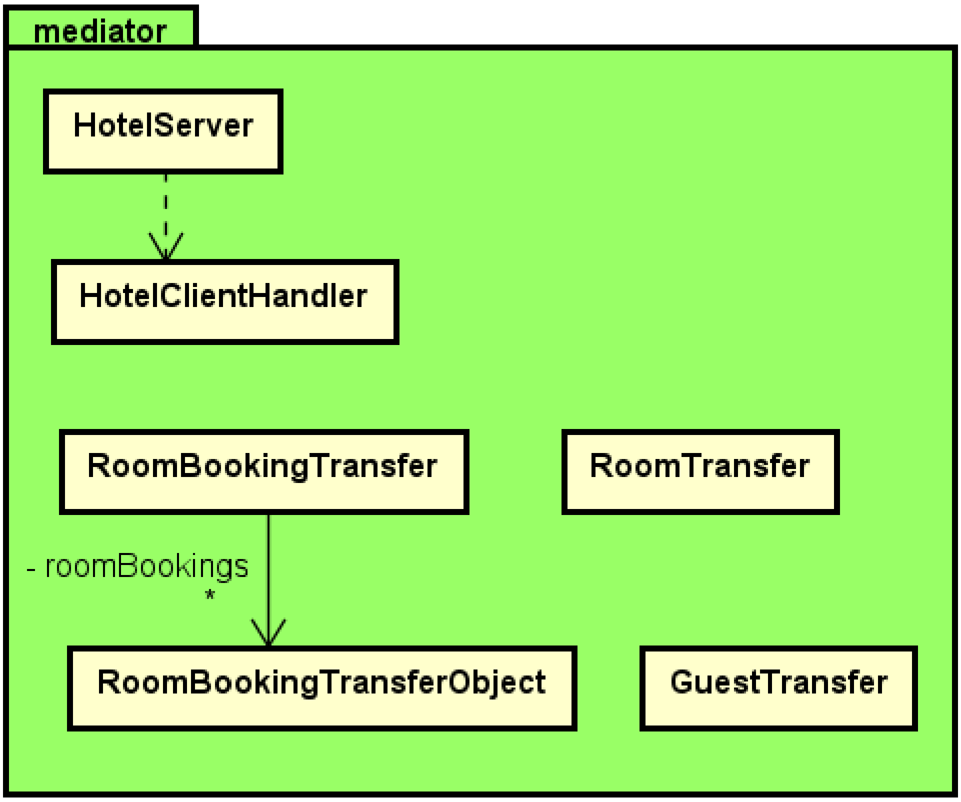


Figure 8 - Mediator package

3.2.2 Client

The client has a general structure much like the server, with the differences being that the model does not have a network package, as communication with the database is handled by the server, and the addition of the view- and viewModel packages, as the client has a GUI.

3.2.2.1 Mediator package

As with the mediator package on the server, this package handles the network communication. The client-side communication is handled by the HotelClient and HotelClientReader classes, and the four transfer classes.

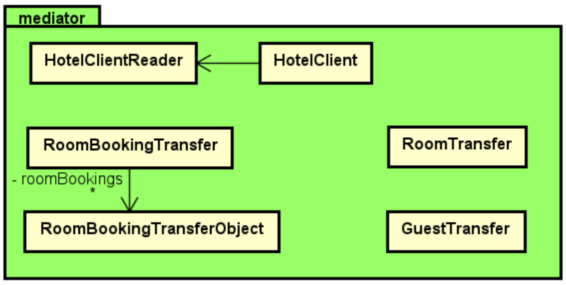


Figure 9 - Mediator package

3.2.2.2 Model package

The client-side model package is almost identical to the one found on the server, with the omission of classes not needed in the client, as most of the business logic is handled by the server.

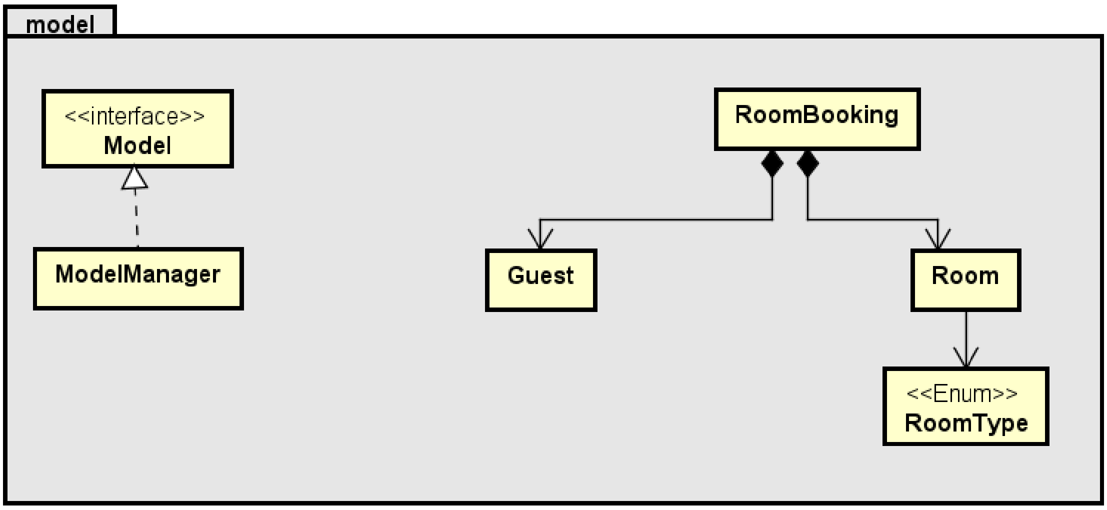


Figure 10 - Model package

3.2.2.3 ViewModel package

The viewModel package contains sixteen viewModel-classes used to mediate between the model and the view packages. Also present in this package is the Helpers sub-package, which holds five classes used to store information.

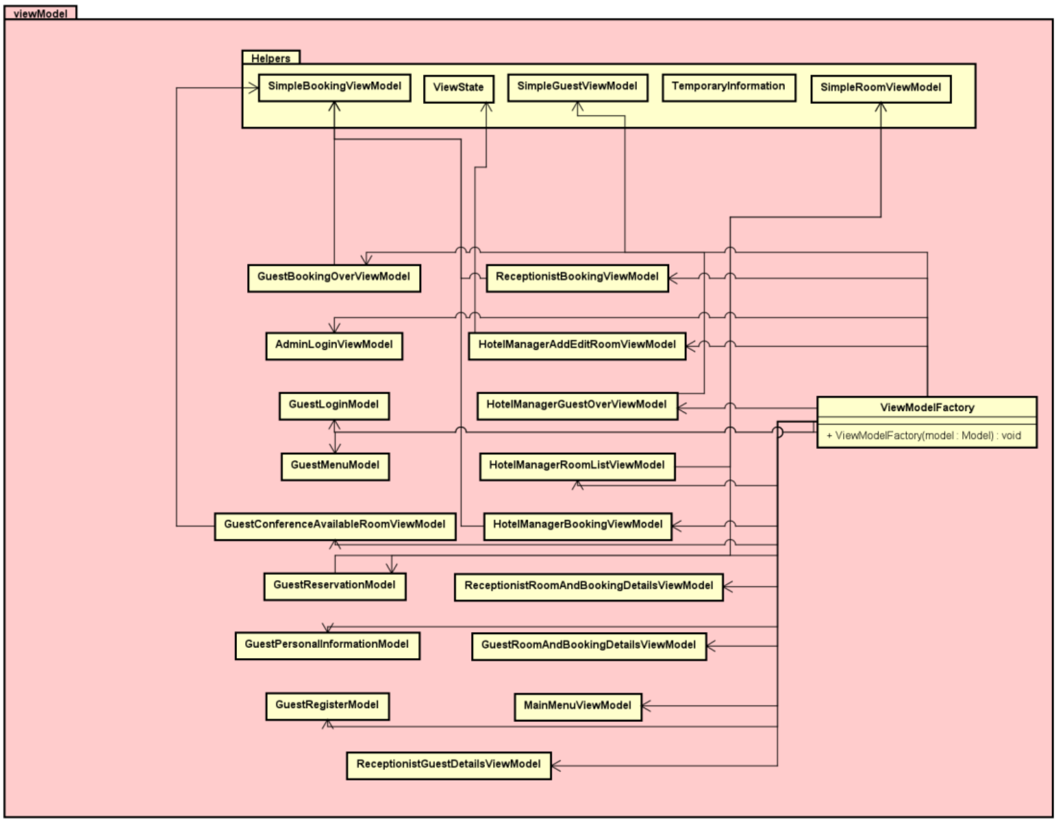

viewModel
Helpers
SimpleBookingViewModel
ViewState
SimpleGuestViewModel
TemporaryInformation
SimpleRoomViewModel
GuestBookingOverViewModel
ReceptionistBookingViewModel
AdminLoginViewModel
HotelManagerAddEditRoomViewModel
GuestLoginModel
HotelManagerGuestOverViewModel
ViewModelFactory
+ ViewModelFactory(model : Model) : void
GuestMenuModel
HotelManagerRoomListViewModel
GuestConferenceAvailableRoomViewModel
HotelManagerBookingViewModel
GuestReservationModel
ReceptionistRoomAndBookingDetailsViewModel
GuestPersonalInformationModel
GuestRoomAndBookingDetailsViewModel
GuestRegisterModel
MainMenuViewModel
ReceptionistGuestDetailsViewModel

Figure 11 - ViewModel package

3.2.2.4 View package

The view package contains the ViewHander, ViewController, and ViewCreator classes as well as all FXML-files needed to run the program. The FXML files, and their corresponding controllers are placed in the sub-package GUI.

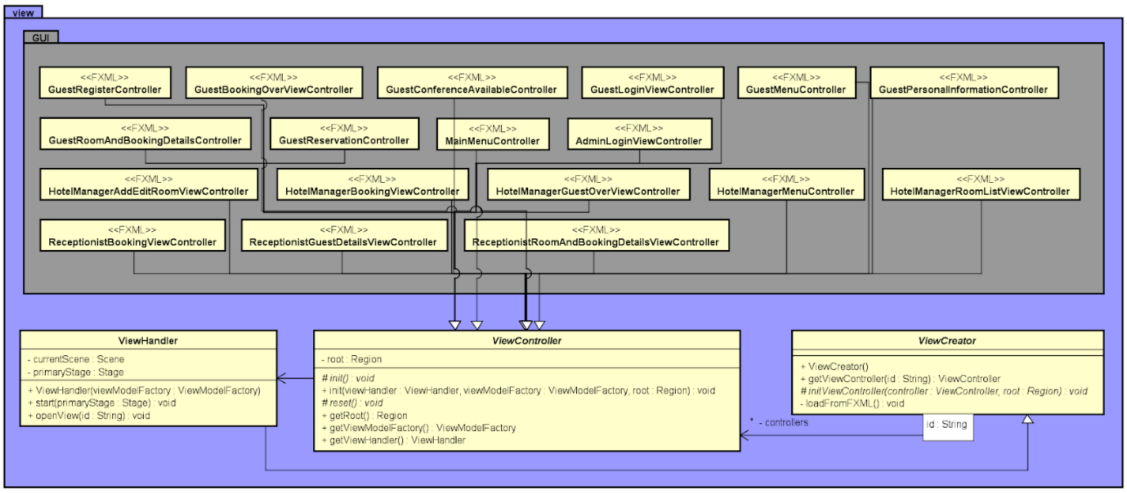

view
GUI
<<FXML>>
GuestRegisterController
<<FXML>>
GuestBookingOverViewController
<<FXML>>
GuestConferenceAvailableController
<<FXML>>
GuestLoginViewController
<<FXML>>
GuestMenuController
<<FXML>>
GuestPersonalInformationController
<<FXML>>
GuestRoomAndBookingDetailsController
<<FXML>>
GuestReservationController
<<FXML>>
MainMenuController
<<FXML>>
AdminLoginViewController
<<FXML>>
HotelManagerAddEditRoomViewController
<<FXML>>
HotelManagerBookingViewController
<<FXML>>
HotelManagerGuestOverViewController
<<FXML>>
HotelManagerMenuController
<<FXML>>
HotelManagerRoomListViewController
<<FXML>>
ReceptionistBookingViewController
<<FXML>>
ReceptionistGuestDetailsViewController
<<FXML>>
ReceptionistRoomAndBookingDetailsViewController
ViewHandler
- currentScene : Scene
- primaryStage : Stage
+ ViewHandler(viewModelFactory : ViewModelFactory)
+ start(primaryStage : Stage) : void
+ openView(id : String) : void
ViewController
- root : Region
# init() : void
+ init(viewHandler : ViewHandler, viewModelFactory : ViewModelFactory, root : Region) : void
# reset() : void
+ getRoot() : Region
+ getViewModelFactory() : ViewModelFactory
+ getViewHandler() : ViewHandler
ViewCreator
+ ViewCreator()
+ getViewController(id : String) : ViewController
# initViewController(controller : ViewController, root : Region) : void
- loadFromFXML() : void
* - controllers
id : String

Figure 12 - View package

3.2.3 Model-View-ViewModel

Following the SOLID design principles, a natural step is to structure and design the coupling between the model and the GUI by following the MVVM (Model-View-ViewModel) architectural pattern.

The purpose of this pattern is: “to separate the state and behavior from the appearance and structure of the UI”. The main principle of the MVVM pattern is that the system can be divided into 3 parts, structured as packages:

The model package, which contains every part of the system that is not part of the GUI. For the Beaver hotel booking system, this includes the business logic and database. The model does not have access to either the view or viewModel but is updated by the viewModel.

The viewModel package serves as a mediator between the view and the model and is responsible for the logic behind the controls in the view. This is achieved by having instance variables for storing view controls by binding. For example, if the view has a text field for the user to input data into, then the viewModel would have a method processing this data and passing it to the model - but without knowing that the data came from specifically a text field. The viewModel has access to the model, but not to the view.

The view package, which contains information about the design, layout, and control of the GUI such as the FXML-files and the basic implementation of these e.g., declaring a textfield but does not contain any functionality regarding the logic of the system. The view has access to the viewModel, but not to the model.

In the Hotel Management System, adherence to the MVVM pattern is achieved by dividing the client side of the system into three main packages: model, viewModel and view. The viewModel package accesses the model package through a ModelManager-object that implements the Model-interface. This makes it possible for the ViewModel to make use of the methods in the Model.

The figure below shows a simplified version of the model, viewModel and view packages in the client, and describes how the MVVM-pattern is followed:

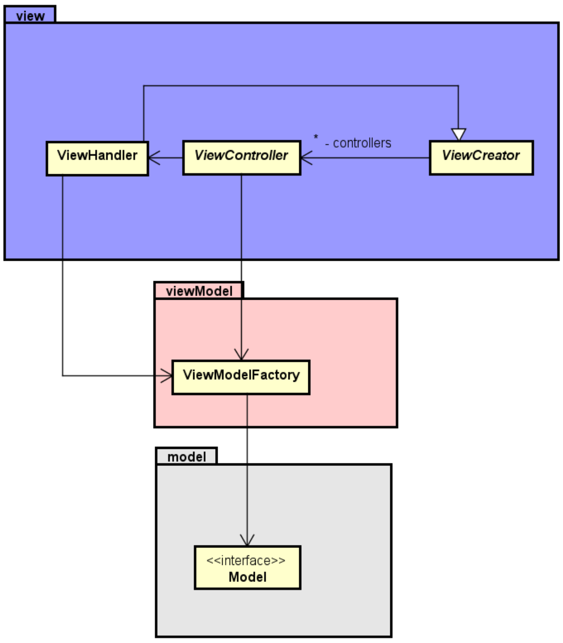


Figure 13 - MVVM

3.3 Design patterns

Apart from dividing the classes into different packages, it was decided to design the system around the SOLID and GRASP (General Responsibility Assignment Software Principles) sets of principles. Designing the system using these, will help tremendously in making sure that the code written during the implementation-phase will be easily maintained- and testable.
Furthermore, it was decided to follow specific Design Patterns to increase maintainability and integrity of the code.

3.3.1 The Factory Pattern

In the Hotel management system, the Factory pattern is used for creation of the ViewControllers for the GUI. This is achieved by having an abstract ViewController class, in which all of the methods that are the same for all ViewControllers are defined, while the methods that differ between the different controllers are declared as abstract methods.
A child class is present for each viewcontroller present in the system, each of these extending the ViewController class, and implementing the abstract methods.
The ViewHandler extends the abstract class ViewCreator following the same principle as above. The ViewCreator is implemented as a Multiton, meaning that each time the getViewController-method is called, a single instance of it is returned.
The reasoning behind this method of structuring is to reduce duplicate code, and by doing this make the system easier to maintain and expand in the future.

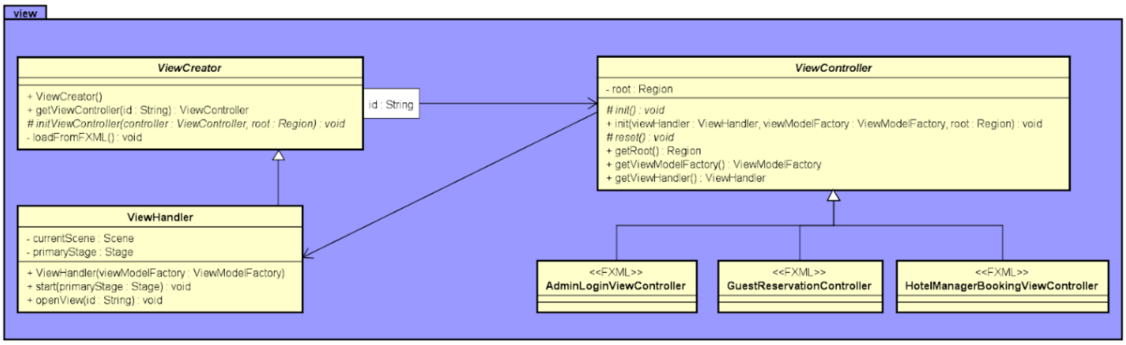

view
ViewCreator
+ ViewCreator()
+ getViewController(id : String) : ViewController
# initViewController(controller : ViewController, root : Region) : void
- loadFromFXML() : void
id : String
ViewController
- root : Region
# init() : void
+ init(viewHandler : ViewHandler, viewModelFactory : ViewModelFactory, root : Region) : void
# reset() : void
+ getRoot() : Region
+ getViewModelFactory() : ViewModelFactory
+ getViewHandler() : ViewHandler
ViewHandler
- currentScene : Scene
- primaryStage : Stage
+ ViewHandler(viewModelFactory : ViewModelFactory)
+ start(primaryStage : Stage) : void
+ openView(id : String) : void
<<FXML>>
AdminLoginViewController
<<FXML>>
GuestReservationController
<<FXML>>
HotelManagerBookingViewController

Figure 14 - Factory method

3.4 Class Diagram

To make implementation as structured as possible, a class diagram has been created showing all classes and methods that need implementation for the system to function. Having designed the system as described above, a class diagram can now be created for use in the implementation phase.

The class diagram builds upon the domain model, but also shows the methods designed for each class.

As shown in both class diagrams shown below, the model-packages contain a Model-interface, to be implemented by the ModelManager-class.

On the server-side, the ModelManager has the instance-variables RoomBookingList and RoomList, each used for holding RoomBooking- and Room-objects, respectively.

A Room-object consists of a roomId, which is the value used to identify the room throughout the system, a variable, numberOfBeds, used to specify the capacity of the room, a price-variable which specifies the daily renting price for that room, and finally a RoomType, specifying the type of the room.

The RoomType variable is to be implemented as an Enum-class, containing the five different kinds of rooms specified during the analysis-phase.

A RoomBooking object consists of a start date and an end date, which is going to be implemented as objects of the LocalDate-class in Java, a Room object, a Guest object and a bookingId used to identify the booking throughout the system.

The four different Transfer-objects in the mediator package are used for storing values sent between the client and the server and are converted to Json-strings by the HotelClientHandler and HotelClient classes.

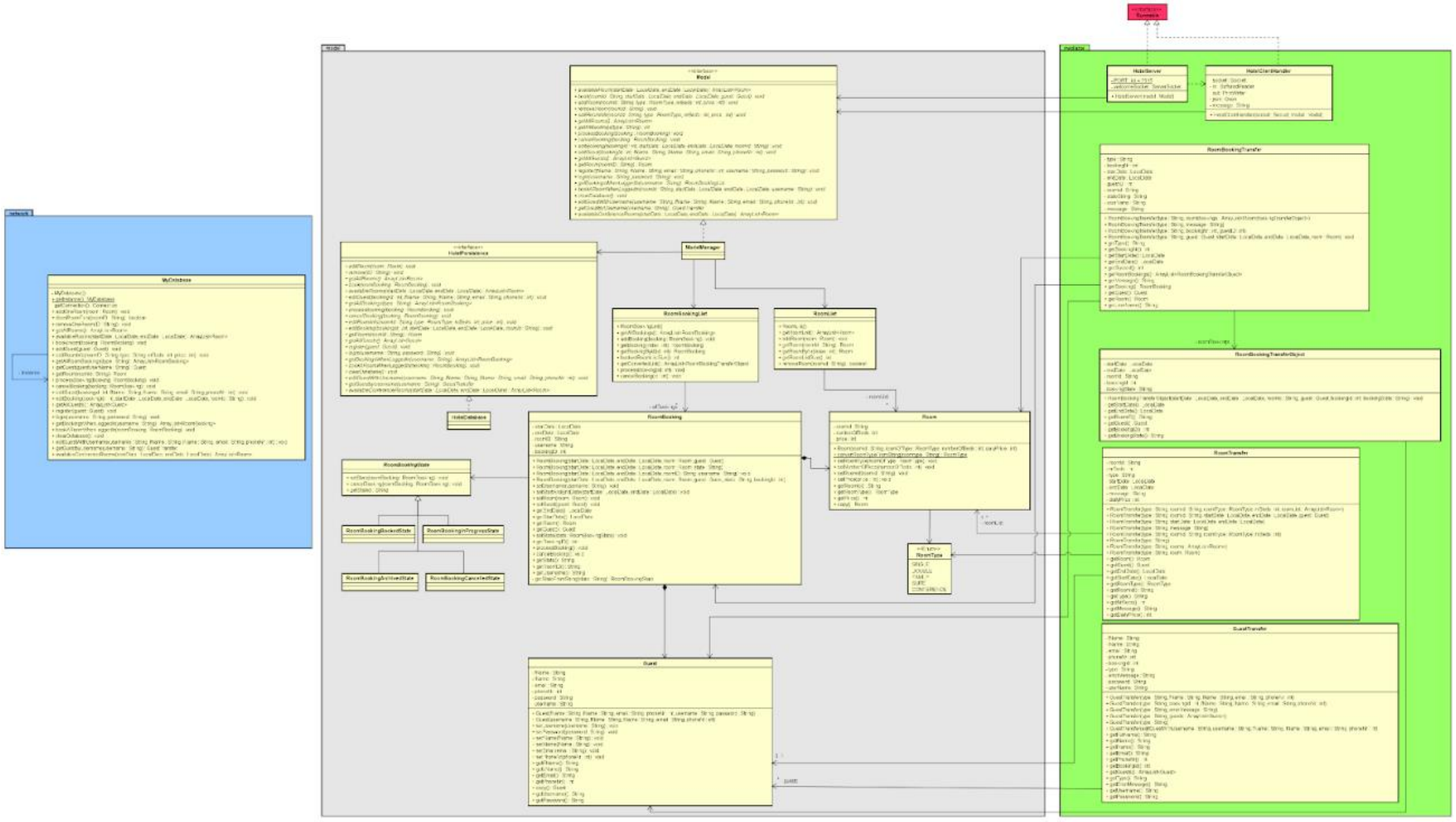


Figure 15 - Class Diagram, serve

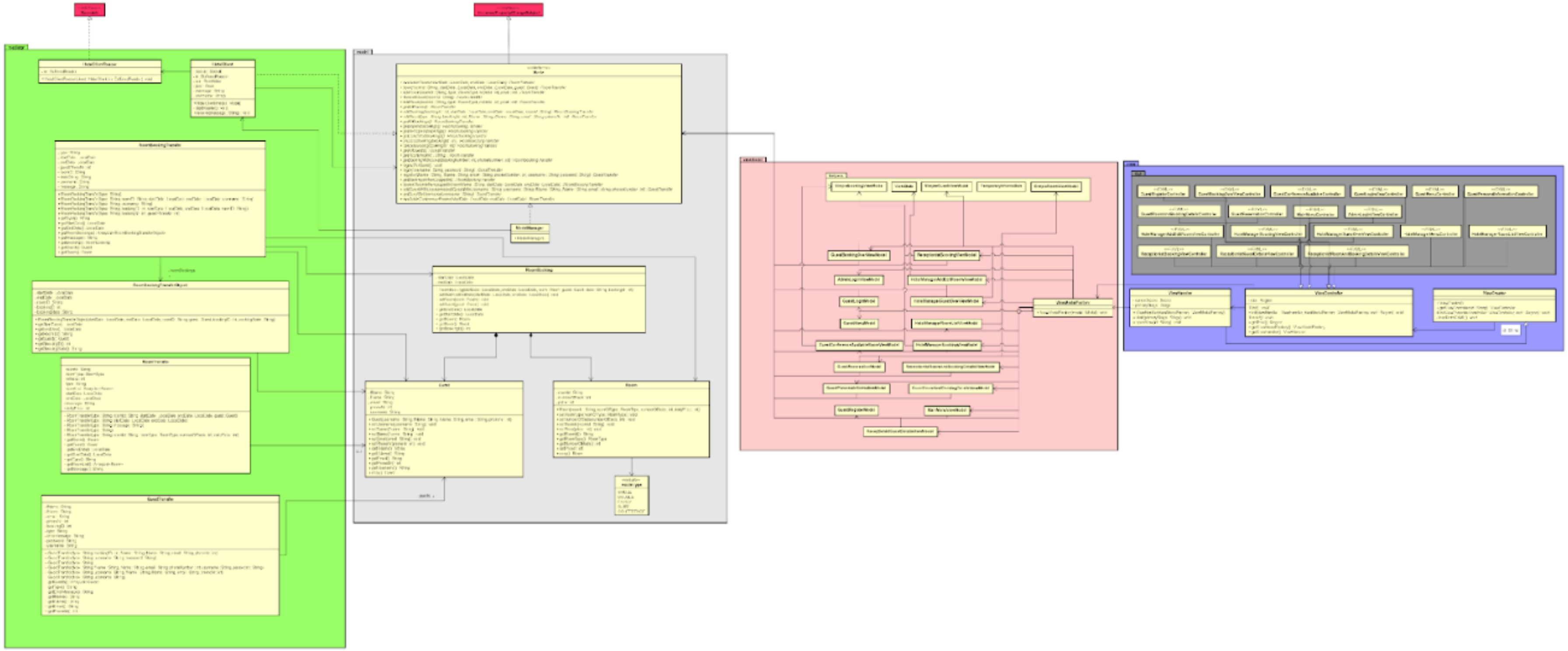


Figure 16 - Class Diagram, client

3.5 Sequence Diagram

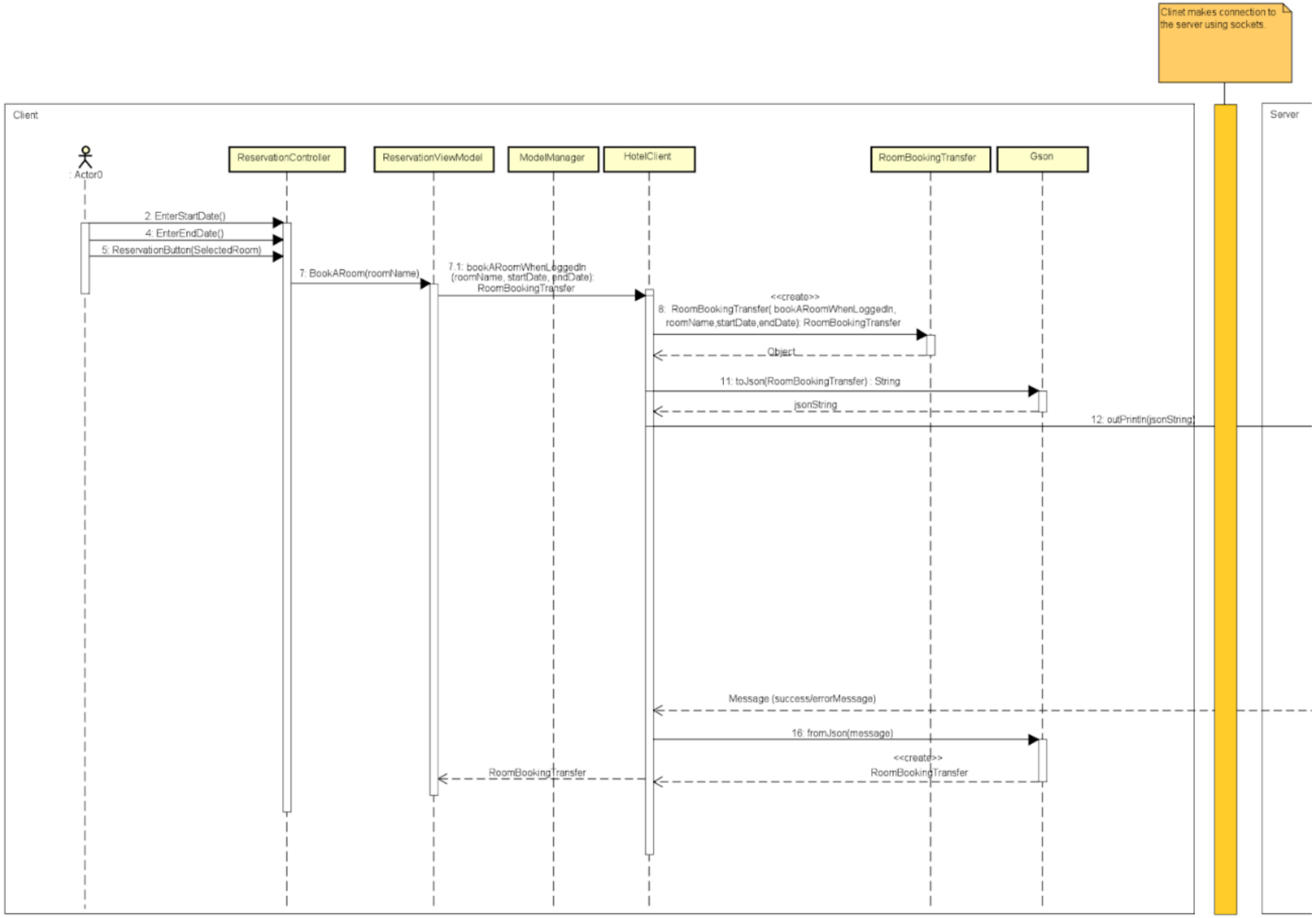
As the operation of booking a room is critical to the system, it was chosen to further describe it by creating a Sequence Diagram, showing the path through the system. This diagram elaborates on the System Sequence Diagram, as it shows what methods will be called throughout both the client and the server when the room-booking operation is run.

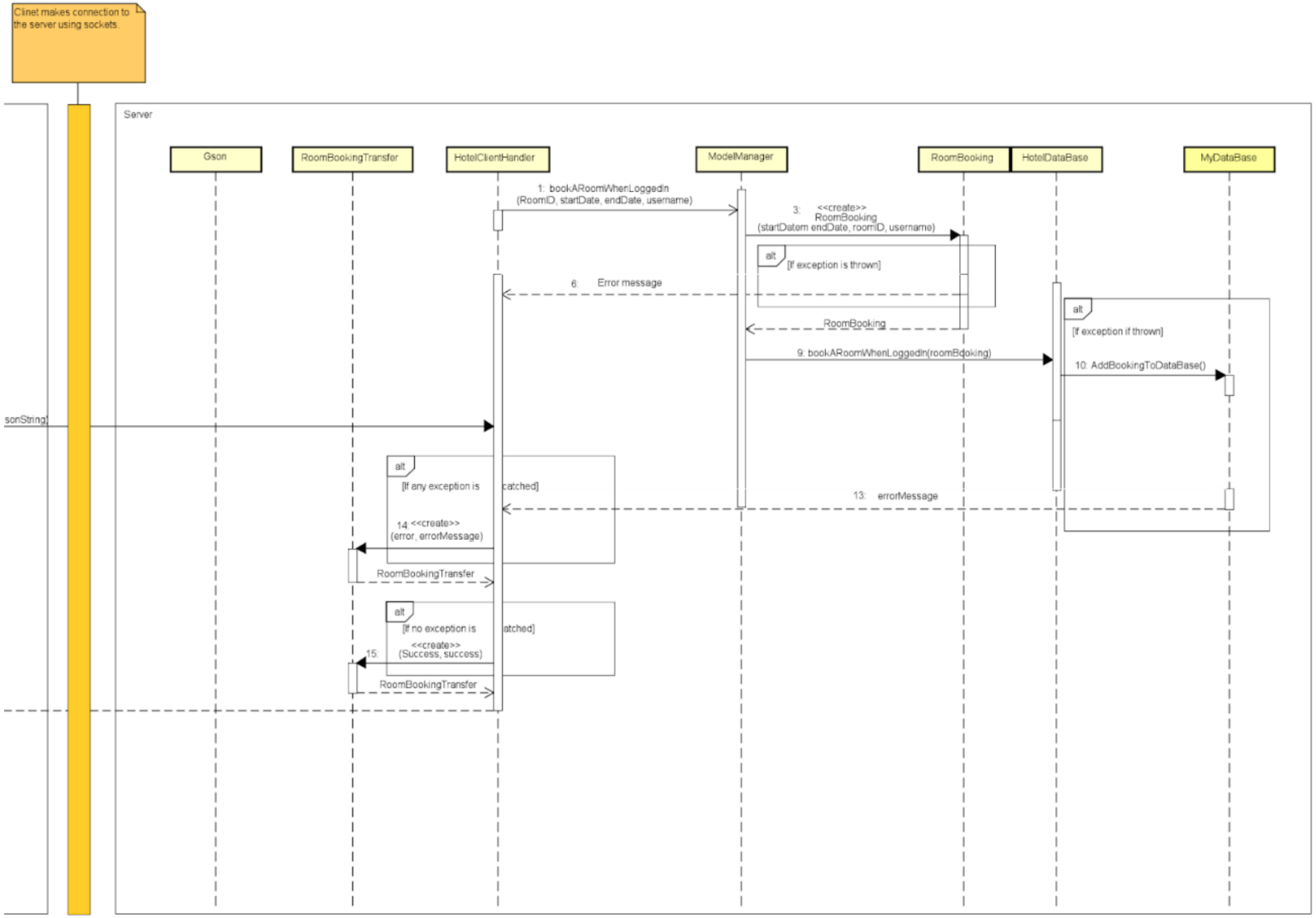
On the client-side, after the Guest activates the ReservationButton, the method is passed on to the viewmodel, ModelManager and to the HotelClient, in which a RoomBookingTransfer-object is created, converted to a Json string and sent to the HotelClientHandler on the server.

The server then processes the received string, converting it first to a RoomBookingTransfer-object, and then creates a RoomBooking from this object, hereby checking if the booking-data is valid in regard to the checks made in the RoomBooking constructor. The booking-object is then passed to the HotelDatabase-class which tries to add it to the database.

If any exceptions are caught during the two steps previously described, a RoomBookingTransfer-object containing the exception error message is returned to the client. If, on the other hand, the booking succeeds, a RoomBookingTransfer-object containing a success-message is returned.

Depending on whether the server-reply is an error/success message, the client will either show the Guest the exception message or confirm that the booking was made successfully.


Clinet makes connection to the server using sockets.
Client
Server
: Actor0
ReservationController
ReservationViewModel
ModelManager
HotelClient
RoomBookingTransfer
Gson
2: EnterStartDate()
4: EnterEndDate()
5: ReservationButton(SelectedRoom)
7: BookARoom(roomName)
7.1: bookARoomWhenLoggedIn (roomName, startDate, endDate): RoomBookingTransfer
<<create>>
8: RoomBookingTransfer(bookARoomWhenLoggedIn, roomName,startDate,endDate): RoomBookingTransfer
Object
11: toJson(RoomBookingTransfer) : String
jsonString
12: outPrintln(jsonString)
Message (success/errorMessage)
16: fromJson(message)
<<create>>
RoomBookingTransfer
RoomBookingTransfer


Clinet makes connection to the server using sockets.
Server
Gson
RoomBookingTransfer
HotelClientHandler
ModelManager
RoomBooking
HotelDataBase
MyDataBase
1: bookARoomWhenLoggedIn (RoomID, startDate, endDate, username)
3: <<create>> RoomBooking (startDatem endDate, roomID, username)
alt
[If exception is thrown]
6: Error message
RoomBooking
9: bookARoomWhenLoggedIn(roomBooking)
alt
[If exception if thrown]
10: AddBookingToDataBase()
sonString)
alt
[If any exception is catched]
13: errorMessage
14: <<create>> (error, errorMessage)
RoomBookingTransfer
alt
[If no exception is atched]
15: <<create>> (Success, success)
RoomBookingTransfer

3.6 Persistence

To ensure persistence of data in the system, it has been decided to implement a database, HotelDatabase, to be used for storage of data.

The first thing to do when designing a database is to create an ER-diagram (Entity Relation Diagram):

The entity diagram shows the basic skeleton of the database to be created, and includes the basic relations and values specified by the Use Cases obtained from the analysis-phase.

It is worth noting, that besides the three normal entities Login, Guest and Room, the diagram contains one Relational Entity: RoomBooking, meaning that it is an entity that exists as a relation between Guest and Room.

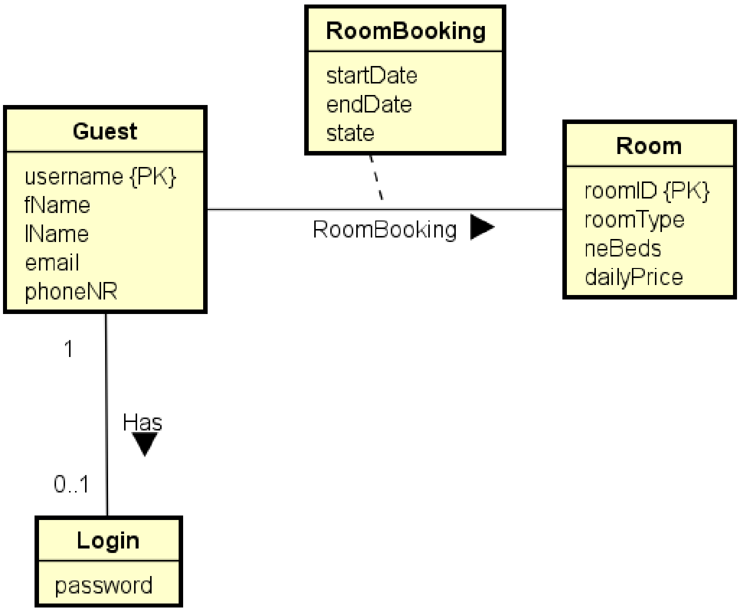


Figure 17 - ER Diagram

After creating the ER-diagram, this was mapped to a relational schema, as shown below:

Guest (username, fName, lName, email, phoneNR) PK: username AK: email AK: phoneNR	Room (roomID, roomType, nrBeds, dailyPrice) PK: roomID
RoomBooking(username,roomID,bookingID, startDate, endDate, state) PK: bookingID FK: username ref Guest(username) FK: roomID red Room(roomID)	Login (password, username) PK: username FK: username ref Guest(username)

The most important changes to note is that primary and alternate keys have been mapped for all entities, meaning that a single row in the database is going to be able to be uniquely identified using DQL.

From the relational schema, a Global Relation Diagram was created, and is to be used when implementing the database.

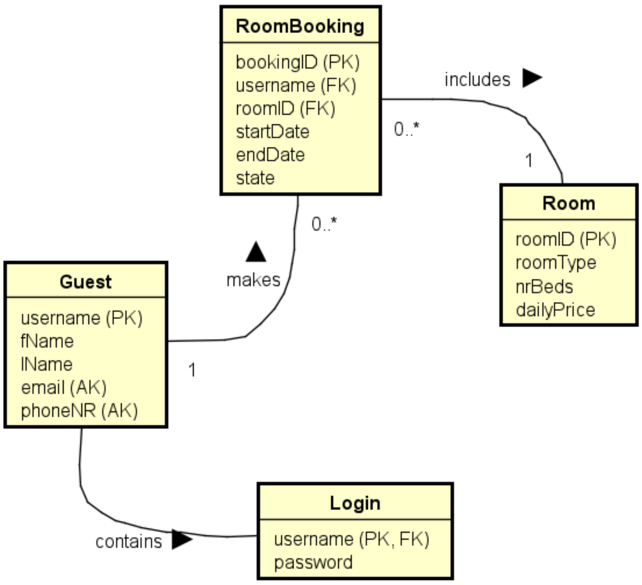


Figure 18 - GR Diagram

3.7 Adapter Pattern

After designing the database, the next step is to design the coupling between the database and the system, using the MyDatabase class.

The use of the adapter pattern is a solution to the problem of having the ModelManager communicating with the MyDatabase-class, as the two are not directly compatible. Following the Adapter-pattern, an interface (HotelPersistence) was created and is to be implemented by the HotelDatabase-class to adapt the methods used in the ModelManager for use with MyDatabase.

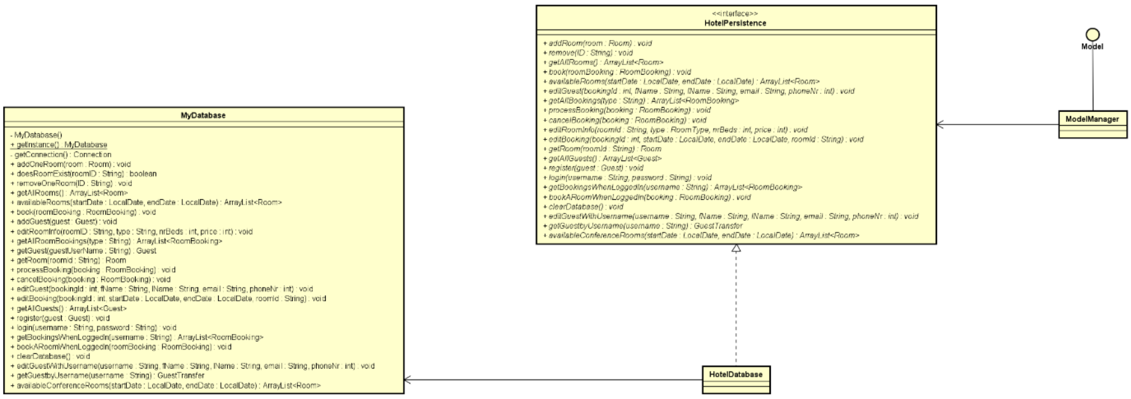

MyDatabase
- MyDatabase()
+ getInstance() : MyDatabase
- getConnection() : Connection
+ addOneRoom(room : Room) : void
+ doesRoomExist(roomID : String) : boolean
+ removeOneRoom(ID : String) : void
+ getAllRooms() : ArrayList<Room>
+ availableRooms(startDate : LocalDate, endDate : LocalDate) : ArrayList<Room>
+ book(roomBooking : RoomBooking) : void
+ addGuest(guest : Guest) : void
+ editRoomInfo(roomID : String, type : String, nrBeds : int, price : int) : void
+ getAllRoomBookings(type : String) : ArrayList<RoomBooking>
+ getGuest(guestUserName : String) : Guest
+ getRoom(roomId : String) : Room
+ processBooking(booking : RoomBooking) : void
+ cancelBooking(booking : RoomBooking) : void
+ editGuest(bookingId : int, fName : String, lName : String, email : String, phoneNr : int) : void
+ editBooking(bookingId : int, startDate : LocalDate, endDate : LocalDate, roomId : String) : void
+ getAllGuests() : ArrayList<Guest>
+ register(guest : Guest) : void
+ login(username : String, password : String) : void
+ getBookingsWhenLoggedIn(username : String) : ArrayList<RoomBooking>
+ bookARoomWhenLoggedIn(roomBooking : RoomBooking) : void
+ clearDatabase() : void
+ editGuestWithUsername(username : String, fName : String, lName : String, email : String, phoneNr : int) : void
+ getGuestbyUsername(username : String) : GuestTransfer
+ availableConferenceRooms(startDate : LocalDate, endDate : LocalDate) : ArrayList<Room>
<<interface>>
HotelPersistence
+ addRoom(room : Room) : void
+ remove(ID : String) : void
+ getAllRooms() : ArrayList<Room>
+ book(roomBooking : RoomBooking) : void
+ availableRooms(startDate : LocalDate, endDate : LocalDate) : ArrayList<Room>
+ editGuest(bookingId : int, fName : String, lName : String, email : String, phoneNr : int) : void
+ getAllBookings(type : String) : ArrayList<RoomBooking>
+ processBooking(booking : RoomBooking) : void
+ cancelBooking(booking : RoomBooking) : void
+ editRoomInfo(roomId : String, type : RoomType, nrBeds : int, price : int) : void
+ editBooking(bookingId : int, startDate : LocalDate, endDate : LocalDate, roomId : String) : void
+ getRoom(roomId : String) : Room
+ getAllGuests() : ArrayList<Guest>
+ register(guest : Guest) : void
+ login(username : String, password : String) : void
+ getBookingsWhenLoggedIn(username : String) : ArrayList<RoomBooking>
+ bookARoomWhenLoggedIn(booking : RoomBooking) : void
+ clearDatabase() : void
+ editGuestWithUsername(username : String, fName : String, lName : String, email : String, phoneNr : int) : void
+ getGuestbyUsername(username : String) : GuestTransfer
+ availableConferenceRooms(startDate : LocalDate, endDate : LocalDate) : ArrayList<Room>
Model
ModelManager
HotelDatabase

Figure 19 - Adapter pattern

3.8 Singleton

To be sure that only one instance of the database is created, in order to make sure the PostgreSQL-driver is only registered once, the MyDatabase class is to be implemented as a Singleton.

This is done by designing the class to have a private constructor, an instance of itself and an abstract method called getInstance(). The getInstance() method will check if the instance has been initialized, and if it has, return the instance. If the instance has not been initialized, the constructor will be called and the newly initialized instance will be returned.

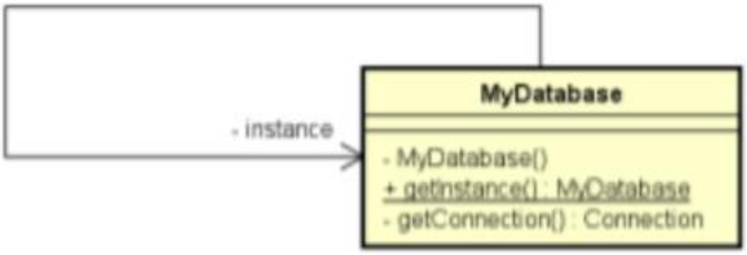


Figure 20 - Singleton

3.9 Deployment Diagram

To clarify how the program is to be deployed after the implementation is done, the following deployment diagram was created

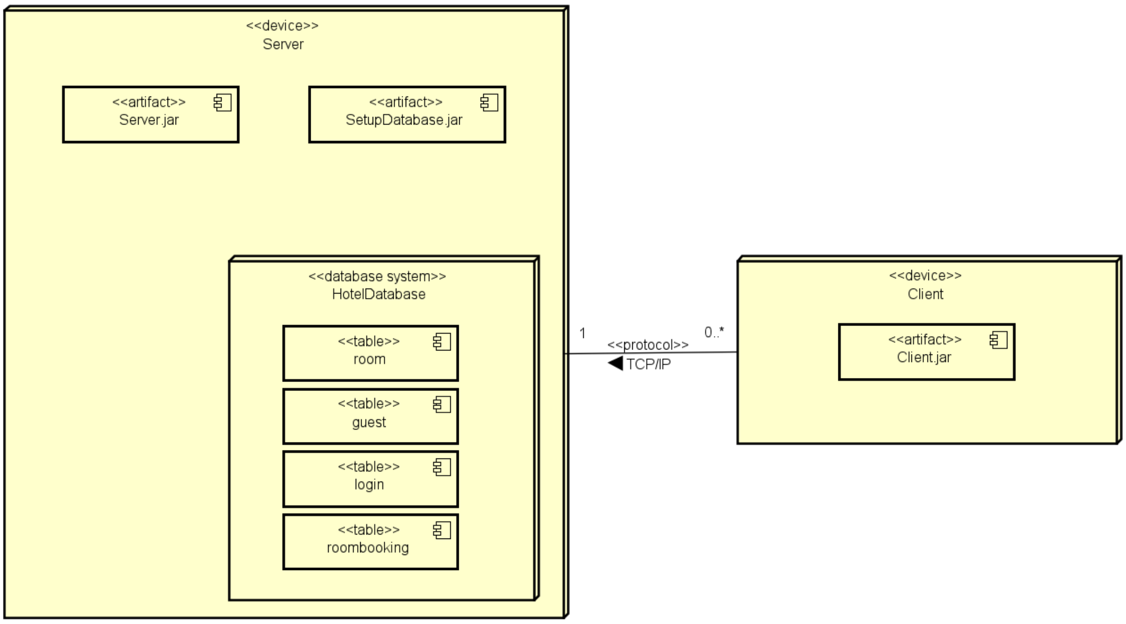


Figure 21 - Deplyment Diagram

The diagram shows that the system will consist of two types of nodes, which are the Server-device and the Client-device. It is shown that the system will need to have one server-device always running, and that multiple client-devices can connect to this using the TCP/IP protocol.

Both devices shown have artifacts, which are the .jar files that should be executed for the system to start. The client-device has one artifact, Client.jar, while the server has two: Firstly, SetupDatabase which is used to set up the HotelDatabase, and its schemas. Secondly, the Server.jar which is used to start and run the server after the database is initialized.

3.10 Graphical User Interface

As the Client-part of the system is intended to be accessible to anybody, no matter their experience and/or skills with computers, it was decided to implement a *Graphical User Interface* for this part.

When designing the GUI - wireframes were made in Balsamiq Wireframes to give the developers a rough sketch of how the GUI should look after implementation.

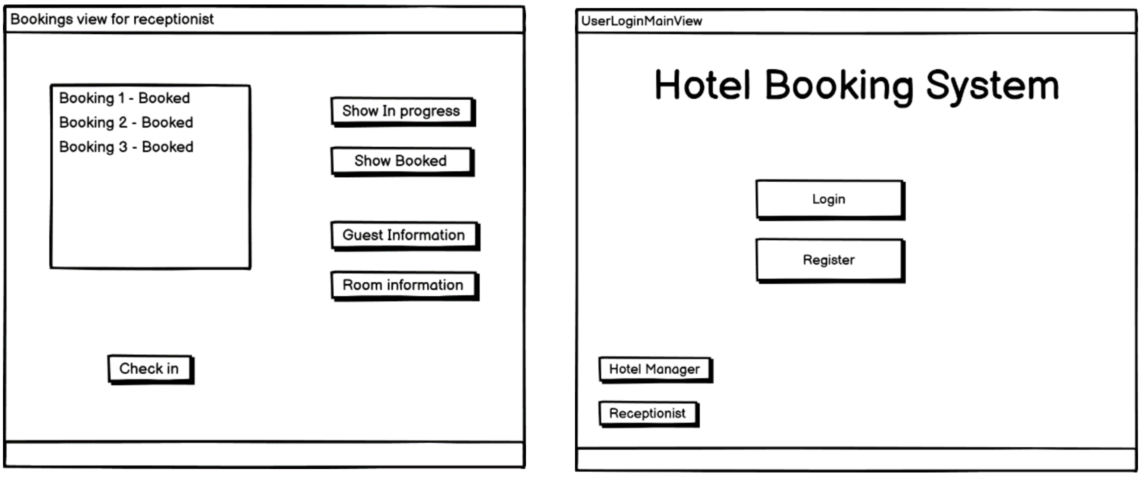
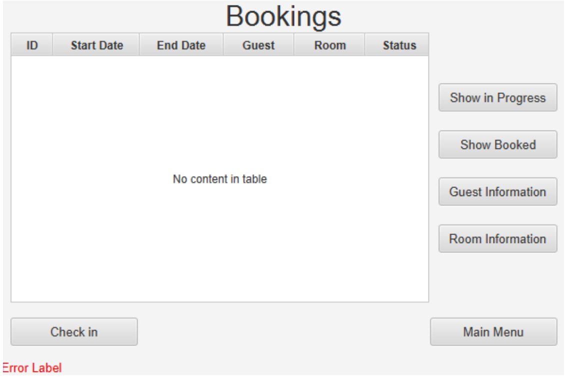


Figure 23 - Wireframe of ReceptionistBookingView

After the wireframes were created, the different GUI windows were designed in *SceneBuilder* with the goal of keeping a consistent look throughout the system.



ID	Start Date	End Date	Guest	Room	Status

No content in table

Figure 24 - SceneBuilder ReceptionistBookingView

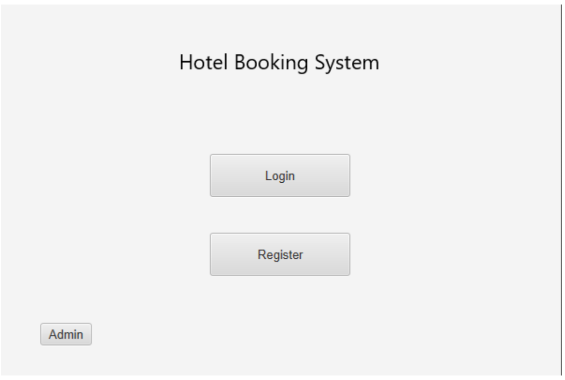


Figure 25 - SceneBuilder Login

As described earlier, the *GUI* functions by having a *ViewController* for each window, each of these having a *ViewModel* that handles the processing of data and delegates the communication with the server.

It was decided to add error-labels to all windows where the user can make a choice which could end with throwing an exception. This choice was made to avoid the system crashing, and to help the user by having the labels show information about the error.

Keeping in line with trying to reduce the possibility of user errors, it was decided that, when possible, when the user has to make a choice, a selection of choices should be given, instead of giving the user the ability to enter a possibly unknown value. An example of this can be seen in the *HotelManagerAddEditRoomView*, where the type of room is chosen using a *ComboBox*, to make sure that only allowed types can be chosen:

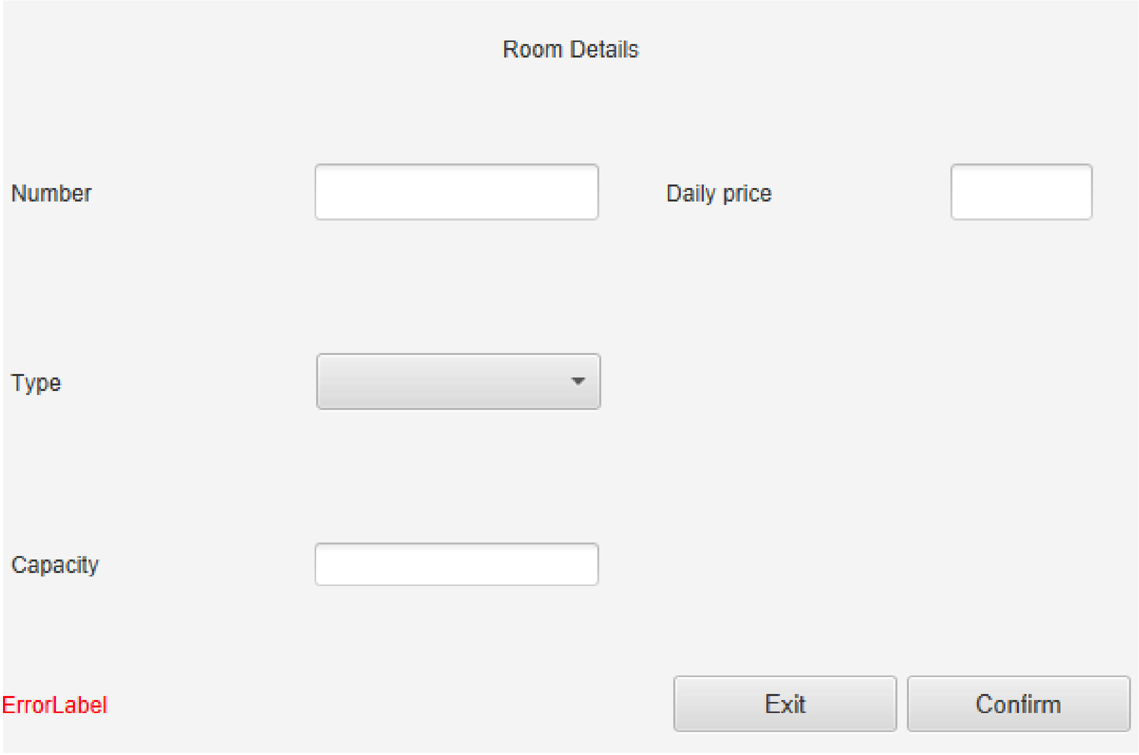


Figure 26 - SceneBuilder RoomDetails

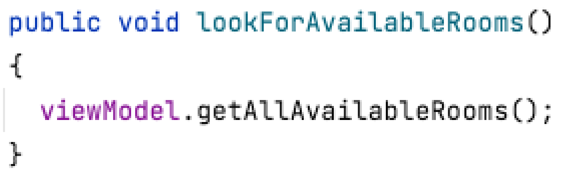
4 Implementation

4.1 GUI

Starting from the interface of the program - GUI whose job is to provide for a user all functionality in a simple window-based system using code logic that is written in the core of the system. To make this possible, we have used one of the libraries - JavaFX-11.0.2.

4.1.1 GUI controller

Each of the windows in a room booking system has its controller, which provides functionality by delegating work to a unique ViewModel created for each controller.



```
public void lookForAvailableRooms()
{
    viewModel.getAllAvailableRooms();
}
```

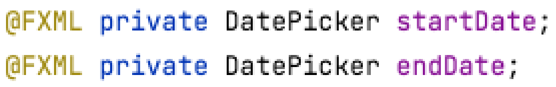
Figure 27 - Button in GUI

Figure 28 - Controller class

By pressing the “Book a room” button (Figure 27) assigned method in the controller class (Figure 28) delegates the work to its ViewModel.

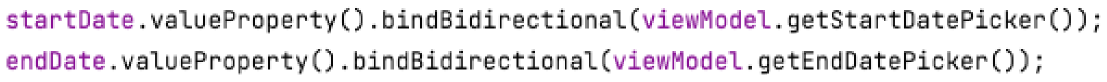
4.1.2 Binding

To ensure that ViewModel has all information needed to call methods in Model we make sure that each ViewController has a binding to its ViewModel class for all fields that the user can interact with. In (Figure 29), we are getting the value from DatePicker (Figure 29) and making binding by calling the get method of ObjectProperties (Figure 30). It makes sure that whenever the user selects a different value, ViewModel instance variables are updated with new information.



```
@FXML private DatePicker startDate;
@FXML private DatePicker endDate;
```

Figure 29 - Instance variables of DatePicker in GUI

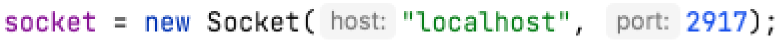


```
startDate.valueProperty().bindBidirectional(viewModel.getStartDatePicker());
endDate.valueProperty().bindBidirectional(viewModel.getEndDatePicker());
```

Figure 30 - Binding to ViewModel

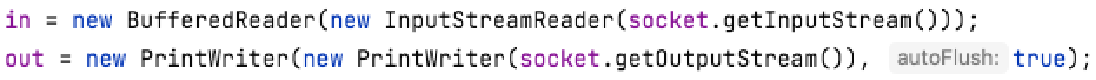
4.1.2.1 Connection to the server

Making a connection to the server we are sending a local port to identify ourselves (Figure 31) to the server, but in our case, we are using localhost since the server is on the same system as a program.
Another part is defining the port (Figure 31), which we will communicate through. As soon as the server accepts our request, we are creating a new Socket.



```
socket = new Socket( host: "localhost",  port: 2917);
```

Figure 31 - Socket

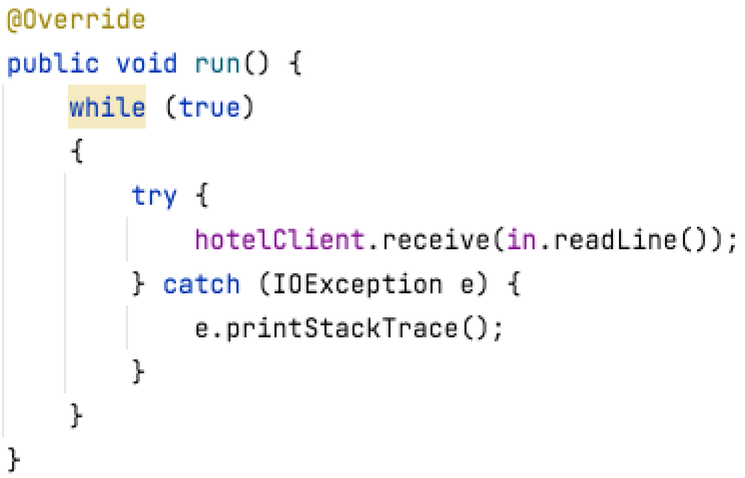


```
in = new BufferedReader(new InputStreamReader(socket.getInputStream()));
out = new PrintWriter(new PrintWriter(socket.getOutputStream(),  autoFlush: true);
```

Figure 32 - Making output and input streams with a connected socket

4.1.2.2 Receiving information from a server

To ensure that the system is always listening for any new messages coming from the server-side, we implemented the HotelClientReader class, which runs using a separate Thread (Figure 33). When HotelClientReader receives a message, it calls a method in the HotelClient class where the received message will be handled.



```java
@Override
public void run() {
    while (true)
    {
        try {
            hotelClient.receive(in.readLine());
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Figure 33 - HotelClientReader class

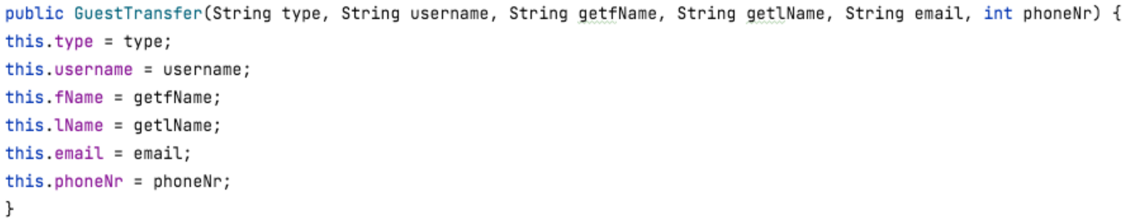
4.1.2.3 Object communication between server and client

For sending and receiving messages, we are creating either of the following objects:

- GuestTransfer
- RoomBookingTransfer
- RoomBookingTransferObject
- RoomTransfer

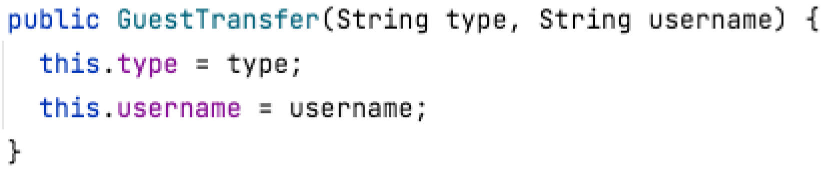
and sending them in JSON format.
(In order to use JSON, we had to use gson-2.8.5 library)
Each object is located on the client-side and server-side to make sure that both sides know how to create those objects and by using get methods, the server or client can get needed information from an object.

Each of the objects mentioned above has at least two constructors (Figure 34, Figure 35). Each constructor is for different scenarios with one thing in common, they all have a “type” it is used on the server-side to understand the task that has to be done, for example, edit, add, remove.



```
public GuestTransfer(String type, String username, String getfName, String getlName, String email, int phoneNr) {
this.type = type;
this.username = username;
this.fName = getfName;
this.lName = getlName;
this.email = email;
this.phoneNr = phoneNr;
}
```

Figure 34 - Constructor for GuestTransfer



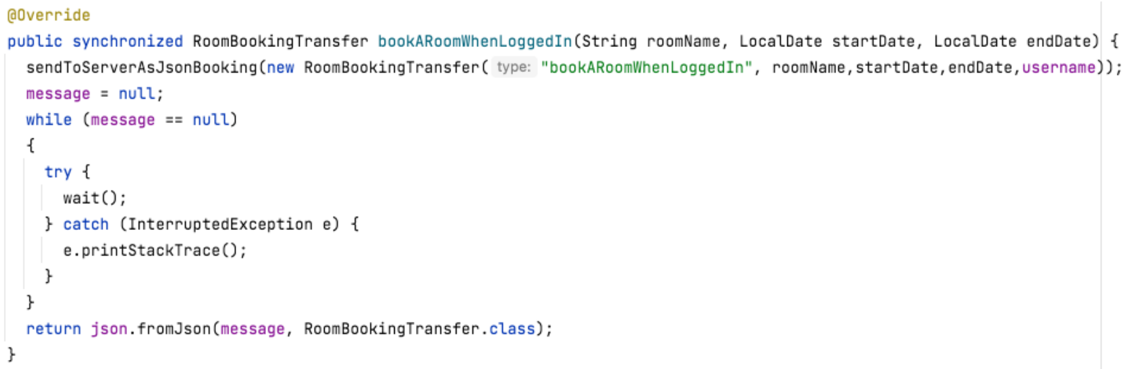
```
public GuestTransfer(String type, String username) {
  this.type = type;
  this.username = username;
}
```

Figure 35 - Constructor for GuestTransfer

Constructor (Figure 34) is used when the user wants to edit his personal details. In this case, the system is creating an object with the type: “editGuestWithUsername”, and username: clients username, and the rest of the arguments are information that has to be changed for this user.

Constructor (Figure 35) is used when the user wants to get detailed information about his account, in this case, the client only needs to send a username that is uniquely identified on a server-side additionally with the type of getGuestByUsername.

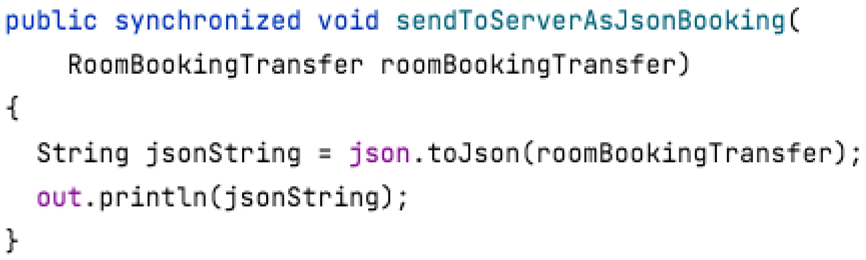
4.1.2.4 Sending information to the server



```
@Override
public synchronized RoomBookingTransfer bookARoomWhenLoggedIn(String roomName, LocalDate startDate, LocalDate endDate) {
  sendToServerAsJsonBooking(new RoomBookingTransfer( type: "bookARoomWhenLoggedIn", roomName,startDate,endDate,username));
  message = null;
  while (message == null)
  {
    try {
      wait();
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
  }
  return json.fromJson(message, RoomBookingTransfer.class);
}
```

Figure 36 - bookARoomWhenLoggedIn method

In this method, we are creating a new “Transfer” object with needed information for the server to receive, and with the created object, we are calling sendToServerAsJsonBooking method (Figure 37) which transfers the passed object into JSON and sends it to the server.



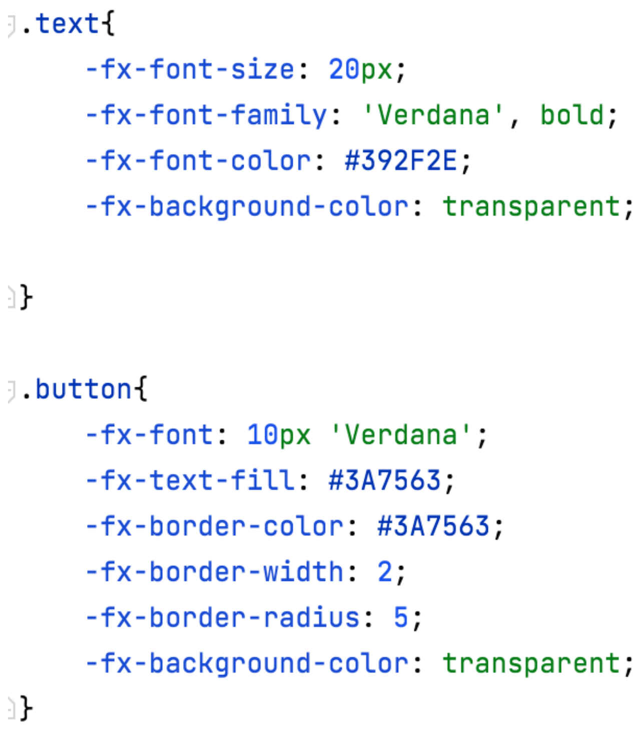
```
public synchronized void sendToServerAsJsonBooking(
    RoomBookingTransfer roomBookingTransfer)
{
  String jsonString = json.toJson(roomBookingTransfer);
  out.println(jsonString);
}
```

Figure 37 - sendToServerAsJsonBooking object

After that, we set the message to null (Figure 11) and put the current Thread to wait until someone notifies, and the message will not be null. As soon as we receive a message from the server, we set the message string to the message received and call notify(). It notifies the waiting thread, which will be the one waiting for an answer. Using wait() and notify(), we ensure that the method will receive the proper information.

4.1.3 GUI Design

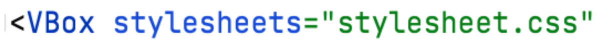
To make our program look more appealing, we added additional CSS elements to it, as seen in (Figure 38)



```
.text{
    -fx-font-size: 20px;
    -fx-font-family: 'Verdana', bold;
    -fx-font-color: #392F2E;
    -fx-background-color: transparent;

}

.button{
    -fx-font: 10px 'Verdana';
    -fx-text-fill: #3A7563;
    -fx-border-color: #3A7563;
    -fx-border-width: 2;
    -fx-border-radius: 5;
    -fx-background-color: transparent;
}
```

Figure 38 - CSS styling

Instead of making changes to each FXML file, we choose to make one CSS file and attach it to each FXML file, as shown in (Figure 39)



```
<VBox stylesheets="stylesheet.css"
```

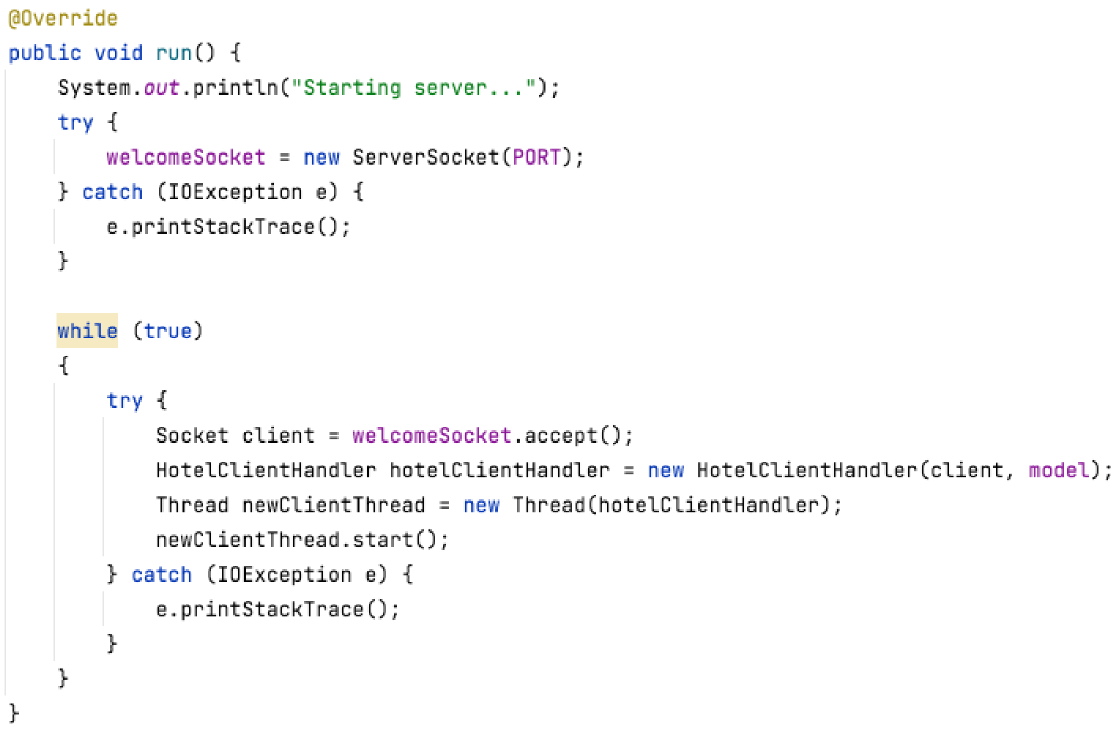
Figure 39 - attaching CSS file to FXML file.

4.1.4 Server handling clients

4.1.4.1 Client connection

On the server-side, we have two classes dedicated to clients, HotelServer and HotelClientHandler.

In HotelServer class we are creating ServerSocket with a selected port (Figure 40)



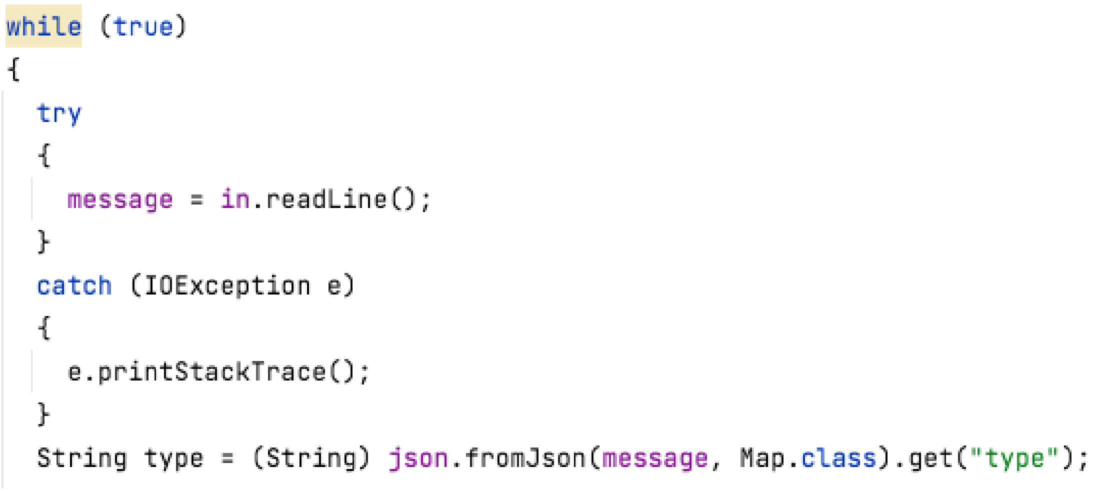
```
@Override
public void run() {
    System.out.println("Starting server...");
    try {
        welcomeSocket = new ServerSocket(PORT);
    } catch (IOException e) {
        e.printStackTrace();
    }

    while (true)
    {
        try {
            Socket client = welcomeSocket.accept();
            HotelClientHandler hotelClientHandler = new HotelClientHandler(client, model);
            Thread newClientThread = new Thread(hotelClientHandler);
            newClientThread.start();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Figure 40 - HotelServer class

After that, we always have a running loop that is waiting for new clients to join. As soon as a new client joins, we create a new Socket for that client. To have the possibility to handle more than one client we create a new object “HotelClientHandler” and we run it with a separate thread.

4.1.4.2 Receiving from a client and sending response back

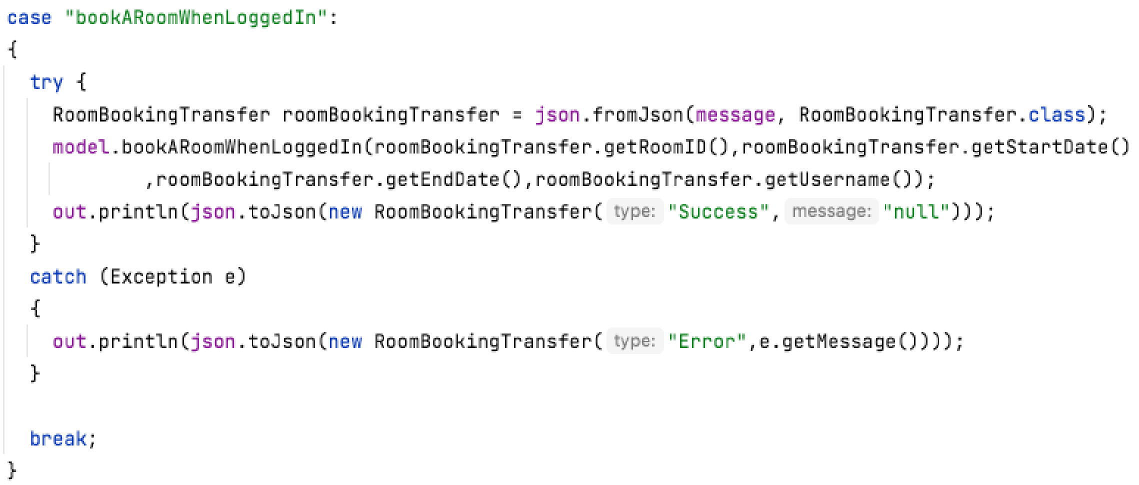
In the HotelClientHandler class, we have a constantly running loop that receives messages from the client side. After receiving the message, we transfer back from JSON format, and using Map.class, we get the value of instance 'type'. Further along, we use type value for a switch statement.



```
while (true)
{
  try
  {
    message = in.readLine();
  }
  catch (IOException e)
  {
    e.printStackTrace();
  }
  String type = (String) json.fromJson(message, Map.class).get("type");
```

Figure 41 - HotelClientHandler class

In the corresponding switch statement, we know what transfer object to expect, and we create it using JSON String received from a client. In this case, we call the model bookARoomWhenLoggedIn method and pass values from the newly created object. If, when calling the method, we do not catch any exception, we know that the room was successfully booked, and we send back a response to the client with the type “success” in another case, if we catch an exception, we send a response with the type “Error”.



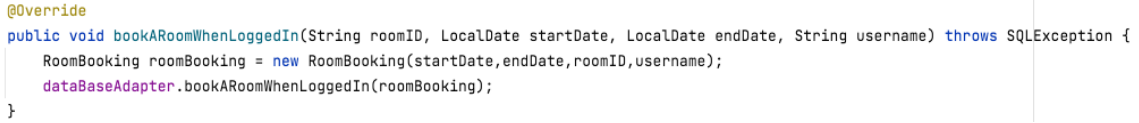
```
case "bookARoomWhenLoggedIn":
{
  try {
    RoomBookingTransfer roomBookingTransfer = json.fromJson(message, RoomBookingTransfer.class);
    model.bookARoomWhenLoggedIn(roomBookingTransfer.getRoomID(),roomBookingTransfer.getStartDate()
            ,roomBookingTransfer.getEndDate(),roomBookingTransfer.getUsername());
    out.println(json.toJson(new RoomBookingTransfer( type: "Success", message: "null")));
  }
  catch (Exception e)
  {
    out.println(json.toJson(new RoomBookingTransfer( type: "Error",e.getMessage())));
  }

  break;
}
```

Figure 42 - Switch statement

4.1.4.3 Server: checking legal values before calling database

Before making a connection to the database, firstly in ModelManager, we make sure that values are legal for example:

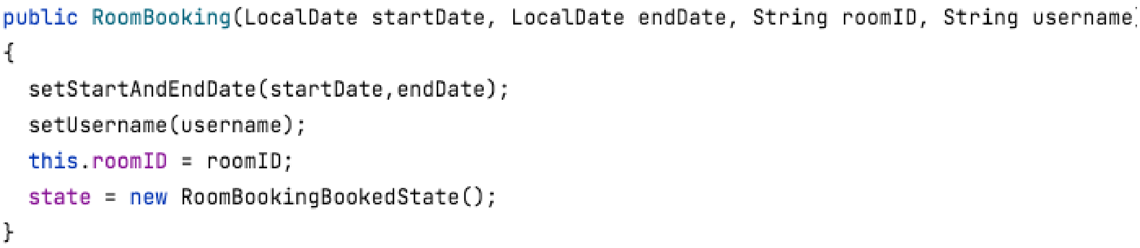
- Values cannot be null
- Email address contains @ and .
- Start and end dates are after today's date



```
@Override
public void bookARoomWhenLoggedIn(String roomID, LocalDate startDate, LocalDate endDate, String username) throws SQLException {
    RoomBooking roomBooking = new RoomBooking(startDate,endDate,roomID,username);
    dataBaseAdapter.bookARoomWhenLoggedIn(roomBooking);
}
```

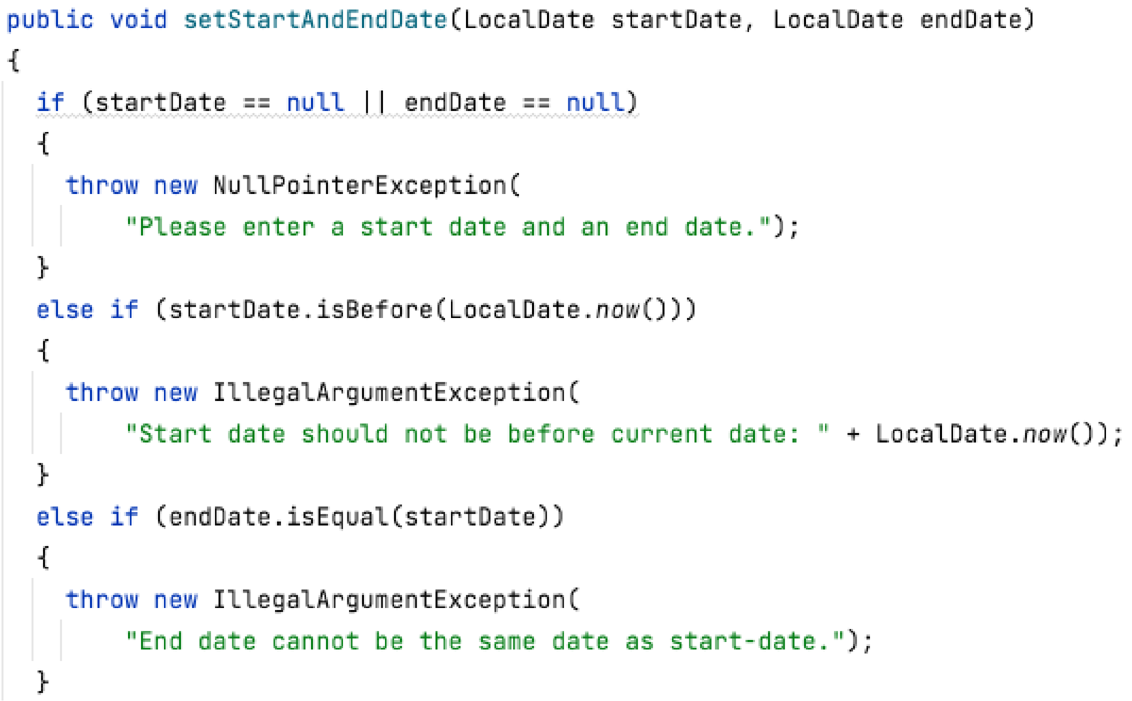
Figure 43 - ModelManager class

In ModelManager (Figure 43), we are creating an object of RoomBooking. Inside of the RoomBooking constructor (Figure 44), we call set methods for values that could be wrong. In set methods (Figure 45), we check if passed values are legal; if not, it throws an exception which throws back to HotelClientHandler and does not proceed to make a connection with the database.



```
public RoomBooking(LocalDate startDate, LocalDate endDate, String roomID, String username
{
  setStartAndEndDate(startDate,endDate);
  setUsername(username);
  this.roomID = roomID;
  state = new RoomBookingBookedState();
}
```

Figure 44 - RoomBooking class



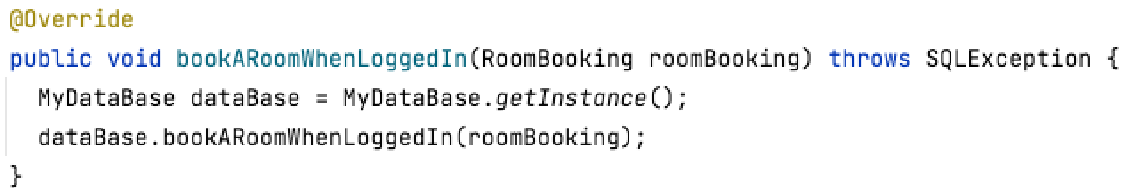
```
public void setStartAndEndDate(LocalDate startDate, LocalDate endDate)
{
  if (startDate == null || endDate == null)
  {
    throw new NullPointerException(
        "Please enter a start date and an end date.");
  }
  else if (startDate.isBefore(LocalDate.now()))
  {
    throw new IllegalArgumentException(
        "Start date should not be before current date: " + LocalDate.now());
  }
  else if (endDate.isEqual(startDate))
  {
    throw new IllegalArgumentException(
        "End date cannot be the same date as start-date.");
  }
```

Figure 45 - RoomBooking class

4.1.5 Server connection to database

4.1.5.1 Instance of MyDataBase class

MyDataBase class is a singleton for reasons covered in the design part. To use this class as a singleton, firstly, we call the method to MyDataBase class to get an instance of a class (Figure 46), then we can use this instance to call a method inside MyDataBase class.

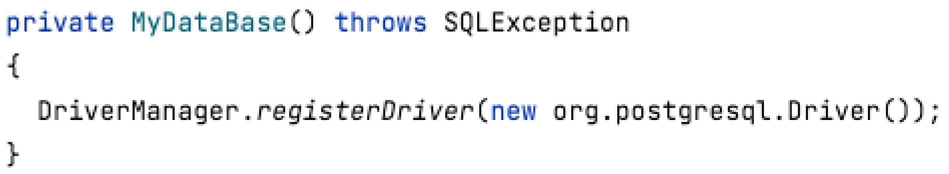


```
@Override
public void bookARoomWhenLoggedIn(RoomBooking roomBooking) throws SQLException {
  MyDataBase dataBase = MyDataBase.getInstance();
  dataBase.bookARoomWhenLoggedIn(roomBooking);
}
```

Figure 46 - HotelDataBase class

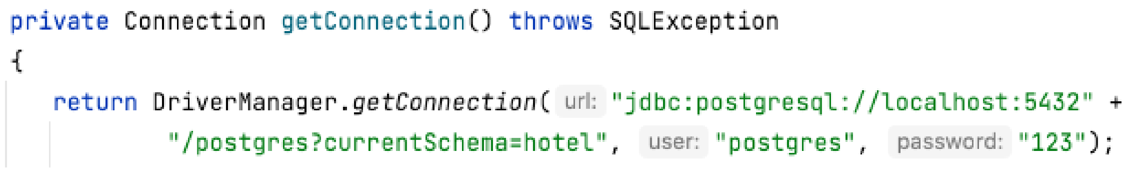
4.1.5.2 Accessing database

The first step to connecting to DataBase is registering a driver, it is done in the constructor because it only needs to be done once as shown in (Figure 47).



```
private MyDataBase() throws SQLException
{
  DriverManager.registerDriver(new org.postgresql.Driver());
}
```

Figure 47 - MyDatabase class

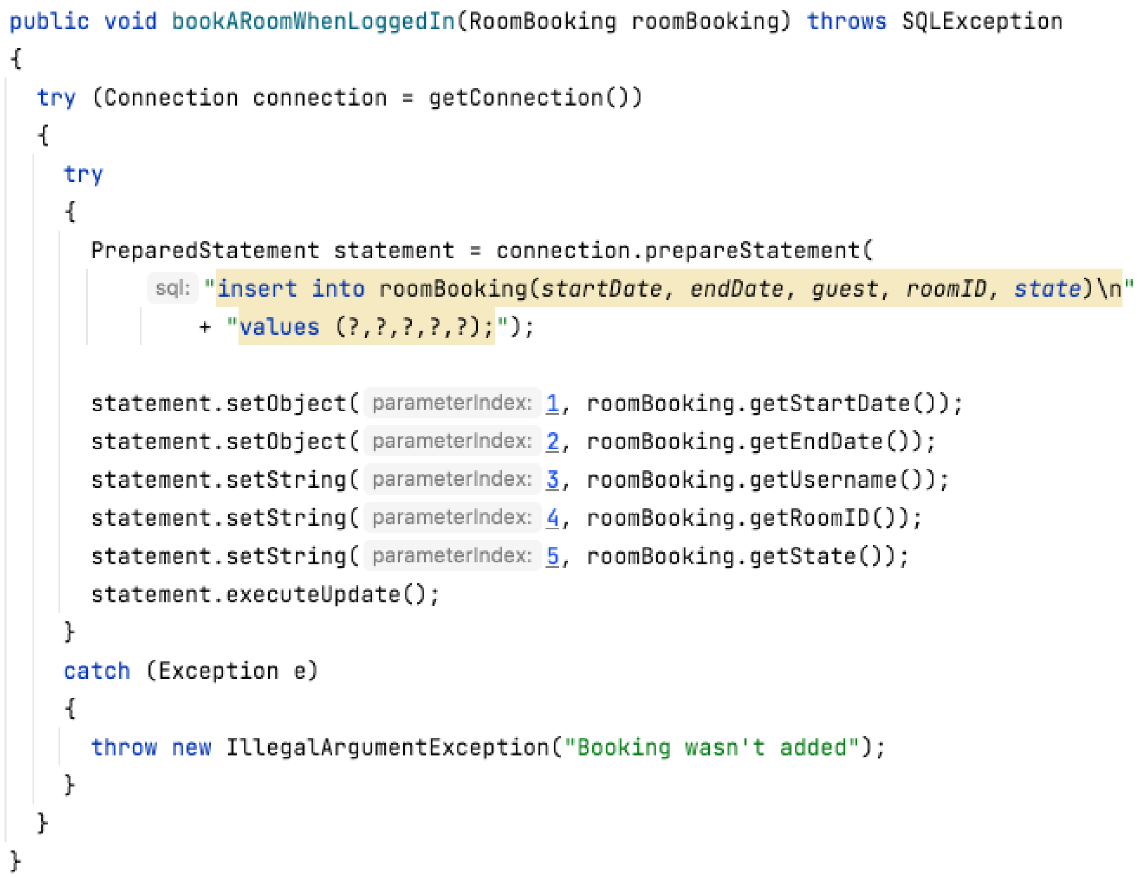


```
private Connection getConnection() throws SQLException
{
   return DriverManager.getConnection( url: "jdbc:postgresql://localhost:5432" +
           "/postgres?currentSchema=hotel",  user: "postgres",  password: "123");
```

Figure 48 - MyDatabase class

getConnection() method (Figure 48) is making a connection to the database through the URL and setting the current schema to ‘hotel’.

4.1.5.3 Inserting values into a database



```
public void bookARoomWhenLoggedIn(RoomBooking roomBooking) throws SQLException
{
  try (Connection connection = getConnection())
  {
    try
    {
      PreparedStatement statement = connection.prepareStatement(
          sql: "insert into roomBooking(startDate, endDate, guest, roomID, state)\n"
              + "values (?,?,?,?,?);");

      statement.setObject( parameterIndex: 1, roomBooking.getStartDate());
      statement.setObject( parameterIndex: 2, roomBooking.getEndDate());
      statement.setString( parameterIndex: 3, roomBooking.getUsername());
      statement.setString( parameterIndex: 4, roomBooking.getRoomID());
      statement.setString( parameterIndex: 5, roomBooking.getState());
      statement.executeUpdate();
    }
    catch (Exception e)
    {
      throw new IllegalArgumentException("Booking wasn't added");
    }
  }
}
```

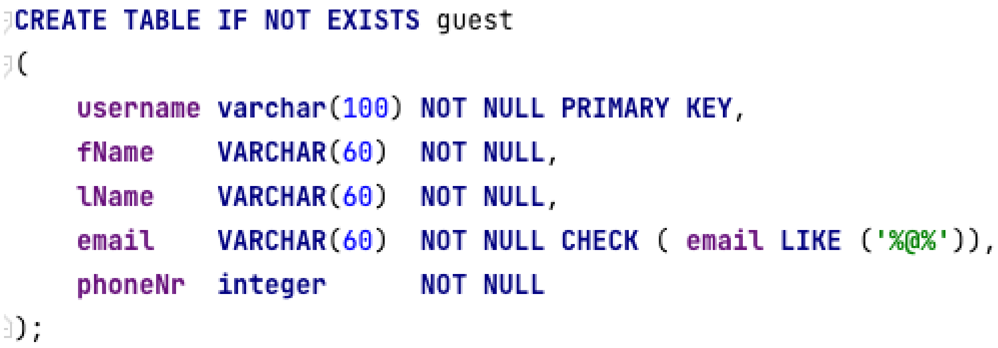
Figure 49 - bookARoomWhenLoggedIn method

Inside of a try (Figure 49), we connect to a database because after everything is executed in a try block, the connection to the database will be disconnected. Inside of the second try block, we create a new PreparedStatement and insert an SQL statement. Instead of values, we place question marks, and later, we set those question marks to values we want to replace with, it is mainly done like that to make sure that no one can by accident disturb SQL code.

4.2 Database implementation

4.2.1 Tables

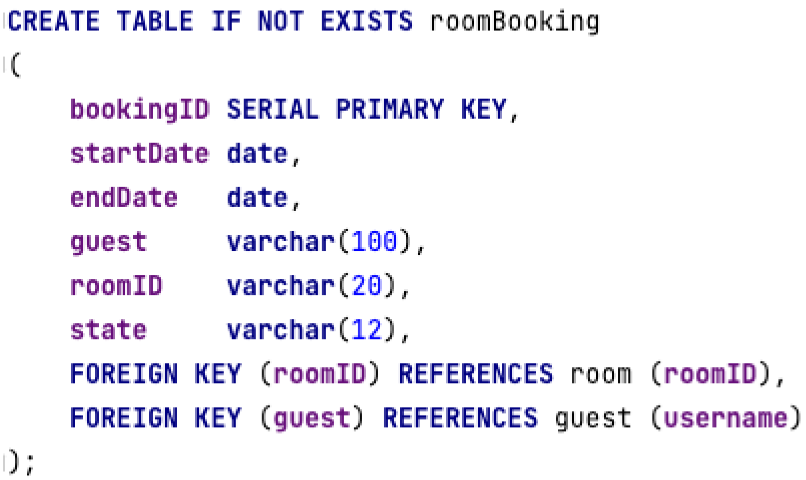
To implement the database, we choose to have four tables: guest, room, room-booking, and login for storing a password. For the guest table, as seen in (Figure 50), we decided that all values must be filled to add a new guest to the table by using the ‘NOT NULL’ keyword. We choose to use a username for the primary key since each user will have a unique login to the system using a username. For email, we decided to make sure that each email address has ‘@’ included since, without it, it would not be a valid email address.



```
CREATE TABLE IF NOT EXISTS guest
(
    username varchar(100) NOT NULL PRIMARY KEY,
    fName    VARCHAR(60)  NOT NULL,
    lName    VARCHAR(60)  NOT NULL,
    email    VARCHAR(60)  NOT NULL CHECK ( email LIKE ('%@%')),
    phoneNr  integer      NOT NULL
);
```

Figure 50 - Guest table

We decided to use two foreign keys referencing guest and room, as shown in (Figure 51) In this case, we make sure that each booking is associated with a guest who made that booking and a room where the booking will take place.

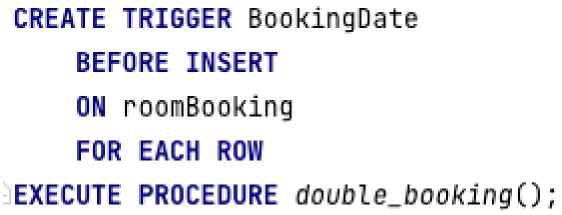


```
CREATE TABLE IF NOT EXISTS roomBooking
(
    bookingID SERIAL PRIMARY KEY,
    startDate date,
    endDate   date,
    guest     varchar(100),
    roomID    varchar(20),
    state     varchar(12),
    FOREIGN KEY (roomID) REFERENCES room (roomID),
    FOREIGN KEY (guest) REFERENCES guest (username)
);
```

Figure 51 - roomBooking table

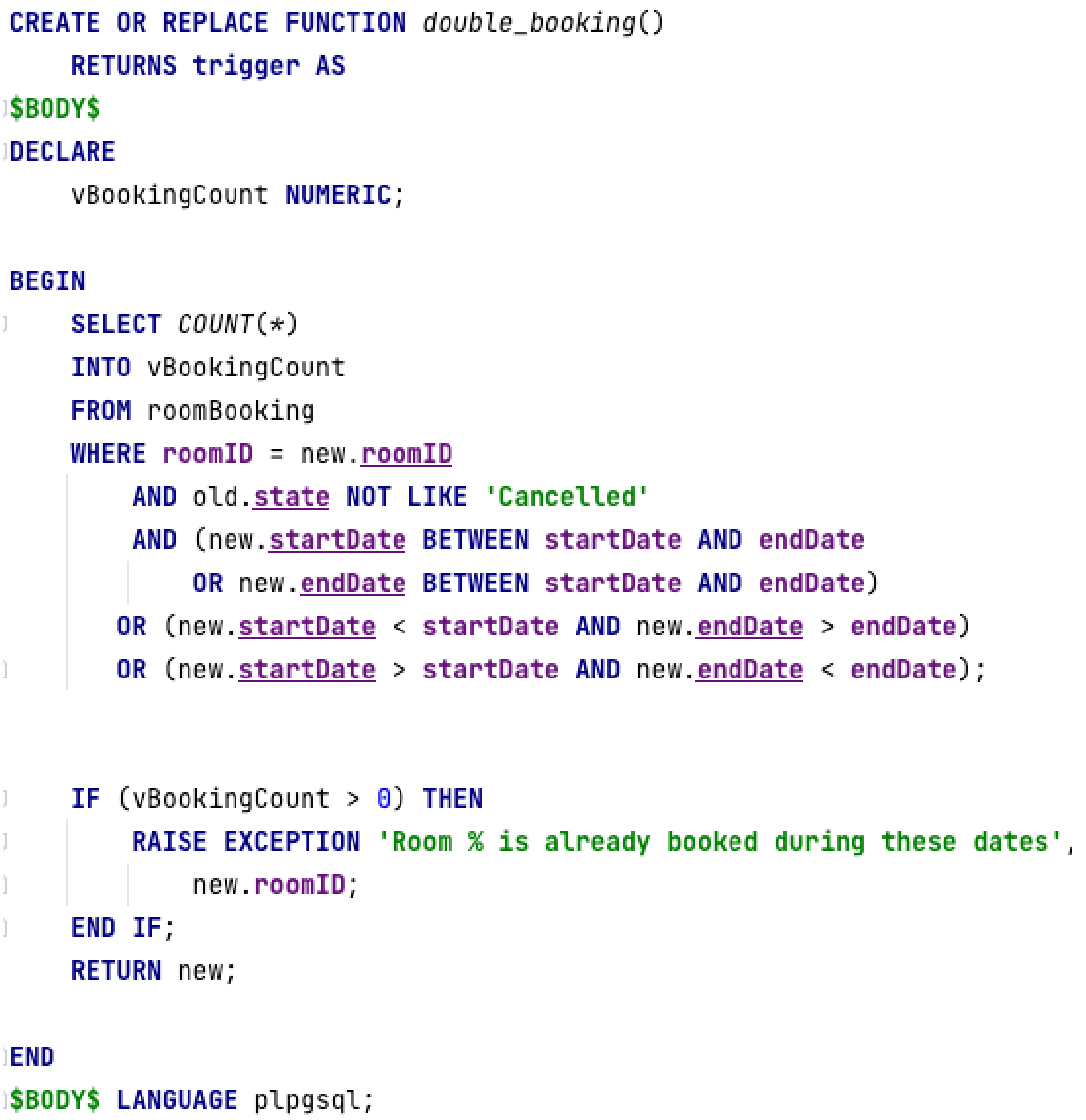
4.2.2 Triggers

To make sure that the room cannot have several bookings simultaneously, we had to implement a trigger and a function. As shown in (Figure 52) we are creating a trigger that will be triggered right before inserting a new booking. A trigger will run the ‘double_booking’ function for each row for the existing room-booking list, and if no exception is thrown, a booking will be successfully added to the list.



```
CREATE TRIGGER BookingDate
    BEFORE INSERT
    ON roomBooking
    FOR EACH ROW
EXECUTE PROCEDURE double_booking();
```

Figure 52 - Trigger



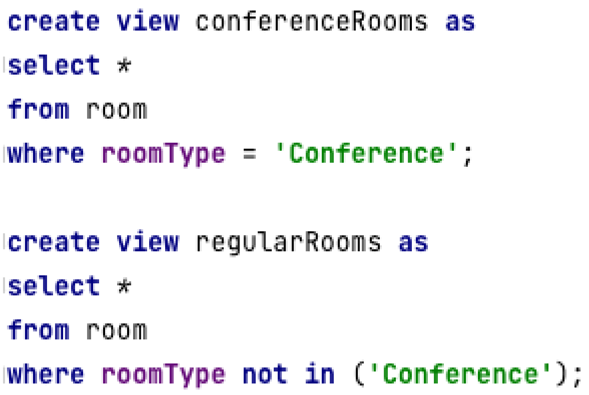
```
CREATE OR REPLACE FUNCTION double_booking()
    RETURNS trigger AS
$BODY$
DECLARE
    vBookingCount NUMERIC;

BEGIN
    SELECT COUNT(*)
    INTO vBookingCount
    FROM roomBooking
    WHERE roomID = new.roomID
        AND old.state NOT LIKE 'Cancelled'
        AND (new.startDate BETWEEN startDate AND endDate
            OR new.endDate BETWEEN startDate AND endDate)
       OR (new.startDate < startDate AND new.endDate > endDate)
       OR (new.startDate > startDate AND new.endDate < endDate);


    IF (vBookingCount > 0) THEN
        RAISE EXCEPTION 'Room % is already booked during these dates',
            new.roomID;
    END IF;
    RETURN new;

END
$BODY$ LANGUAGE plpgsql;
```

Figure 53 - Function

At the beginning of a function, we declare vBookingCount, which will be responsible for storing several overlapping bookings. After that, a function will compare a booking we want to insert and all bookings that are already in a system. For all bookings with the same room, we check if any of the dates interfere with new booking dates, and if it does, we increment a vBookingCouny by one. After looping through all matching bookings, we check if any bookings were found if at least one booking was found, this booking isn’t possible because of overlapping booking dates for the same room.

4.2.3 Views

Using views, we can call specific tables with already filtered rows. As shown in (Figure 54), we have two separate views of the room table. In the first view, we only show rooms that are conference rooms. For the second one, we show all rooms that are not conference rooms. Views help us easily distinguish room types when displaying only conference rooms or only standard rooms.



```
create view conferenceRooms as
select *
from room
where roomType = 'Conference';

create view regularRooms as
select *
from room
where roomType not in ('Conference');
```

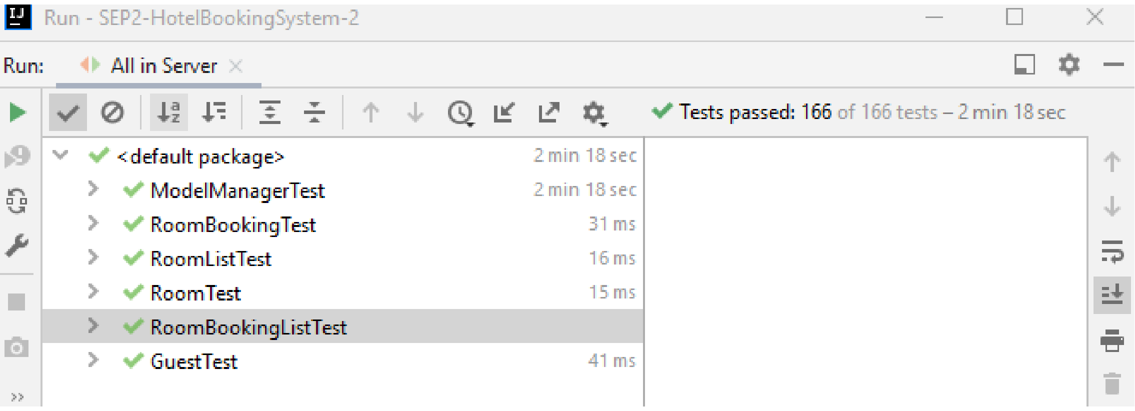
Figure 54 - Views

5 Test

To ensure functionality and quality of the Beaver Hotel Booking System, a series of tests were made. This was also done to provide the customer with proof of the functionality of the system.
To get the best possible coverage of tests, both Black Box and White Box testing were done.

5.1 White Box Testing:

White box testing deals with testing the actual methods implemented in the system, to ensure that they function as intended. JUnit was utilized to perform white box testing on the following classes in the *Model* package on the server: *Guest, RoomBooking*, *RoomBookingList*, *RoomList* and *ModelManager.*
The first four classes were chosen because they form the backbone of the business logic of the program, and the *ModelManager* was chosen because it is the class that links the Server-functionality together with both the database and the client.
A total of 166 unit tests were made, and all of them passed.
For each method tested, the *ZOMB+E* testing structure. This was chosen to make sure that each method was tested for the following cases where applicable: *Zero, One, Many, Boundary, Exceptions*.

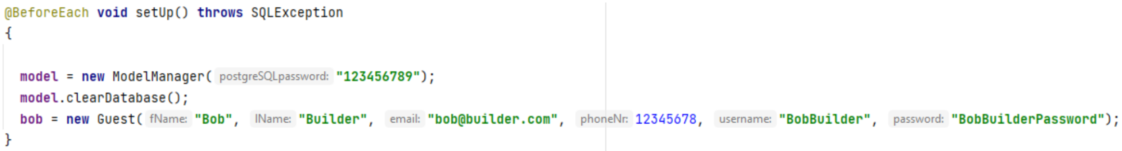


✓ Tests passed: 166 of 166 tests – 2 min 18 sec

✓ <default package>	2 min 18 sec
✓ ModelManagerTest	2 min 18 sec
✓ RoomBookingTest	31 ms
✓ RoomListTest	16 ms
✓ RoomTest	15 ms
✓ RoomBookingListTest	
✓ GuestTest	41 ms

Figure 55 - jUnit Results

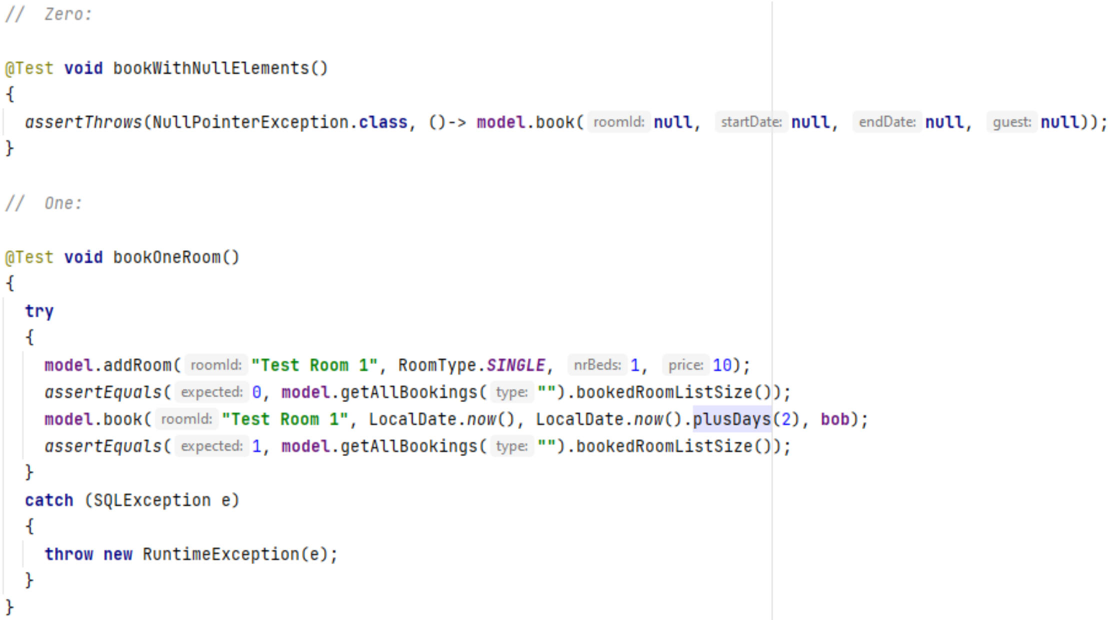
Before each test, a *setUp()*-method is run, initializing the *ModelManager*-object, removing all content from the database, and creating a Guest object to be used. This is done to ensure that each test is run on the same clean slate.



```
@BeforeEach void setUp() throws SQLException
{

  model = new ModelManager( postgreSQLpassword: "123456789");
  model.clearDatabase();
  bob = new Guest( fName: "Bob",  lName: "Builder",  email: "bob@builder.com",  phoneNr: 12345678,  username: "BobBuilder",  password: "BobBuilderPassword");
}
```

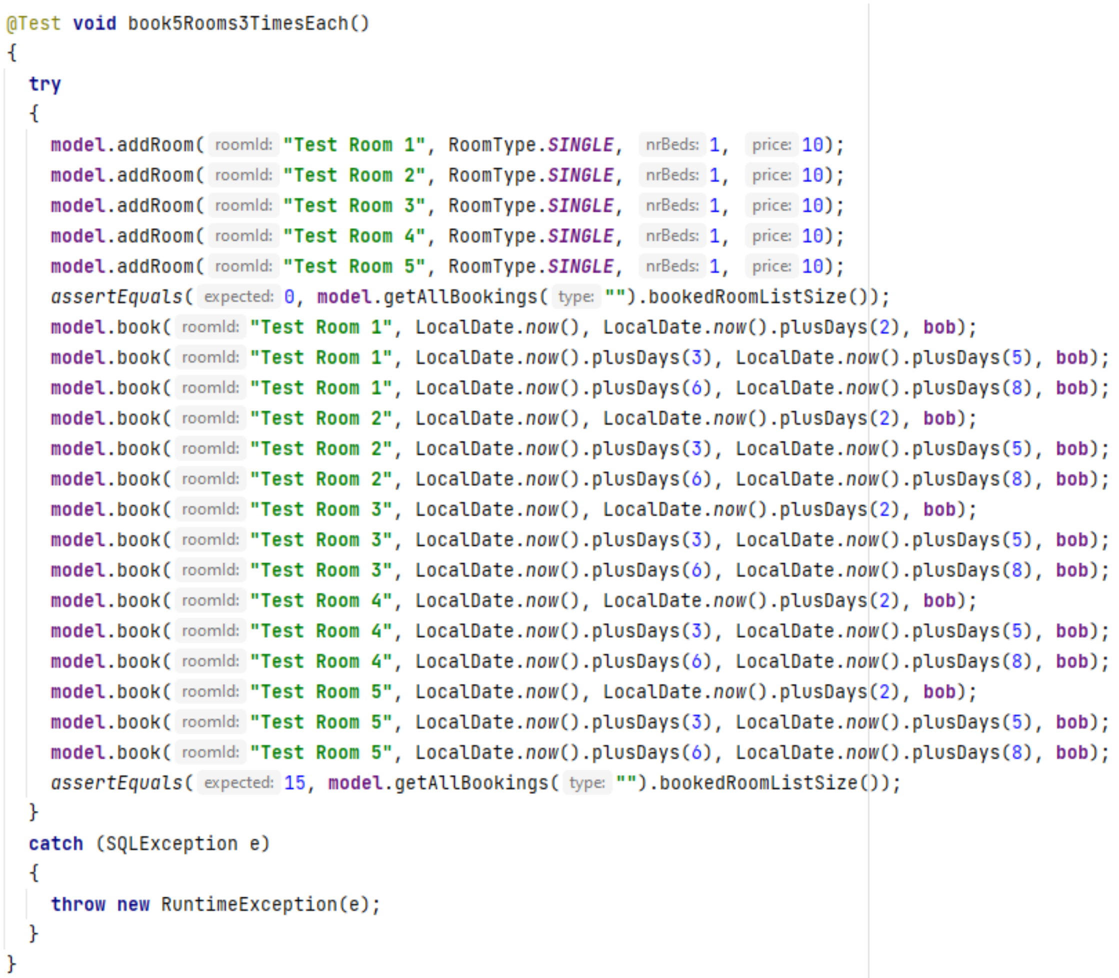
Figure 56 - setUp() method

The following figures shows a selection of the testing of the *book*-method in the *ModelManager*:



```
//  Zero:

@Test void bookWithNullElements()
{
  assertThrows(NullPointerException.class, ()-> model.book( roomId: null,  startDate: null,  endDate: null,  guest: null));
}

//  One:

@Test void bookOneRoom()
{
  try
  {
    model.addRoom( roomId: "Test Room 1", RoomType.SINGLE,  nrBeds: 1,  price: 10);
    assertEquals( expected: 0, model.getAllBookings( type: "").bookedRoomListSize());
    model.book( roomId: "Test Room 1", LocalDate.now(), LocalDate.now().plusDays(2), bob);
    assertEquals( expected: 1, model.getAllBookings( type: "").bookedRoomListSize());
  }
  catch (SQLException e)
  {
    throw new RuntimeException(e);
  }
}
```

Figure 57 - ModelManager testing



```
@Test void book5Rooms3TimesEach()
{
  try
  {
    model.addRoom( roomId: "Test Room 1", RoomType.SINGLE,  nrBeds: 1,  price: 10);
    model.addRoom( roomId: "Test Room 2", RoomType.SINGLE,  nrBeds: 1,  price: 10);
    model.addRoom( roomId: "Test Room 3", RoomType.SINGLE,  nrBeds: 1,  price: 10);
    model.addRoom( roomId: "Test Room 4", RoomType.SINGLE,  nrBeds: 1,  price: 10);
    model.addRoom( roomId: "Test Room 5", RoomType.SINGLE,  nrBeds: 1,  price: 10);
    assertEquals( expected: 0, model.getAllBookings( type: "").bookedRoomListSize());
    model.book( roomId: "Test Room 1", LocalDate.now(), LocalDate.now().plusDays(2), bob);
    model.book( roomId: "Test Room 1", LocalDate.now().plusDays(3), LocalDate.now().plusDays(5), bob);
    model.book( roomId: "Test Room 1", LocalDate.now().plusDays(6), LocalDate.now().plusDays(8), bob);
    model.book( roomId: "Test Room 2", LocalDate.now(), LocalDate.now().plusDays(2), bob);
    model.book( roomId: "Test Room 2", LocalDate.now().plusDays(3), LocalDate.now().plusDays(5), bob);
    model.book( roomId: "Test Room 2", LocalDate.now().plusDays(6), LocalDate.now().plusDays(8), bob);
    model.book( roomId: "Test Room 3", LocalDate.now(), LocalDate.now().plusDays(2), bob);
    model.book( roomId: "Test Room 3", LocalDate.now().plusDays(3), LocalDate.now().plusDays(5), bob);
    model.book( roomId: "Test Room 3", LocalDate.now().plusDays(6), LocalDate.now().plusDays(8), bob);
    model.book( roomId: "Test Room 4", LocalDate.now(), LocalDate.now().plusDays(2), bob);
    model.book( roomId: "Test Room 4", LocalDate.now().plusDays(3), LocalDate.now().plusDays(5), bob);
    model.book( roomId: "Test Room 4", LocalDate.now().plusDays(6), LocalDate.now().plusDays(8), bob);
    model.book( roomId: "Test Room 5", LocalDate.now(), LocalDate.now().plusDays(2), bob);
    model.book( roomId: "Test Room 5", LocalDate.now().plusDays(3), LocalDate.now().plusDays(5), bob);
    model.book( roomId: "Test Room 5", LocalDate.now().plusDays(6), LocalDate.now().plusDays(8), bob);
    assertEquals( expected: 15, model.getAllBookings( type: "").bookedRoomListSize());
  }
  catch (SQLException e)
  {
    throw new RuntimeException(e);
  }
}
```

Figure 58 - ModelManager testing

The following table shows the results of the tests shown above (a list of all tests can be found as Appendix 6.1):

Method name	Action	Expected result	Actual result	Passed / Failed
bookWithNullElements	book a room with all arguments given as null	throw an exception	throw an exception	🟩
bookOneRoom	book one room	bookedRoomsListSize = 1	bookedRoomsListSize = 1	🟩
book5Rooms3TimesEach	book 5 rooms 3 times each	bookedRoomsListSize = 15	bookedRoomsListSize = 15	🟩

5.2 Black box testing

To test use cases, we've used the black box testing method. In a table shown below is shown a summary of all use cases, how many passed, and how many didn't. Detailed use case testing can be found as Appendix 6.2.

Use case	Passed	Failed
Manage and view hotel rooms	14	0
Approve the booking	3	0
Book a room	4	0
View my booking	2	0
View all bookings	2	0
View all guests	2	0
Cancel a booking	3	0
Log in	3	0
Registration	4	0

6 Results and Discussion

As described in the previous section, the system underwent a series of tests to assess the functionality of the system. The Black Box testing shows that the system is fulfilling the requirements, and the jUnit tests performed on the server shows that the logic of the program is functional.

Our program currently has all functionality required from critical priorities and a significant part of high priorities. All of this functionality was tested using black-box, and white-box testing, and all of the tests were successfully passed. As for white box testing, we only included the server side since most of the logic is done on the server-side. The only part that wasn’t tested was client-side because it takes a minor part of the functionality. As a team, we did not see any of the flaws on the client side but having tested it we might find some unexpected behaviours of the system.

7 Conclusions

The purpose of this project was to create a different solution for a currently opening Beavers hotel. The system that is presently in use makes it difficult for the hotel employees to maintain the data up to date and organized. Furthermore, the booking process for the guests is inconvenient and unsustainable, if the hotel is required to compete with other hotels, as people today prefer to have access to the booking system instead of calling in (Travel Booking Statistics (2022-2021)).

The project task was to create a system where above mentioned difficulties could be fixed or well improved depending on the time constraints. According to the instructions given by the stakeholders, a set of user stories were created and later on decided into functional and nonfunctional requirements as this will be the fundamental part of the project's inception.

In conclusion, the final product is a solid working system that will be helpful when used by the administrators of the hotel as well as when new guests will be interested in booking the hotel's amenities. Although throughout the project many requirements were not completed due to time constraints, currently the system covers all the critical and many high priority requirements and works flawlessly as shown in the testing segment of the report.

8 Project future

The current product is now a well working system but is still in its infancy and has a lot of potential for improvements and expansions.

Firstly, to attract more customers, the product may improve on the design of the application as currently it has a very basic layout. Moreover, the application is still lacking many essential parts of good room representation such as for example a picture of the rooms as well as having further description of the available room, to have a better understanding of a product that is being purchased.

Besides the cosmetic changes, the application still has a lot of potential to expand in terms of more functionality for the administration of the hotel. In the future having the ability to see the statistics of the overall bookings alongside the financial details. Furthermore, the functionality from the customers perspective is lacking as well. The further version of the product could implement a review system where new guests could view what was said about the previous stays as well as the preceding guest could express their opinion so that the hotel could improve if there is a need. Another functionality that may have potential should include applying discounts for long term customers as well as more AI like generator of prices, in low seasons and vise versa.

Lastly, as the hotel is in Horsens, Denmark most of its customers will most likely be Danes, it would be beneficial to include an option for Danes that struggle with English language to be able to switch the menu between both English and Danish.
In conclusion, the project is well rounded and works well, but most mentioned improvements and extensions are necessary in the long run to add, as that would allow more organized workflow for the employees and more attractive layout for the guests.

9 Sources of information

Gamma, E. et al., 2002. *Design Patterns – Elements of Reusable Object-Oriented Software*, Available at: http://books.google.com/books?id=JPOaP7cyk6wC&pg=PA78&dq=intitle:Design+Patterns+Elements+of+Reusable+Object+Oriented+Software&hl=&cd=3&source=gbs_api%5Cnpapers2://publication/uuid/944613AA-7124-44A4-B86F-C7B2123344F3.

Larman, C., 2004. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development*,

Online Travel Booking Statistics 2020-2021, Available at: Over 70+ Online Travel Booking Statistics (2020-2021) (condorferries.co.uk)

Process Report

Software Technology Engineering

Semester 2 - Class Y

Hotel Management System
Process Report

Group 5
Christian Hougaard Pedersen (315269)
Nina Wrona (315202)
Justina Bukinaitė (315199)
Karolis Sadeckas (315225)

Supervisors:
Steffen Vissing Andersen

Date of completion:
02/06/2022

The number of characters in the main text: 29.625

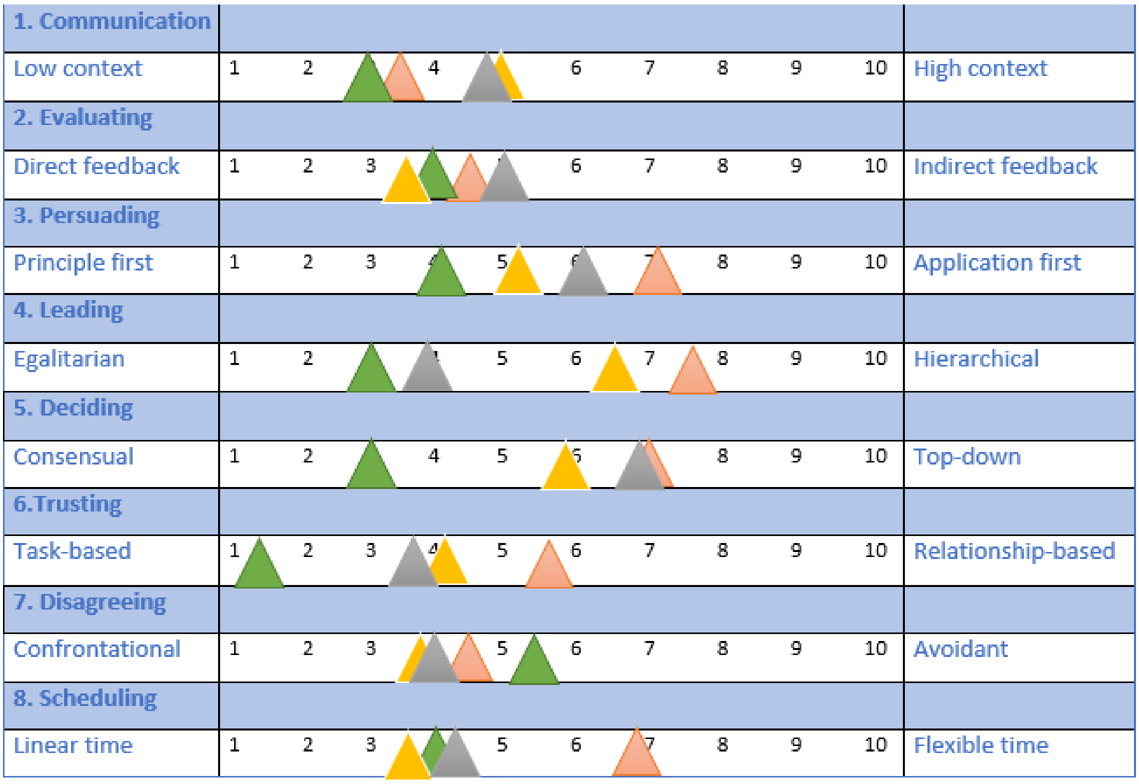
Table of content

Introduction

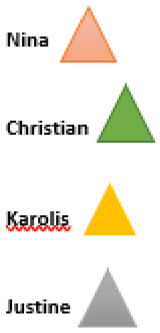
The purpose of this document is to reflect about collaboration between members of group 5 during the semester project. The names of cooperatives are Justine Bukinaite, Karolis Sadeckas, Christian Hougaard Pedersen and Nina Anna Wrona.
During Semester Project group 5 followed Unified Process including iterations following one another, as well as SCRUM framework. Starting from Inception, Elaboration, Construction and finishing with the Transition phase we went through all UP phases. Group 5 has gone through a huge progress while working on the Hotel Booking System with the supervision of Steffen Vissing Andersen. Starting with chaotic meetings, working at school every day and finishing with organized assemblies with agreement about flexibility to work at home.

Group Description

We all have different backgrounds and personalities. The most visible difference is that we come from different countries. Nina comes from Poland, Christian from Denmark and both Justine and Karolis are lithuanian. We analyzed our group's cultural map and realized that the nationality did influence the way we collaborate and communicate with each other.



1. Communication											
Low context	1	2		4		6	7	8	9	10	High context
2. Evaluating											
Direct feedback	1	2	3			6	7	8	9	10	Indirect feedback
3. Persuading											
Principle first	1	2	3		5		7	8	9	10	Application first
4. Leading											
Egalitarian	1	2			5	6	7	8	9	10	Hierarchical
5. Deciding											
Consensual	1	2		4	5	6		8	9	10	Top-down
6.Trusting											
Task-based	1	2	3	4	5	6	7	8	9	10	Relationship-based
7. Disagreeing											
Confrontational	1	2	3		5	6	7	8	9	10	Avoidant
8. Scheduling											
Linear time	1	2	3		5	6	7	8	9	10	Flexible time



Firstly danes are known for giving and being used to receiving direct feedback from the supervisors and colleagues. Fortunately in our group everyone had the same preferences in this field and as a result we did not encounter problems with misunderstanding the intentions of giving precise and clear criticism or approval of the work.

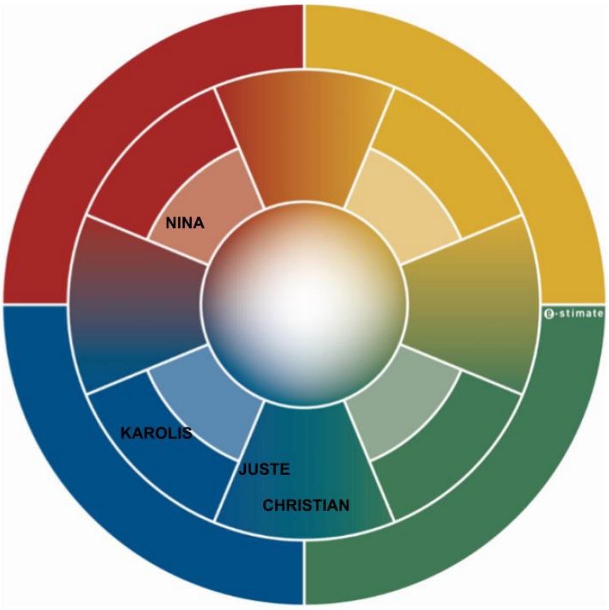
There is a noticeable difference between our group's approach towards supervisors and hierarchy in the group. Eastern European part of our group sees the supervisor as a person who is more distanced as a result of the hierarchical power he has at his current position in Via. In contrary Christian did not see the status gap separating the students from the teachers. That did not cause conflicts, however during discussions we could see the difference in the approach. For example, when the group wanted to organize a meeting with the supervisor, and the most suitable date for that would be a Danish holiday, Christian did not hesitate with writing to Steffen if the meeting would be possible at that exact date. On the other hand Nina was really concerned about not disturbing his free day and was expressing that she thinks it is rude to make such a request.

When it comes to scheduling the meetings, most of the group was coming later than initially said, so Nina having a more flexible time approach, did not cause any disagreement. The group has agreed that as long as the tasks are done on time, one can come late.

Group 5 contained 3 people with the majority of blue personal profiles. Two of them have social-oriented attributes, which means that they were partially green as well. This involved Justine and Christian. Karolis is also on the blue side of the diagram, however with some attributes of yellow. Nina has a red personality with a touch of green.

Justine was the Scrum Master. She used her detail-oriented personality to create structured meetings, and alongside, their documentation. On the other hand she encouraged the group to socialize more. Intention was good however there were only a few times when the team met having no project related purpose.

Nina took the Product Owner role during the Review and Sprint Planning meetings. She was taking advantage of her dominating personality and always aimed high with choosing the amount of work for the upcoming Sprint. During the review meetings she tried to test all features of the program to make sure that everything that was added to the program did not interrupt the workload of the previous version.



Chris, Karolis and Justine, as Team Members, were debating with the Product Owner about the product and how much they are able to accomplish in three days. Karolis was constantly coming up with new ideas on how to improve the program, which the Product Owner could consider. Christian, on the other hand, was always protecting the team from the Product Owner. Whenever something did not work, he was coming up with arguments that were supposed to take the blame off the group.

During Daily Scrum and Scrum Planning Meetings there were times when Team Members with blue personalities were analyzing all details of the feature or programm, that Nina, with red personality, found unnecessary and time-wasting. To solve that issue group 5 decided to set a timer for every meeting, so that it always takes the same amount of time. Enough for discussion, but not enough to annoy the red personality.

Project Initiation

Group 5 was created at the beginning of the semester. Justine, Christian and Karolis were working together on a SEP1 project and they decided they would like to work together again. Nina joined the team to fulfill the requirement of having 4 people groups.

In the group contract the group has included rules and consequences of violating those rules. We agreed to meet consistently during both the tuition and project period and notify each other when a person is going to be late. Thankfully we did not have to manage any conflicts caused by not following these rules.

Team has planned the project firstly with Background Description. Subsequently members followed SCRUM methodology. They have built a Product Backlog, using Product Owner’s description of the system. Additionally group 5 has prepared a time schedule in order to prevent setbacks before the deadline. These plans were a helpful backbone during the project period. However, according to appearing circumstances, the Product Backlog as well as the time schedule were updated during the implementation of the product.

Project Description

Group 5 has defined the problem based on the Product Owner’s story. The Product Owner had said that he struggles with collecting and keeping all the booking, customers and rooms data structured and up to date. Following his description, the team came to a conclusion that his current method of booking rooms in the hotel is inefficient. Members determined the definition of purpose as *“The purpose of this project is to provide our customer with a tool to manage room booking of the hotel in order to avoid overbooking and help to collect hotel customers’ details in a structured way”.*

Group 5 aimed for creating a useful, easy to manage tool, that would take the burden of writing everything manually by the manager of the hotel or the receptionist.

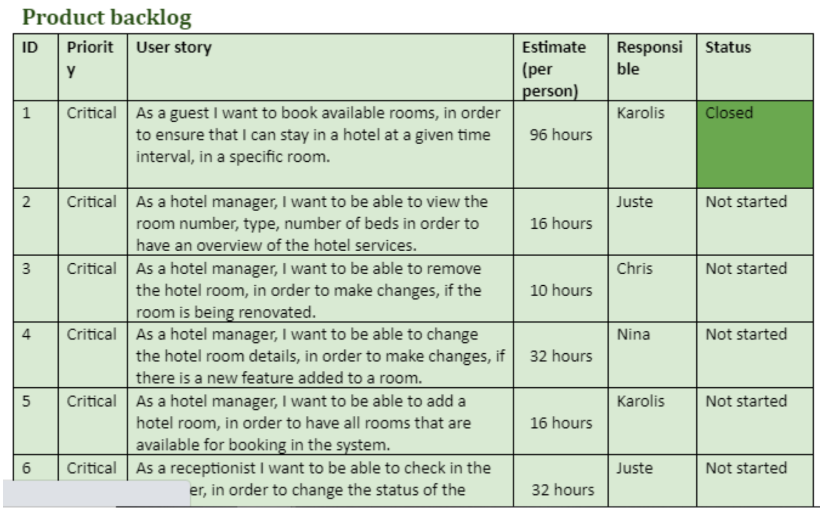
The initial goal of the project was to create a fully functional booking system, with separate access for hotel manager, receptionist and guest. The Hotel Manager was supposed to just manage rooms in the hotel, the receptionist to manage booking details as well as guest information and Guest aimed for booking rooms in the hotel. Additionally the finances were supposed to be tracked on the Hotel Manager’s account. This plan turned out to be too ambitious. The group has not considered the fact that the Product Owner may want to add additional requirements according to his preferences, which changed the original plan. The Product Owner decided to add additional functionality for guests in order for them to be able to change their booking information alongside with their personal details. As to the hotel manager the additional functionality was to cancel chosen booking and the receptionist to check in and check out guests.

Even if we did not implement all expected requirements, the primary goal was realistic from a timewise perspective and we did achieve the expected amount of Use Cases.

Project Execution

After the Inception phase group 5 began with the Elaboration phase, according to the Unified Process. In UP every phase contains iterations, which in this case were called Sprints. Team has combined the development process with SCRUM methodology to increase the efficiency of the work.

After we had learned about the purpose of incorporating SCRUM, we were very motivated to try out this system. The huge advantage of these two methods is that they are strict and clear. The UP process helped the group to distinguish the periods of the project. During Inception the group has focused only on outlining key requirements, scheduling milestones, assessing the risks and creating an initial project plan. Then, during Elaboration, we have focused on creating Use Cases for the most important to the system requirements, which was followed by making a Product backlog. Package diagrams and basic class diagrams were created at that point. Afterwards during the longest phase, Construction, the design of the system was created followed by its implementation. This part is divided into multiple iterations, where the whole program is built. Finally at the end of the last iteration the product was ready to launch.



Product backlog

ID	Priority	User story	Estimate (per person)	Responsible	Status
1	Critical	As a guest I want to book available rooms, in order to ensure that I can stay in a hotel at a given time interval, in a specific room.	96 hours	Karolis	Closed
2	Critical	As a hotel manager, I want to be able to view the room number, type, number of beds in order to have an overview of the hotel services.	16 hours	Juste	Not started
3	Critical	As a hotel manager, I want to be able to remove the hotel room, in order to make changes, if the room is being renovated.	10 hours	Chris	Not started
4	Critical	As a hotel manager, I want to be able to change the hotel room details, in order to make changes, if there is a new feature added to a room.	32 hours	Nina	Not started
5	Critical	As a hotel manager, I want to be able to add a hotel room, in order to have all rooms that are available for booking in the system.	16 hours	Karolis	Not started
6	Critical	As a receptionist I want to be able to check in the er, in order to change the status of the	32 hours	Juste	Not started

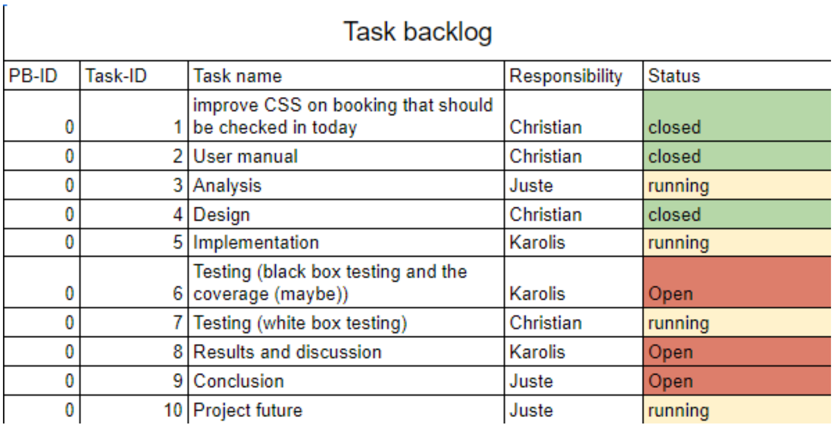
Transition, last project phase, included documentation and delivering the full product to the Product Owner.

All iterations were led by SCRUM methodology. Sprints carried on for three days with the leadership of Justine, our SCRUM master. Every Sprint started with the Planning

Meeting, which helped the group to structure their tasks, by building the Task Backlog. The burndown chart was then updated as well.

First few Planning meetings were disturbed by one of the team members, who wanted to discuss the ideas for their implementation. Afterwards the Scrum Master introduced a time limit for the meetings, so that everything can go smoothly and according to the meeting plan. On a daily basis we had Daily Scrum meetings to keep each other updated and request help if needed. Depending on the day it was either at noon or 2PM. Sprints were always finished with Review and Retrospective meetings. At the end of every Sprint Product Owner, Nina, was assessing the outcome of the work. Afterwards the Retrospective Meeting was held. During that last meeting each group member reflected on what went well and what is there to improve, regarding work environment and group atmosphere.

The retrospective meetings were crucial for the team workload development. For example, teams have come up with the idea of working from home during the Sprint. In the next retrospective meeting we have concluded that 3 days from home and online meetings are less efficient then the ones made face to face. On the other hand, the group did not want to resign from the advantages of working from home. Finally we tried to have one day of the Sprint as home office work, which worked perfectly. On top of that we have learned to divide sprint tasks into smaller pieces and pay more attention while merging the code in the GitHub platform.



Task backlog

PB-ID	Task-ID	Task name	Responsibility	Status
0	1	improve CSS on booking that should be checked in today	Christian	closed
0	2	User manual	Christian	closed
0	3	Analysis	Juste	running
0	4	Design	Christian	closed
0	5	Implementation	Karolis	running
0	6	Testing (black box testing and the coverage (maybe))	Karolis	Open
0	7	Testing (white box testing)	Christian	running
0	8	Results and discussion	Karolis	Open
0	9	Conclusion	Juste	Open
0	10	Project future	Juste	running

As a result of following the SCRUM methodology we had a fully working program at the end of almost every Sprint. We also managed to avoid the situation where two people were accidentally working on the same task. This was achieved by tracking which tasks were not started, started or finished, in the Task Backlog.

The group was satisfied with the final outcome of the project. The system is visually appealing, functional and ready to launch. We have fulfilled all the requirements chosen by the Product Owner and all of them were approved by him. The least successful part of the project was that our group has never put effort into getting to know one another. That was not necessary for the project outcome, however we believe that it would be more enjoyable to work with people known from not only work-related perspectives.

Personal Reflections

Christian:

For me, working on this semester project has been, by far, the best experience I have ever had working in a group. The process, even if challenging at times, has been an interesting and fulfilling learning process.

During the previous semester project, my group unfortunately had to let one of our team members go, so when it came to forming groups for this semester, we were one person short of being a four-person group.
To our luck, Nina suggested that she would like to join our group, and we all quickly agreed upon that.

This, of course, led to a change in the combination of nationalities in the group as we now had two Lithuanians, a Dane and a Pole. Even though English is not the first language of any of us, all communication in the group has been done in English without any problems worth mentioning, even if I sometimes struggle with finding the correct way of expressing myself in a concise manner in English.

Culturally, we have not encountered any conflicts, partly because of the similarities of our cultures, and also in part because everyone has had some experience working with people of other cultures. As a Dane, however, I have been quite aware of the difficulties other nationalities often have in understanding us, because most Danes, myself included, use a lot of sarcasm. Because of this, I have tried to make myself as understandable as possible when communicating with my groupmates.
In my opinion, it has been really beneficial to have learned about cultural differences, as a lot of misunderstandings can come from simply not having the same understanding of a given situation and how to act according to this.

When creating the group contract, it was quite important to Karolis, Justina and me to put more effort into making it precise, as the lack of precision regarding the rules was

one of the biggest obstacles we faced last semester. Thus, the rules in the group contract for this semester were far more precise than last semester. For example, for this semester, it was decided that we would meet at least once per week during the tuition period, more specifically, on Wednesdays and that if any person in the group was having problems with understanding parts of the course material, we would all help to the best of our abilities.
In practice, for most weeks, we ended up meeting up at least two or three times a week to work on course-assignment or to help each other when problems arose with the course-material. This has been quite beneficial, as we of course all have our different strengths and weaknesses regarding the course material, and most oftenly, explaining a theory or such to someone else helps quite a lot with your own understanding of the subject.

One thing I would like to include in a future group contract, if the SCRUM framework is to be used, is a rule stating that working from home should only be done on the second day of the sprint as we came to realize the importance of having a full day to get everything combined and ready for the sprint review. We, unfortunately, experienced this when, one scrum period we worked from home, and only met up for the last half of the last day. This led to us not being able to have anything approved by the Product Owner.

The experience of having worked on a, even if smaller in scope, similar project last semester has shown itself during this project in many ways. The most obvious, for me, is that we have all gotten better at listening to each other's ideas and points of view, and being constructive when discussing potential solutions to problems encountered.

As the personality types of Karolis, Justina and I, are primarily blue (based on our E-stimate Personal Profiles), we tend to spend a lot of time discussing potential solutions and details about every aspect of the project and in these situations, it has been beneficial to the group to have Nina as a member, as her personality type is primarily red, meaning that she has been really good at stepping in and stopping the discussions in order to not spend more time on these than necessary.

Of course, working on this project has not been without conflicts but luckily we have been able to keep them from escalating past the third step of the conflict escalation ladder, meaning that we have always managed to find a common ground instead of the argument escalating and becoming personal, instead of being about the subject at hand.

As described earlier in this report, this project has been made using the SCRUM and Unified Project frameworks, and as this was the first time using these frameworks, some difficulties were expected. Luckily, we were able to have our supervisor, Steffen, present for the first planning- and review meetings. This was especially helpful for me, as having him present, and giving us feedback after the meetings gave me a sense of comfort in that we were on the right track.
I think, that we quite easily adapted to following SCRUM, and because of the predefined structure present when using SCRUM and UP, you always had a set goal in mind so I was never in doubts about what to do, as was quite oftenly the case last semester, as the project then was made in one iteration, so it was not until the very end of the writing period that the system actually came together.
As a person, whose primary personality type is blue, I however have had some difficulties coming to terms, and adhering to the principle regarding that as soon as a task is marked as done, it is done and should not be changed any further. I seem to always be able to find small details I would like to change, and for the first sprints, I had a hard time convincing myself not to go back and change existing things in the system. However, both on a personal level but especially when working in a group, I have come to understand the advantage of working this way, as it means that at any point of the process, also partly due to the daily scrum meetings, everyone knows the progress that each of us has made.

The experiences I have earned during this project are definitely something I will carry with me moving forward, as I see this project as the best experience I have had working in a group.

Justina:

Project work process

Our group went through a natural formation process, as we were already three people that wanted to continue working together. Fortunately, Nina, the newest member of our team, offered to join and since she seemed like a perfect fit, as she had certain personality traits that we were missing, we were happy to form a group with her.

Finding a topic for the project was an easy decision, because after a bit of brainstorming we had two topics in mind and after talking it over with our supervisor we chose a topic that seemed a bit more fit for the second semester students, whereas the other one might have been a bit too complicated.

When we started the planning process, in order to be on top of things, we started documenting everything in Meeting minutes, where we wrote down the discussions of the current meeting and topics for the next one. Following that, we used Google docs to organize our groups files and messenger or discord to communicate with each other.

After choosing the topic for the project, in order to get a better understanding of what kind of products similar to our project are on the market, we looked up some examples online. After considering what would be a realistic goal for us as second semester students, we made a logical decision to stick to the basic hotel management and booking system that is not part of the website and has basic requirements.

Since this semester we were introduced to the concept of SCRUM, that is a framework for teams to follow when working on a project, we tried implementing it into our project process. The project period had 8 Sprints, meaning it was divided into 8 parts and during each sprint we kept taking extra requirements that were initially made by us.

After each sprint we held a review and retrospective meeting and after that a new sprint planning meeting.

Using SCRUM was a helpful organizational decision to keep track of the progress of the project, as well as to constantly be on top of what tasks are being done and by who. That being said I feel that it would have been better implemented if we would have had a longer period of working on this project and the sprints would have taken more than three days. Having only three-day Sprints sometimes felt that we were having more meetings than actual work. However, using SCRUM was a learning curve since it has taken us time to get used to when having meetings to stick to the relevant topics. However, it improved over time and in the future, we will be more capable of sticking to the SCRUM rules, since now we are more familiar with the framework and how to adjust it to our specific team work process.

In my opinion, we successfully achieved our goal that was set in the beginning of the project. Although we did not finish all the requirements that we created, I feel that we were productive and followed the SCRUM correctly, to fulfill the goal of having a working product that has a basic booking and management system for the hotel.

Teamwork

In the beginning of the semester, we have signed a group contract that specifies what is expected of the group mates and the consequences, if the tasks are not fulfilled. I feel that all of us followed it to the best of our abilities and that has led to good teamwork and a pleasant time together.

Being a team SCRUM master, I feel I fulfilled my duties well, even though as most things this semester it has been a learning process and I had to improve, so that my group could fully gain the advantages of having a SCRUM master. I sometimes struggled to be more strict when the topic throughout the Sprint planning meeting went off track, because I felt that it was difficult to not sound too rude.

Our group was determined to do a great job and therefore we worked to the best of our ability. During the Sprint planning meetings, everybody took the tasks that they felt is best for them and that usually led to us finishing the tasks assigned. Even when someone struggled with a task there was always someone else that may help and working in pairs often helped to improve understanding of the system better and achieve more difficult tasks faster. I feel like our team did not struggle too much with motivation, although in some instances when we struggled to get something working, it was a very exhausting process.

Being in a group where people come from different countries as well as from different backgrounds wasn’t very influential on teamwork as we all come from EU countries. That being said, all of us having different character traits have been influential on the work. During the semester we completed a profile test, where each of us were put in a certain color category that contains a variation of certain types of people. As three of the people in our group had mostly only blue and some green tendencies in their profiles, we have noticed that in the first semester that has really influenced our workflow. We always had the tendency to focus too much on one problem and that always took too much time. Whereas this semester we had Nina who has red profile tendencies and therefore, she really has helped us to not get stuck on one problem and always keep moving forward as well as keeping up with the deadlines and being more thorough at meeting up.

In conclusion, while working on this project in a team, I have developed better time-management skills, since I had the role of a SCRUM master. Following that I learned that having a framework can really improve overall performance of the group as well as it helps to keep track of distinctive tasks and roles that everybody poces.

Nina:

Working with Christian, Justine and Karolis was a pleasure. I felt calm and relaxed during the whole semester project, from the beginning to the very end. Structured meetings, clearly identified tasks and the flexibility of working at home was a huge life-changer. I had a lot more internal motivation, as a result of being assigned specific tasks at the beginning of every iteration of the project. I knew exactly what I needed to do and how fast I needed to be to make it.

During the project period I have felt responsible for keeping my group on track. By taking the role of the Product Owner I could lead the group through tasks that I found relevant. I believe that the outcome of this action was having strict rules and tasks to accomplish in a specific amount of time, which kept the group in the time schedule.

All group members delivered maximum to the group. We decided to divide tasks to utilize each member’s expertise. However if one of the team members wanted to broaden one's knowledge about a certain subject, we have been assigning him to make the part that included it.

The biggest challenge of working with two Lithuanians and a Dane was that I did not always understand their sense of humor. I did not know if what Christian was saying was sarcasm or a joke.

I have learned a lot by making this project. Not only coding, but also I discovered that you do not need to use stress as your motivator, but pure joy and excitement to deliver a good product. Both ways are motivating you internally and last longer than external motivation like one’s parents expectations. Nonetheless it is nicer to frame your mindset in a way you see this project as an opportunity instead of a requirement.

I have worked a lot more with my group colleagues, however I coded mostly alone. Last semester I told myself that I should work more with my group of fellows. And initially I thought about coding. However now I have come to the conclusion that working together should mean communicating clearly, making sure everyone knows exactly what to do and discussing your ideas of implementation. It is not necessary to code together, but it is necessary to make sure the group idea of the project is consistent and clear.

The huge advantage of Problem Based Learning is that we, as Developers, are not caged by the borders of specific exam questions. Contrary, we use our creativity to lead us through the project. We formulate a problem, which gives us the basics to come up with an appropriate idea to fulfill the customer's wish. And we follow up by brainstorming and looking for suitable solutions. As a result we create a goal that we want to achieve. In our case a program we want to deliver.

Karolis:

This semester I was excited to start working on a project because I've wanted to use all gained knowledge during this semester. Taking a closer look at the programming part of the project, I've felt more confident this semester because of having some kind of vision regarding creating simple systems that developed throughout the first semester. During project preparation, I felt good about the idea that we can make our decision and choose the topic ourselves. We've decided to develop a room booking system for a hotel. The idea seemed simple and maybe too basic, but on the other hand, I was happy with the decision because I knew that there wouldn't be a problem with understanding the fundamentals of the program. The simplicity of an idea allowed us to spend more time focusing on creating the program itself and selecting suitable approaches to each situation.
Personally, during this project, the newest thing for me was following the Scrum framework, but luckily during the first few sprints, I got to know the good part of following a scrum. I saw how the scrum framework followed by the UP process gave the project a strong structure. It was much easier to see only this sprint vision and focus on it, even though, in some cases, it was a little hard not to take a peek forward and make additional tasks that are not in a backlog. Overall, it gave the project more structure, which stands a big part of working on a project that includes several people. It was always clear to me what tasks I was responsible for and the overall vision for each sprint. Being a more blue-type personality gave me comfort knowing that our project is under great structure and constantly on the right track.

Reflecting on our team, I was happy with our productivity and ability to make decisions as a team. It was a productive and faster pace than last semester. Mostly it was because of Scrum but nether-less our new group member personality which scores most of the points on the red side which also became great product owner for this project. As out-come we've had more balance this semester with the amount of time spent discussing details versus actually making decisions. Compared to last semester, when all of our members had the bluest personalities, it used to be hard to get out of the loop of discussing small details that didn't change a lot. So for this semester, it was a joy to have an excellent balance and keep it simple while making significant progress.

Supervision

The team's cooperation with the supervisor was satisfactory. We had no problems with reaching Steffen. He was always available for us through email, Discord and face to face meetings. He has been answering our questions about the details of following Unified Process, as well as Java-related questions. He also attended two of group 5 SCRUM meetings and gave us helpful feedback, which we analyzed and corrected in later gatherings.

Conclusions

To give an outline of our reflections we have constructed a few points which could help the team in the future group work.

We do recommend keeping the flexibility to work at home for people, who are more productive that way. Create the agenda for meetings before they start, to keep them short and precise. On top of that, meet outside of work in order to create the environment where everyone is feeling comfortable. However, we do not recommend making important meetings via network and keep meeting at least once during the Construction phase iterations. We have experienced that not seeing one another results in chaotic group work and misunderstanding each other's intentions.

All in all we hope this knowledge will stay with us until the next group work and that we will avoid making the same mistakes next time.

Resources

https://via.itslearning.com

https://www.ohio.edu/university-college/sites/ohio.edu.university-college/files/Internal-vs-External-Motivation.pdf

https://studienet.via.dk/projects/Engineering__project_methodology/SitePages/Home.aspx