1. Purpose

This document describes the software intended for use at the front desk at the Linnaeus Hotel. The software is bespoke and designed from the ground up for this express purpose. As such, the software handles all reservations, keeps track of room availability, guests accounts, and bills.

The software keeps a list of the rooms in the two hotel buildings, and their attributes. The clerk can search these, to find the ones available, and suitable for some clients' demands (Req_01). These can then be booked, that is, reserved for that specific client during a specific time (Req_02).

Later, it keeps track of if the customer has arrived (Req_04), and if they have subsequently left (Req_05). If they changed their mind, a booking can be cancelled. If this happens too late, a fee is automatically added to the account (Req_07).

Clients data is stored as well, allowing for repeat visitors to be rewarded, and bills to be tied to a person (Req_02, Req_06).

This document describe how this is to be achieved.

2. General priorities

When designing the system we wanted to keep a clear MVC separation, to facilitate easy changes and additions, and housekeeping of the system. It also needed to be fast, because of the two speed constraints given to us by the Linnaeus Hotel (NonFunc_01, NonFunc_02).

Beside the MVC, we focused on making several reusable and discrete components that interact to create the specified system.

As the system was realized as a three-tier application stack (a client in each hotel building, both communicating with the same server, who in turn stores and retrieves data from a database), there had to be a way of communication between the layers. We wanted to keep this communication simple and stateless.

3. Outline of the design

After analyzing the legacy software supplied, we came to the conclusion that it had too many flaws and idiosyncrasies, which led us to settle for a total rewrite. It didn't fulfill all the requirements, and some of them would be very hard to implement, requiring something akin to a rewrite but within the constraints of an outdated system (see the separate document “Analysis of the Jhotel software” for additional details on this).

We opted to make a new system from scratch, which would be more fitting for our purposes from the start. This system would be divided into three parts, the *client*, the *server*, and the *database*. The server would take requests from the client, and make all operations on the data (e.g. make reservations). The client would be the control panel used at the hotel desk, querying the server for data retrieval or entry. Each hotel could use one instance of the client each, both connected to the same server, to keep the data in one place.

Our model is centered around the *booking*. The *booking* has a time span, references several *rooms* and a *customer*. The *room* has a room number, assigned hotel, quality level, view, and so forth. It also keeps track of neighbouring rooms. The *customer* holds the detail of a specific customer, such as name and address.

The client will ask the server for a list of rooms at startup, and keep them in memory; they are not very prone to change (and in fact, changing them is outside the scope of this system). However bookings and guests are queried for continuously and are not kept locally, to assure they are always up to date.

The client consists of a class *FacadeController* that serves as the core of the application. This class makes sure to display the right GUI window at the right time, and holds references to any data currently stored in the client, and queries the server for data retrieval/storage at the correct times.

The server, being just a server, revolves around a realization of the *AbstractServer* class. It sits idle and waits for any incoming requests from the client, then decides what action to take in response, mostly retrieving things from the MySQL database, and repackaging it in a way the client understands.

4. Major design issues

The first and most influential decision on the whole process was whether to attempt to re architecting the old jHotel software, or to design a completely new system. This warranted an analysis of jHotel, which was not favourable (see the document “Analysis of the jHotel software” for complete details surrounding this decision).

We discussed if we wanted to build a three-tier application (client, server, database), or if a monolithic approach would be better. Because there were nonfunctional requirements specifying the maximum times for some operations (NonFunc_01, NonFunc_02) a monolithic approach was tempting because of the high speed, because every call would be internal. But we quickly realized that we would run into issues because there are two hotels; they would both need to keep track of what was going on in the other hotel, so we could not avoid network communication. Better than to embrace it, because with a client-server relationship we could keep a central database, thus having only one state for both hotels, never having to bother about netsplits, and the cleanup after those.

The next question then was if the client was to be thick or thin. We decided to keep it pretty thin, to continue the thought of having a low chance of the two hotels having disagreeing information. The client would not keep any data after it was done being displayed. Later we decided the clients could in fact have the room data in local memory, as this was deemed relatively static.

The MVC pattern felt pretty natural, and we didn’t think of any real contenders. It is a well known, tried and tested pattern, and we felt confident in using it. Nevertheless, this was a decision that we took, so we still added it to the tree (see figure 1 for our decision tree).

One of our greater issues with the jHotel software was the home cooked database, and we wanted to avoid painting us into another similar corner - the JDBC has support for a wide array of relational databases, so there really was no reason not to use one of them, as they also are very fast, indeed far beyond anything we could create.

The last major decision was regarding the client-server communication. To send data back and forth, some scheme had to be devised. Most importantly, how would the object data be transmitted? Between JSON and XML, JSON has a slightly smaller footprint, and is easier to parse.

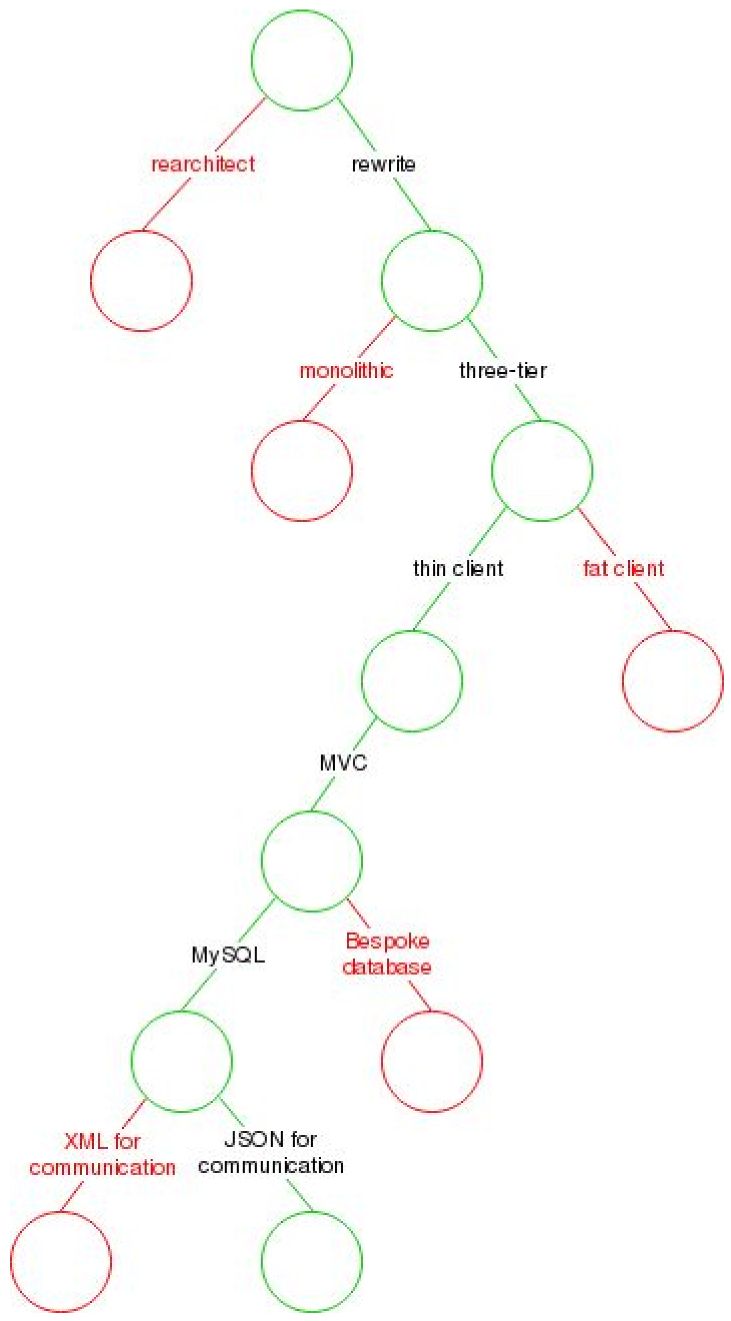


Fig 1: decision tree

5.1 Software architecture

In this section we will discuss the architecture of the delivered software. We will mention a few patterns that has been implemented and also discuss pros and cons of these choices.

5.1.1 The Facade pattern

We have chosen to use the Gang of four(GoF) Facade pattern to be used in the client side. We call the facade class “FacadeController”, and it is basically a single class that is responsible for receiving method calls and send them towards the correct class. Advantage of using this pattern is that it hides the underlying system and anyone implementing new functionality would have easy to find how to use the previous functions since the main ones all go through the facade. This pattern is quite negative for patterns like low coupling and high cohesion since most classes will need an association to the facade, and since the facade needs to take care of all kinds of calls, that is not very high cohesion. But with the pros and cons compared we decided that it was a good choice for this application to have a facade in the middle.

5.1.2 The proxy pattern

Another GoF pattern that we used is the proxy pattern. The proxy pattern can be used in multiple ways, and the way the proxy pattern was implemented is called “virtual proxy”. This was chosen for two reasons, both to save bandwidth which leads to a faster application, but also for security. Since it’s implemented to the Customer class one must consider if it is necessary to send personal data like personal number, passport number, and credit card information when it is not needed. Therefore a Customer interface is used then two classes, a RealCustomer which is in the model and a ProxyCustomer which is part of the controller. When a server request for bookings is made instead of the RealCustomer, a ProxyCustomer will be used which only have id, first name, last name and a RealCustomer. Initially the RealCustomer is null, but the ProxyCustomer has a private method to ask for the full data which will be called if a field asked for is null, i.e. if passport number is called for. Then next time the passport number is called the proxy notice that it has a value and simply pass the value from the RealCustomer. The proxy pattern for customer can be seen in the class diagram figure 4.

There is another important aspect of the Proxy pattern to be considered is the security. This is not implemented, but if in future updates the client request that customer details will only be sent if a separate security check first is passed, this new feature can be added to the ProxyCustomer without too much designing. This type of proxy is called the protection proxy.

5.1.3 Iterator pattern

The software has been designed with encapsulation in mind. Iterator pattern is part of this and simply instead of letting a class holding and dataset returning the whole List as a class a iterator is used which hides the data structure and will not let classes using the List fiddle with it in any unintended way.

5.1.4 General responsibility assignment software patterns(GRASP)

GRASP is a set of pattern which is important to consider in any object oriented project. Our project is no difference. Most part of the application is trying to keep low coupling and high cohesion. Indirection is thought of for many intermediate classes that your forward the call rather than get the class that has the actual call. Creator is thought of to decide where to create the different objects. Polymorphism is used by the proxy pattern but not much more than that.

5.1.5 Model View Controller(MVC)

In the client side a MVC pattern has been used, however the model of the client side is just a temp holder for the objects, and all business logics is in the server. But still thought it was nice to keep MVC separation so the temp model holder has no dependencies outside the model package.

5.2 Design Components

In this section we divide the software architecture in its component and explain each of these.

5.2.1 The set of components

1. **GUI/Controller**
   This component consists mainly of fxmls, the handling of these and their respective controller class. Most of the controller classes are linked to the FacadeController class, since the FacadeController is the bridge between the controller classes and the switching of screens and fxmls. The fxml controller classes consist almost entirely of methods and functions for the graphical interface. The FacadeController class is not a part of the view component per say, but serves as the bridge between the ScreenController and the different fxml controller classes.

   The interface needed for this component is user interaction, and the provided interface is a graphical display of the model as well as a request to update the current model.

2. **Facade/ClientModel**
   This component is the backbone of the client. It holds a temporar model which it replaces as soon as the view changing the current window, this to assure that a current model is up to date to other hotels running on the same server. It has two input interfaces, where both is basically requests to update the local model, but one is connected to the GUI/Controller and the other to client communication. The provide

interface is commands to either update the view or to update the model, the later is connected to the client network communication component.

3. **Client network communication**
   This component is a set of classes that handles messages to and from server.
   It has required interface that takes messages from server, parsing it and deliver a call to the provide interface which is a network transmission. It also has an required interface on the other way that takes an input from the server, parse the message and calls the appropriate call as an provide interface.

4. **Server network communication**
   This component is the counterpart to the *client network communication* component. It has a required and a provide interface toward the client, and another pair toward the *data access object*. It's job is to receive messages from the client, extract the instructions, and with the help of the DAO create a response to send back. All communication is initiated by the client.

5. **Data Access Object(DAO)**
   The *data access object*, or the DAO, is responsible for querying the database, and fetching data from it. Any data that is required by the client also has to be transformed into a format that the client can work with.
   It has a requires interface toward the network communications layer, where it wants to receive an instruction. In the same way, it has a provide interface toward the database, where it in turn gives other instructions, for data storage and retrieval. To collect any data from the database, it has a requires interface there, and predictably, also a provides interface toward the network communication component.

6. **Database**
   The database module is where the data is stored. This is always on, always in RAM, and made specifically for speedy retrieval of more or less specific information.
   It supplies a requires interface, where the DAO queries it with SQL statements, and a provides interface where it presents its results.

5.2.2 Component Diagram

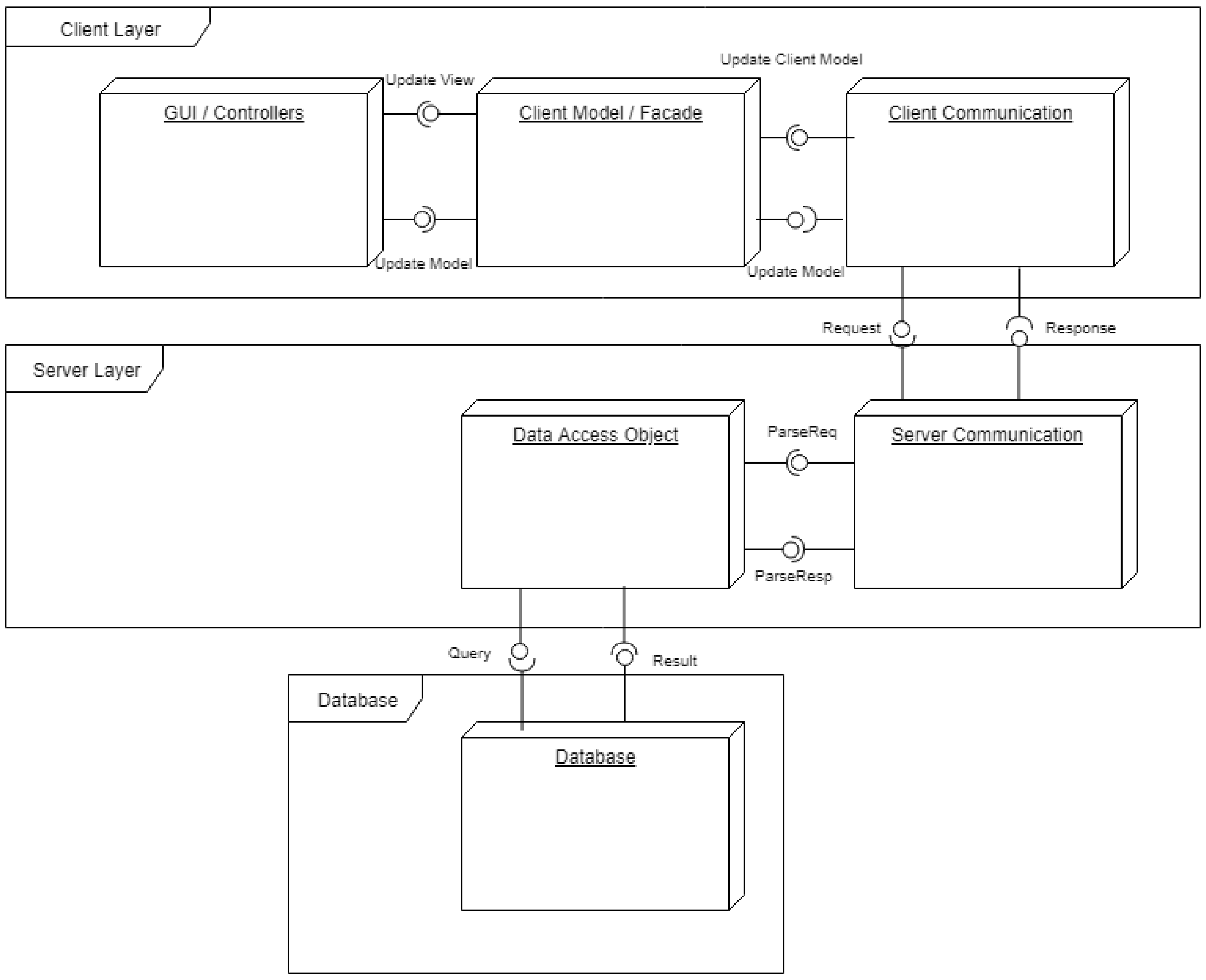


Fig. 2: Component diagram

5.3 Class diagrams

In this section the class diagrams is presented. Each component can be seen but to get a better overview we merge the server components to one diagram and also the Facade client model component with the client network communication component.

5.3.1 The GUI / Controller

Here (fig. 3) we can clearly see that almost all controllers are connected to the facade controller. The facade controller is not a part of the view per say, but is the bridge between the fxml controllers and the ScreenController class, which handles the changing of screens and their respective controller classes. The fxml controller classes consists mainly fxml properties and functions for the controllers and containers in the fxml file. ScreenController also has the enum class Screen, which is just a string to the resource location of the fxmls.

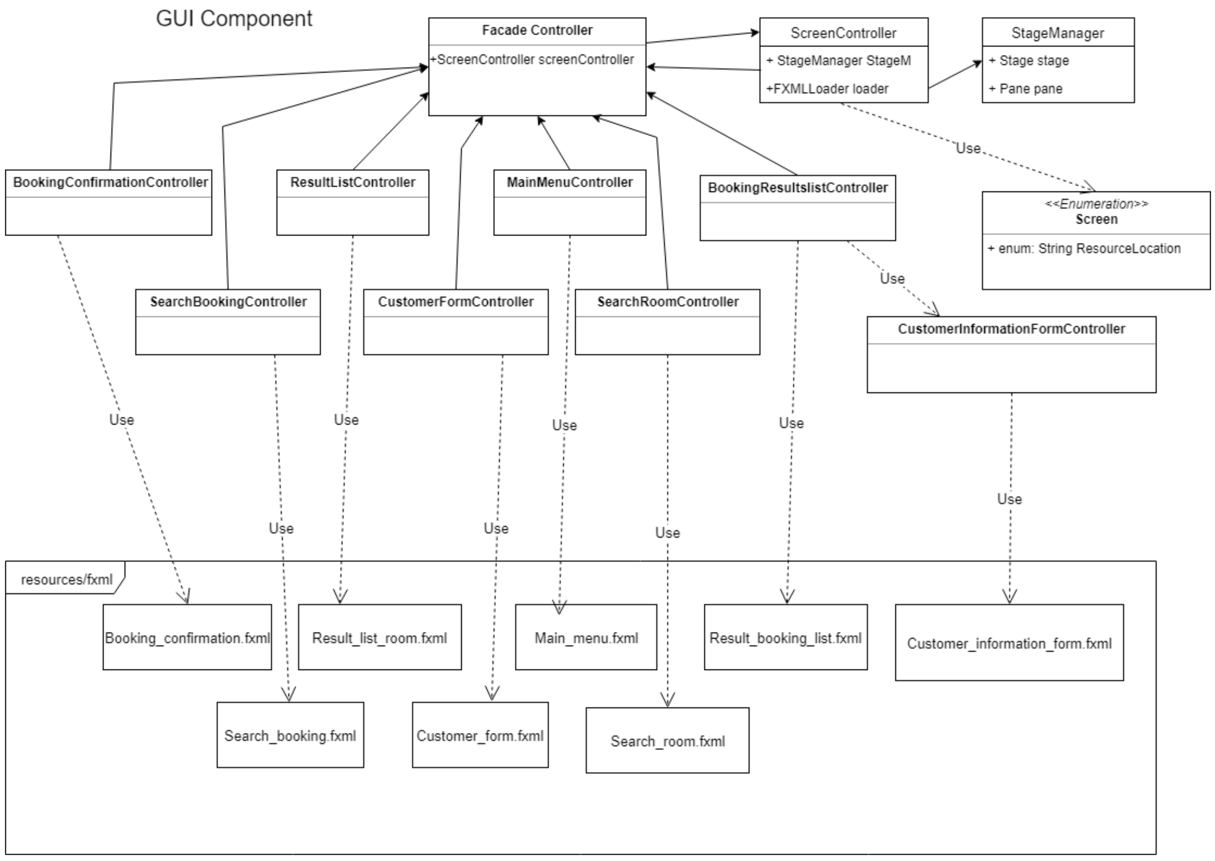


Fig. 3: the gui/controller class diagram

5.3.2 The Facade/Client model/Client network communication

As seen in this class diagram figure 4, FacadeController is a main point of communication between the classes. The classes varies in detail level, this is model on the level of importance. Variables and simple methods is not considered important for the design, but for example the FacadeController it’s very detailed since that is important for any one who wishes to develop new functionality to this software.We can see a few circular dependencies which is a result of the facade controller since most of the classes (except the model) need an facade controller to be able to communicate with the rest of the application.

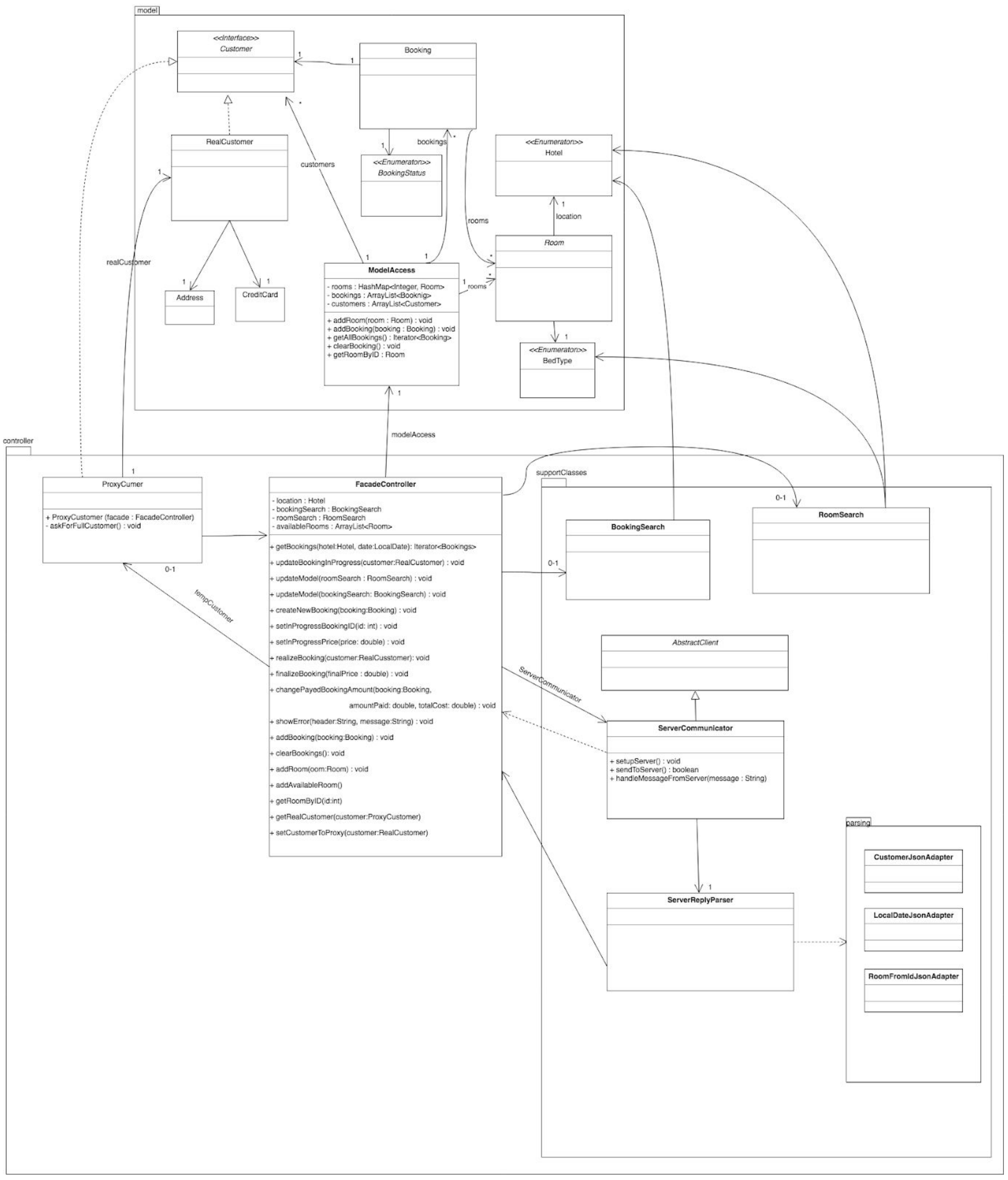

model
<<Interface>>
Customer
Booking
RealCustomer
<<Enumeraton>>
BookingStatus
<<Enumeraton>>
Hotel
Room
Address
CreditCard
ModelAccess
- rooms : HashMap<Integer, Room>
- bookings : ArrayList<Booknig>
- customers : ArrayList<Customer>
+ addRoom(room : Room) : void
+ addBooking(booking : Booking) : void
+ getAllBookings() : Iterator<Booking>
+ clearBooking() : void
+ getRoomByID : Room
<<Enumeraton>>
BedType
customers
bookings
rooms
location
realCustomer
modelAccess
controller
ProxyCumer
+ ProxyCustomer (facade : FacadeController)
- askForFullCustomer() : void
FacadeController
- location : Hotel
- bookingSearch : BookingSearch
- roomSearch : RoomSearch
- availableRooms : ArrayList<Room>
+ getBookings(hotel:Hotel, date:LocalDate): Iterator<Bookings>
+ updateBookingInProgress(customer:RealCustomer) : void
+ updateModel(roomSearch : RoomSearch) : void
+ updateModel(bookingSearch: BookingSearch) : void
+ createNewBooking(booking:Booking) : void
+ setInProgressBookingID(id: int) : void
+ setInProgressPrice(price: double) : void
+ realizeBooking(customer:RealCusstomer): void
+ finalizeBooking(finalPrice : double) : void
+ changePayedBookingAmount(booking:Booking,
amountPaid: double, totalCost: double) : void
+ showError(header:String, message:String) : void
+ addBooking(booking:Booking) : void
+ clearBookings(): void
+ addRoom(oom:Room) : void
+ addAvailableRoom()
+ getRoomByID(id:int)
+ getRealCustomer(customer:ProxyCustomer)
+ setCustomerToProxy(customer:RealCustomer)
tempCustomer
supportClasses
BookingSearch
RoomSearch
AbstractClient
ServerCommunicator
+ setupServer() : void
+ sendToServer() : boolean
+ handleMessageFromServer(message : String)
ServerReplyParser
parsing
CustomerJsonAdapter
LocalDateJsonAdapter
RoomFromIdJsonAdapter

Fig. 4 The class diagram of Facade Controller, client model and the client side network communication classes.

5.3.3 Server

Since the server consists of very few classes, they could all be presented in the same diagram (figure 5). The *HotelServer* class is the main class, and together with the *server.abstractServer* package it makes up the “Server network communication” component. The AbstractServer and ClientConnection classes makes up the actual server connection functionality, while HotelServer overrides some methods in AbstractServer to tailor it to communication with the DAO.

The SqlDAO class contains methods for gathering data or forming an insertion request, on a higher level. It then calls on the SqlQueries class for more specific actual calls to the database. The SqlDAO also formats the response to the client.

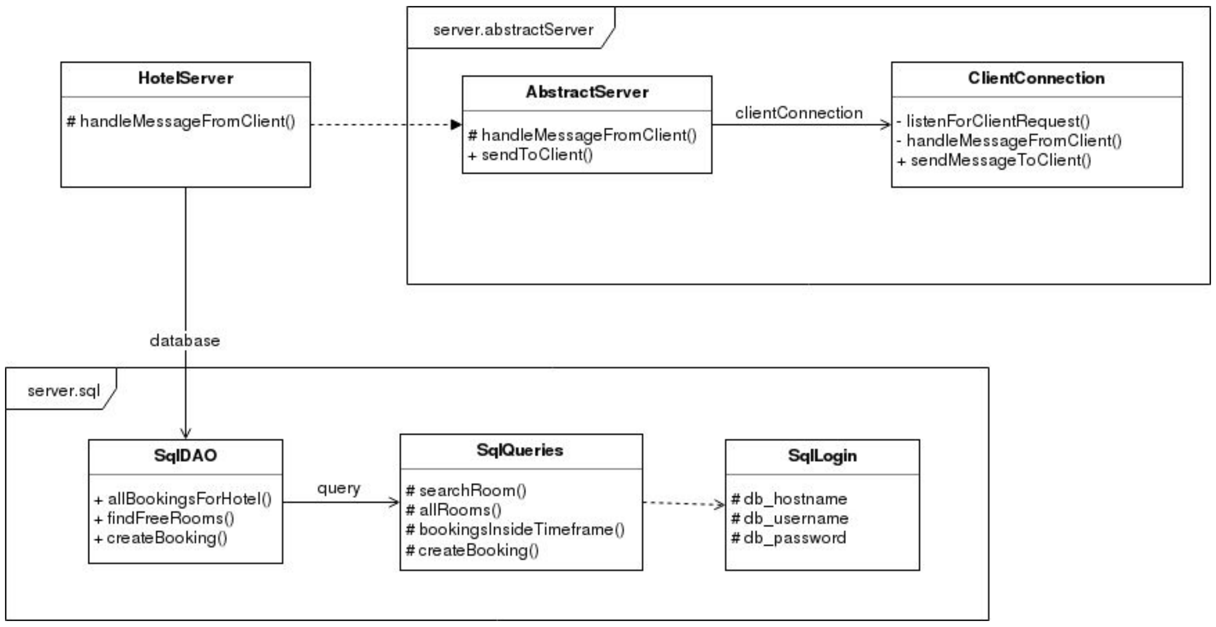


Fig 5. Class diagram of the server, and the packages it contains.

5.3.4 Database

The database is a relational database. It holds objects as “rows”, and attributes in “columns”, so that all instances creates a “table” (basically it’s a class definition, and all its instances, at once). You can ask for objects from a table, and filter them depending on their value in any of their columns.

The classes in our database (see Fig. 6) are the Customer class, detailing any guest that has once booked a room in the system. The Booking class represent a reservation of one or more rooms by a single customer. The Room class holds the information for a room. There is also the roomBinding table, which is more of a utility class. It makes it easier for one booking to reference several rooms.

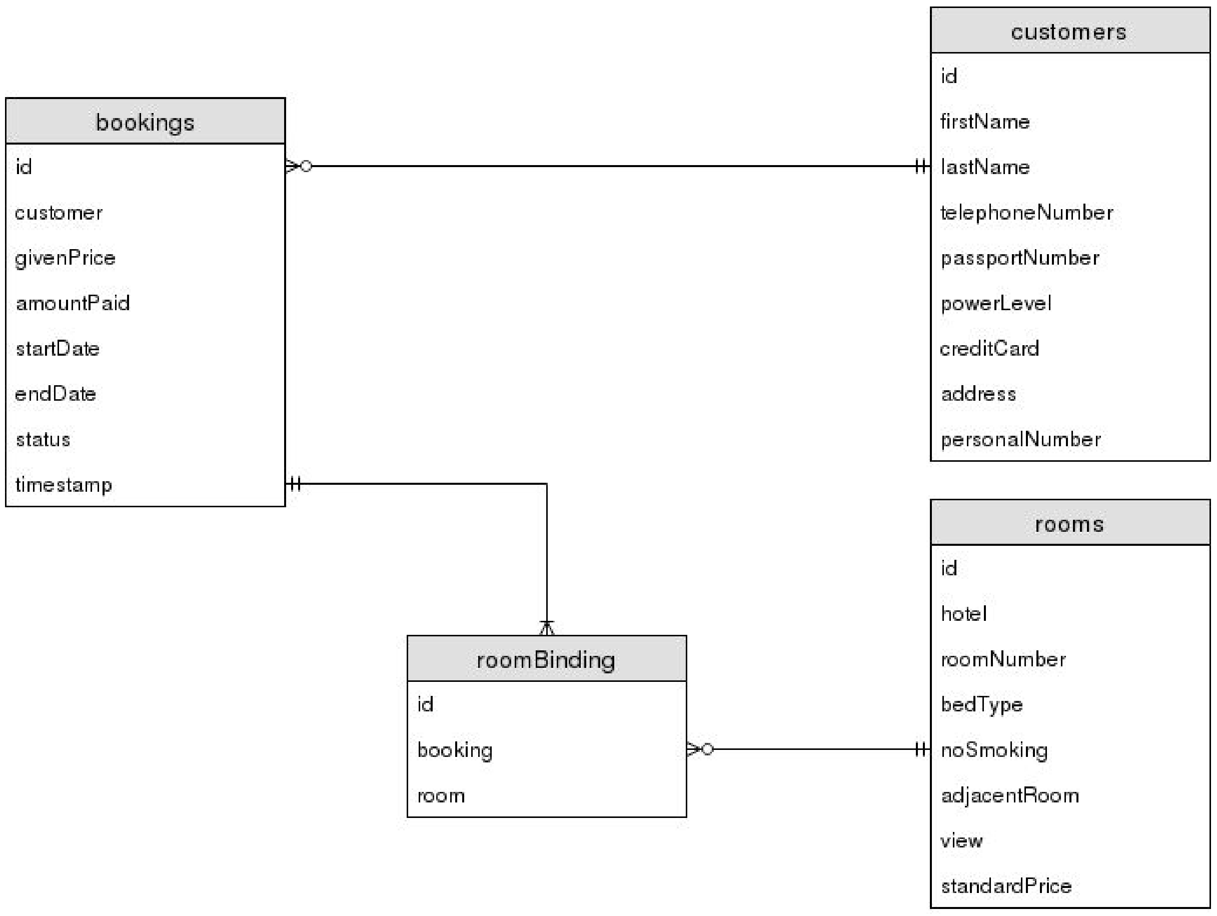


Fig 6: small ER diagram over the database.

5.4 Design Principles

Early in the project we discussed if any part of the software could be completely reusable. When we had decided for a Client-Server model it became obvious that the Client-Server communication could be completely reusable for any future client-server application that send a string through tcp connection. It could probably be an option to reuse any existing component for this, but after little researching we decided to create this from scratch to have full control. The client-server model in the literature “Object-Oriented Software Engineering” chapter 3 by T.C. Lethbridge and R. Laganière was used as a inspiration to get good reusability.

Overall we tried to keep good practice of understandable variable names and clear comments throughout the classes which helps for any future updates of this software.

5.5 Sequence Diagrams

All user use cases assume that you are in the main menu unless stated otherwise.

5.5.1 - Use Case 1 - Check In

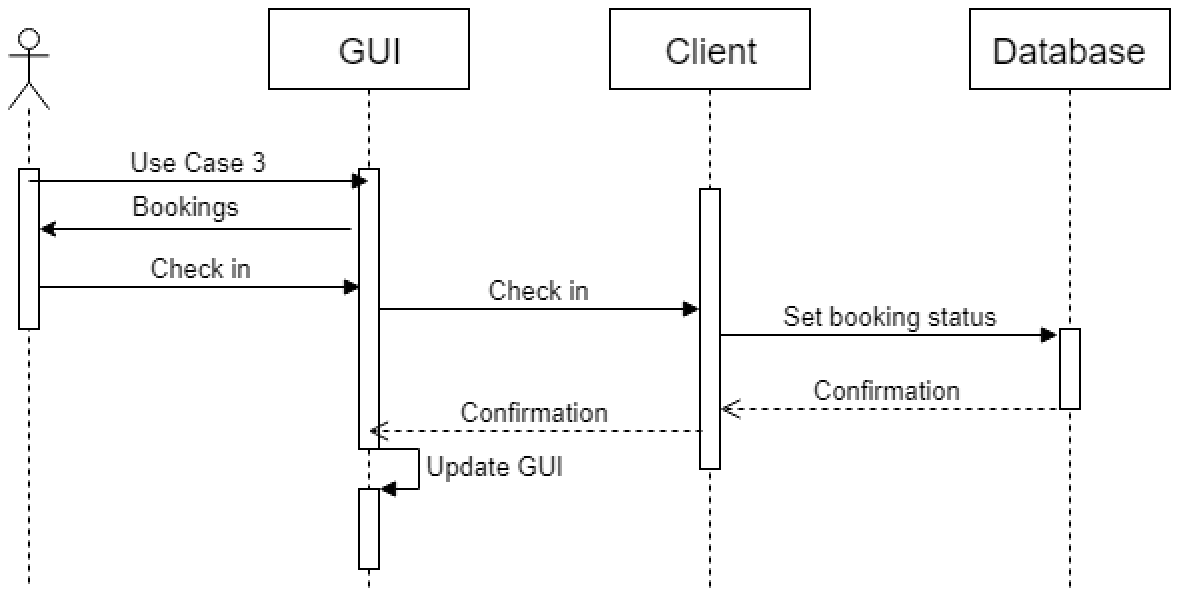
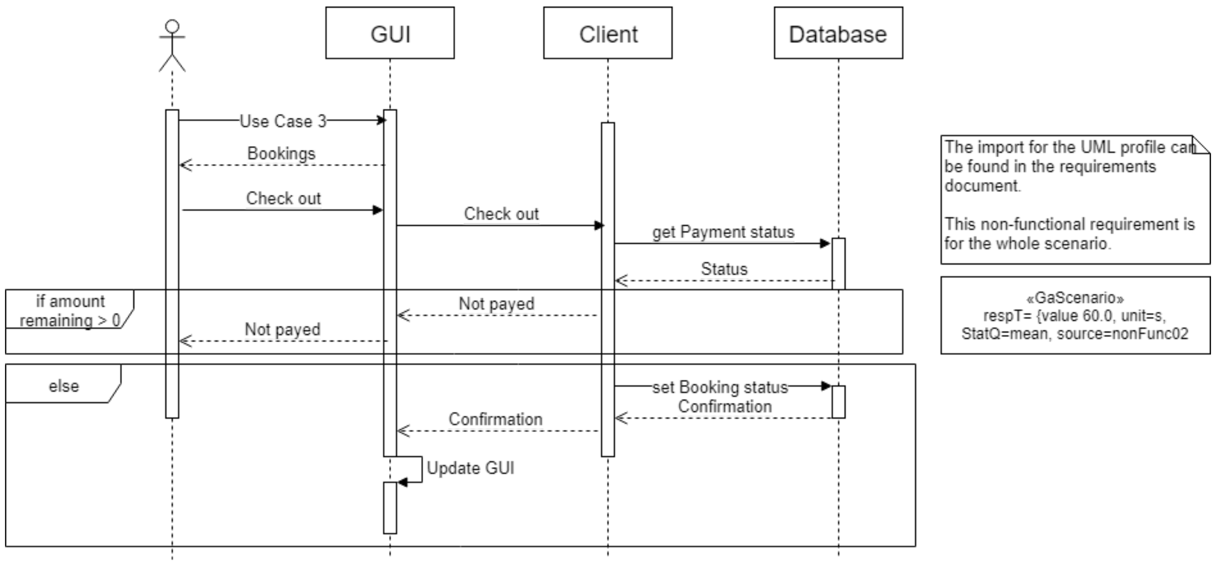


Fig. 7: Sequence diagram for use case 1 (check in).

As you can see in fig.7, this use case extends use case 3 (see figure 9). Once you get the booking results, you select one of the bookings and press “check in”. The client will then send a request to the database to update the booking status. The database will then change the status to “checked in” and will send a confirmation message back to the client. Once received, the client will update the model, and the GUI will update the view with the new information.

5.5.2 - Use Case 2 - Check out



GUI
Client
Database
Use Case 3
Bookings
Check out
Check out
get Payment status
Status
if amount remaining > 0
Not payed
Not payed
else
set Booking status
Confirmation
Confirmation
Update GUI
The import for the UML profile can be found in the requirements document.
This non-functional requirement is for the whole scenario.
«GaScenario» respT= {value 60.0, unit=s, StatQ=mean, source=nonFunc02

Fig. 8: Sequence diagram for use case 2 (check out).

As the previous use case, this (fig.8) is also an extension of use case 3 (see figure 9). This is very similar to use case 1 however, here we send a request to check the payment status of the selected booking. Once received, we control if it the booking has been paid or not. If not, then we send an error message to the GUI. If it is paid, we send a request to change the booking status to “checked out”. Once we get the confirmation, we update our model and also update the view with the new information. According to requirement NonFunc_02 , this process must take 60 seconds or less, which is also denoted by the marte profile.

5.5.3 - Use Case 3 - Find Reservation

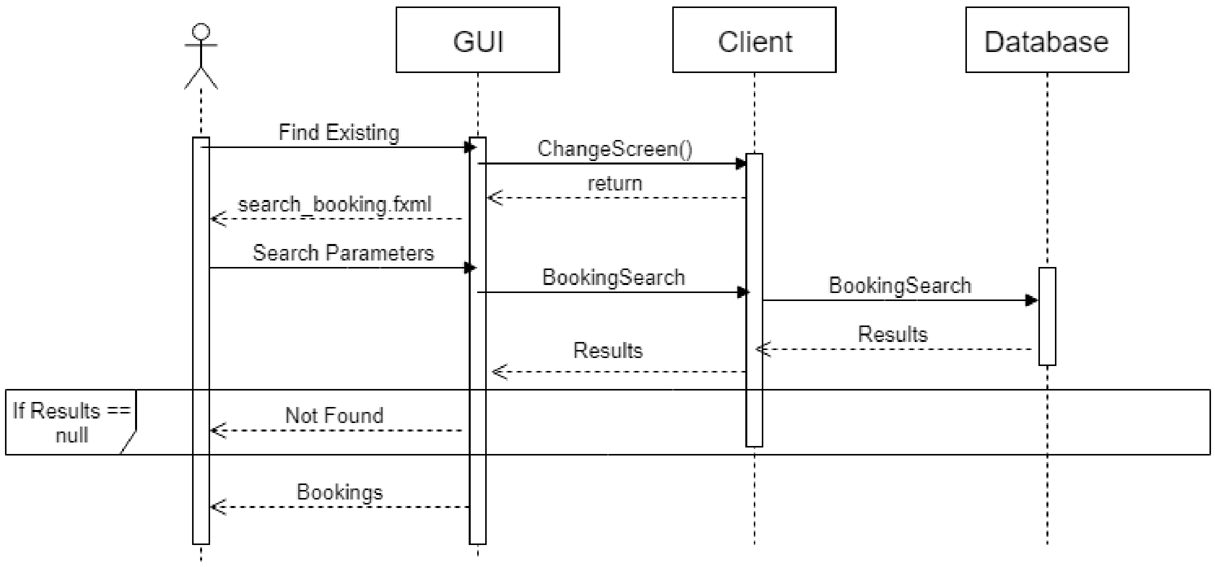
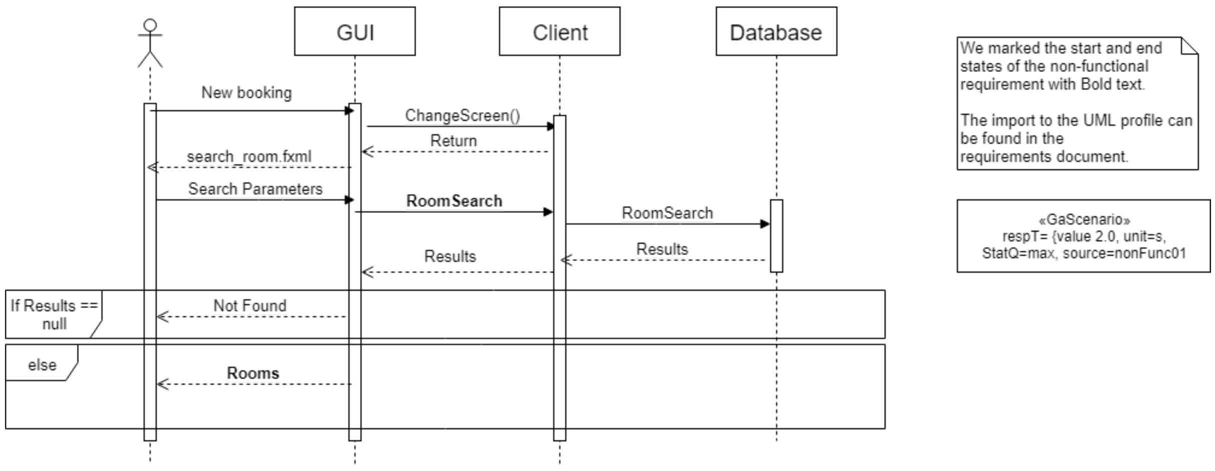


Fig. 9: Sequence diagram for use case 3 (find reservation).

This (see fig. 9 for the graphical representation) is the basic use case for finding an already existing booking. Starting from main menu, we press the “find existing” button. Then we send a request to change screen to the client, who then changes the screen to a form with search parameters. If you enter one or more parameters and press find booking, we create a BookingSearch object which we send to the client , who in turn parse the object into a server request. We then get the results back in an array. If the returned array does not have any entries, there was no booking found. Otherwise we change the screen to a result list and display the bookings matching the search parameters.

5.5.4 - Use Case 4 - Find Rooms



GUI
Client
Database
New booking
ChangeScreen()
Return
search_room.fxml
Search Parameters
RoomSearch
RoomSearch
Results
Results
If Results == null
Not Found
else
Rooms
We marked the start and end states of the non-functional requirement with Bold text.
The import to the UML profile can be found in the requirements document.
«GaScenario» respT= {value 2.0, unit=s, StatQ=max, source=nonFunc01

Fig. 10: Sequence diagram for use case 4 (find rooms).

This use case is the basic case for finding one or mutliple rooms. Figure 10 shows the flow of this use case as well as the alternative flow.
When pressing the “new booking” button we send a changeScreen request to the client , which in turn returns a form (similarly to use case 3). Once filled in, we create a RoomSearch object with the parameters entered, and send it to the client. The client then parses this object in to a server request, and we get an array of available rooms back. If the array is empty, we give an error message, otherwise we display a list of available rooms matching the search parameters given. According to the NonFunc_01 requirement, the searching of available rooms should take a maximum of 2 seconds. This denoted to the right and the start and end state of the non functional requirement is marked in bold text to increase clarity.

5.5.5 - Use Case 5 - Book Room / Rooms

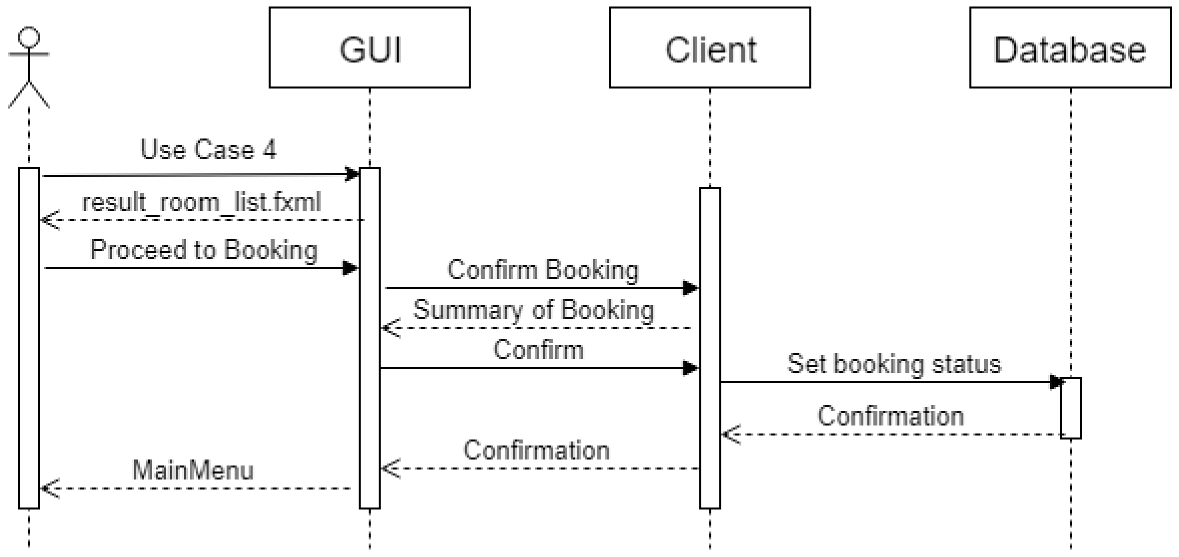


Fig. 11: Sequence diagram for use case 5 (book room/rooms).

This (see figure 11) is an extension to the previous use case. Once we have gotten a list of rooms and selected a room we wish to book, we press “proceed to booking”. Here we get a form where we enter the customers information. When pressing “confirm booking” we get a summary of the booking with the price, date, name etc. Once pressing confirm, we send the customers information to the client. The client then proceeds to send a server request to create this booking. Once a confirmation is received, the user is then returned to the main menu.

5.5.2 - Use Case 6 - Cancel Reservation

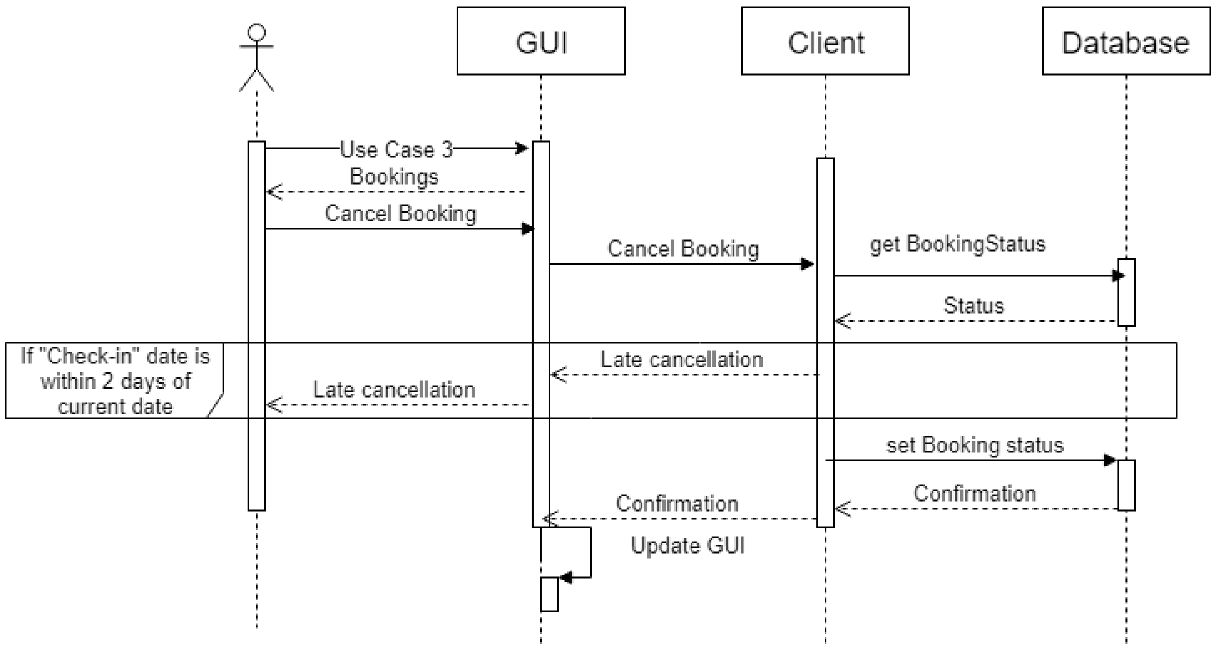


Fig. 12: Sequence diagram for use case 6 (cancel reservation).

This use case (figure 12) is an extension to use case 3 (figure 9). Once the booking has been received and you press the “cancel booking button”, we send a server request to check the booking status and the “check in” date. If the date for checking in is in 2 days or less, it will be regarded as a late cancellation and the user will be notified. Then we send a request to the client, who in turn creates a server request to change the booking status. The database will send a confirmation once the status has been updated. Once received, the GUI will update with the new information.

5.5.2 - Use Case 7 - Pay Booking

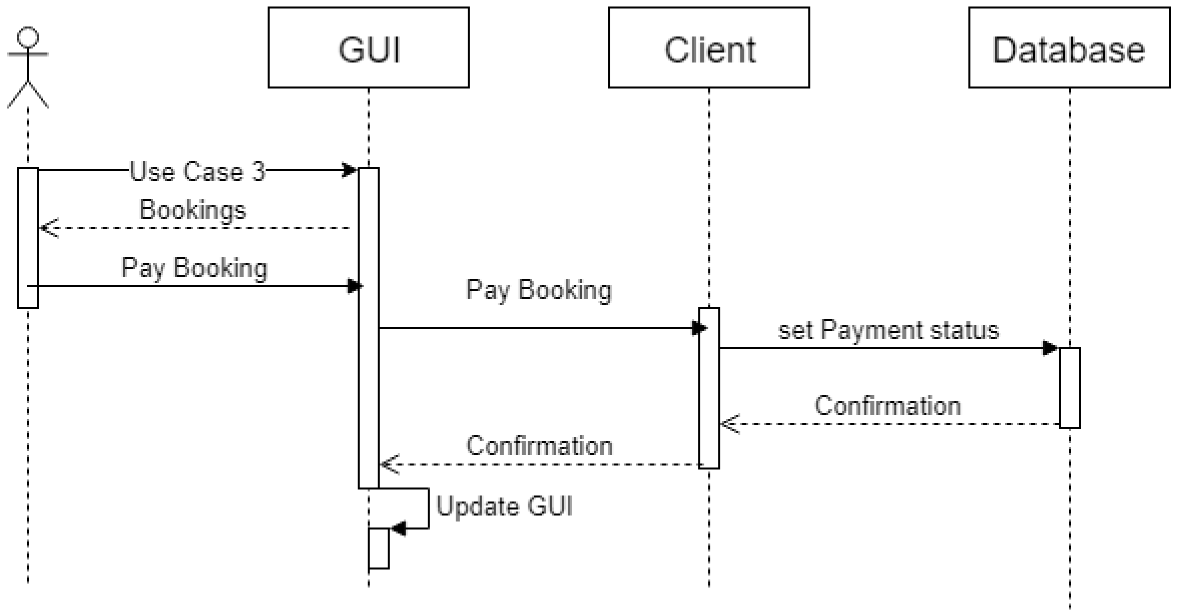


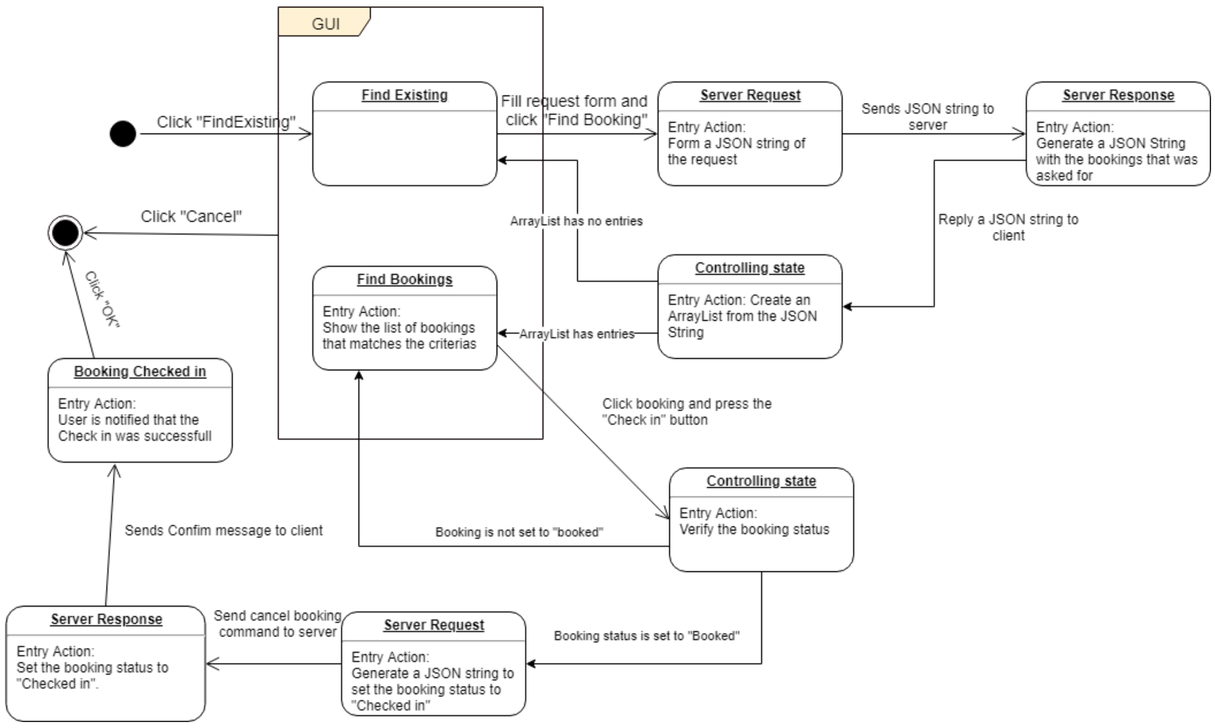
Fig. 13: Sequence diagram for use case 7 (pay booking).

This is use case (figure 13) also an extension of use case 3 (see figure 9). Once we select a booking and press “Pay booking”, the client sends a request to set the payment status. Once received, the server will send a confirmation. When the client gets the confirmation, the GUI will be updated with the new information.

5.6 State-machine Diagrams

All of these diagrams assume that you have just started up the application or are in the main menu. All of these diagrams have a “GUI” frame, which is just to increase and to easier model the graphical user interface. For every state in the “GUI” frame, there is a fxml or screen for that respective state.

5.6.1 - Use Case 1 - Check in



GUI
Click "FindExisting"
Find Existing
Fill request form and click "Find Booking"
Server Request
Entry Action: Form a JSON string of the request
Sends JSON string to server
Server Response
Entry Action: Generate a JSON String with the bookings that was asked for
Reply a JSON string to client
Controlling state
Entry Action: Create an ArrayList from the JSON String
ArrayList has no entries
ArrayList has entries
Find Bookings
Entry Action: Show the list of bookings that matches the criterias
Click "Cancel"
Click "OK"
Booking Checked in
Entry Action: User is notified that the Check in was successfull
Click booking and press the "Check in" button
Controlling state
Entry Action: Verify the booking status
Booking is not set to "booked"
Booking status is set to "Booked"
Server Request
Entry Action: Generate a JSON string to set the booking status to "Checked in"
Send cancel booking command to server
Server Response
Entry Action: Set the booking status to "Checked in".
Sends Confim message to client

Fig. 14: State machine diagram for use case 1 (check in).

To cover all the states of use case 1 (figure 14), we have to include use case 3 (see figure 16) as use case 1 is an extension of use case 3. The states in the GUI frame is where we change the screen or graphical user interface to another screen or view. All of the states in the GUI frame can reach the end state by pressing cancel.

5.6.2 - Use Case 2 - Check out

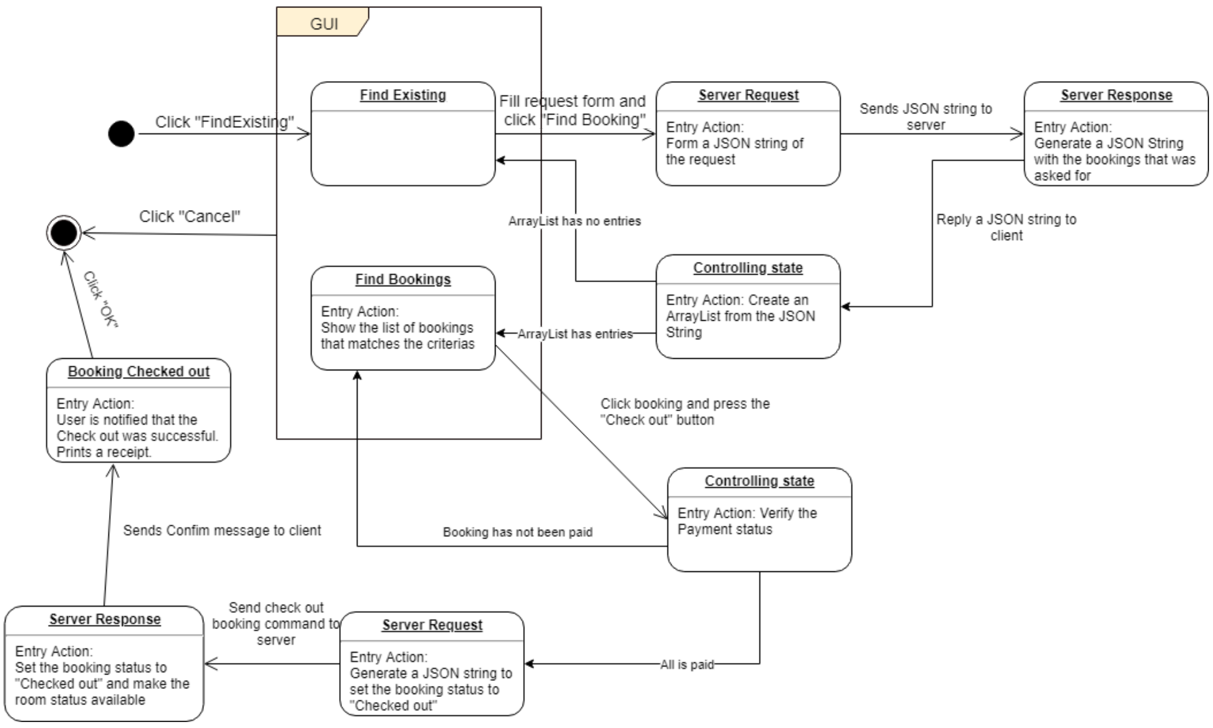

GUI
Click "FindExisting"
Find Existing
Fill request form and click "Find Booking"
Server Request
Entry Action: Form a JSON string of the request
Sends JSON string to server
Server Response
Entry Action: Generate a JSON String with the bookings that was asked for
Reply a JSON string to client
Click "Cancel"
ArrayList has no entries
Controlling state
Entry Action: Create an ArrayList from the JSON String
ArrayList has entries
Find Bookings
Entry Action: Show the list of bookings that matches the criterias
Click "OK"
Booking Checked out
Entry Action: User is notified that the Check out was successful. Prints a receipt.
Click booking and press the "Check out" button
Controlling state
Entry Action: Verify the Payment status
Booking has not been paid
Sends Confim message to client
All is paid
Server Request
Entry Action: Generate a JSON string to set the booking status to "Checked out"
Send check out booking command to server
Server Response
Entry Action: Set the booking status to "Checked out" and make the room status available

Fig. 15: State machine diagram for use case 2 (check out).

Just like use case 1 (figure 14), we also need use case 3 (see figure 16) in order to model all the states in this use case (figure 15). The big difference between this and the previous diagram is that once the “check out” button has been pressed, we have a controlling state to control the payment status of the booking. If the payment status is paid, we then enter the server request state where we send the request to change the booking status from “checked in” to “checked out”.

5.6.3 - Use Case 3 - Find Reservation

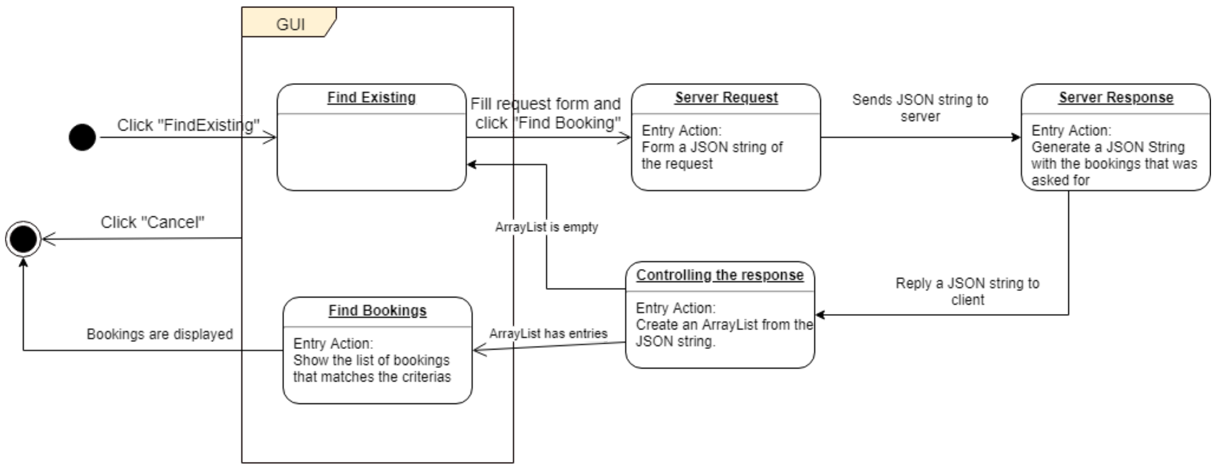

GUI
Click "FindExisting"
Find Existing
Fill request form and click "Find Booking"
Server Request
Entry Action:
Form a JSON string of the request
Sends JSON string to server
Server Response
Entry Action:
Generate a JSON String with the bookings that was asked for
Reply a JSON string to client
Controlling the response
Entry Action:
Create an ArrayList from the JSON string.
ArrayList is empty
ArrayList has entries
Find Bookings
Entry Action:
Show the list of bookings that matches the criterias
Bookings are displayed
Click "Cancel"

Fig. 16: State machine diagram for use case 3 (find reservation).

This use case (figure 16) is the base case for use case 1,2,6 and 7. This is simply to find an already existing booking by going to the find existing state. This is done by the “find existing” button on the main menu. This will load a search form and when one or more parameters has been entered and the user presses “Find booking”, we enter the server request state where we request to find the booking. Once the server has responded, we create the array of bookings matching the parameters. If the array has no entries, there was no booking found, and the user is returned to the “Find Existing” state. If it has entries, we enter the “Find Bookings” state, where we display the entries in a list.

5.6.4 - Use Case 4 - Find Rooms

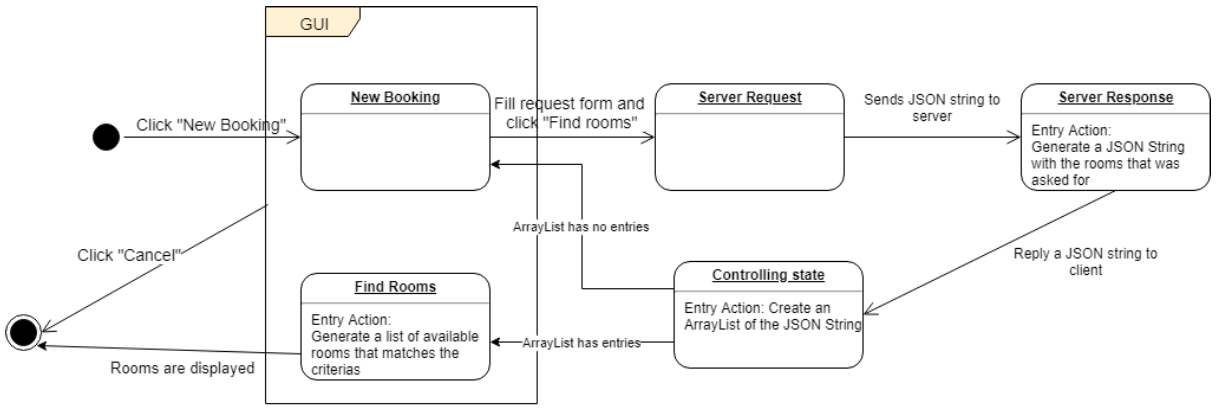

GUI
Click "New Booking"
New Booking
Fill request form and click "Find rooms"
Server Request
Sends JSON string to server
Server Response
Entry Action: Generate a JSON String with the rooms that was asked for
Reply a JSON string to client
Controlling state
Entry Action: Create an ArrayList of the JSON String
ArrayList has no entries
ArrayList has entries
Find Rooms
Entry Action: Generate a list of available rooms that matches the criterias
Click "Cancel"
Rooms are displayed

Fig. 17: State machine diagram for use case 4 (find rooms).

This use case (figure 17) is similar to use case 3 (fig 16) in many ways, and also serves as a base case for use case 5 (figure 18). We press “New Booking” and get a form with different parameters. Once the parameters have been entered, we form a request and send it to the server. Once we get the reply, we create an array of rooms. If the array has no entries, we send an error message saying “no rooms found” and send the user back to the “New Booking” state. If the array has entries we change screen and enter the “Find Rooms” state where we display the rooms in the array.

5.6.5 - Use Case 5 - Book Room / Rooms

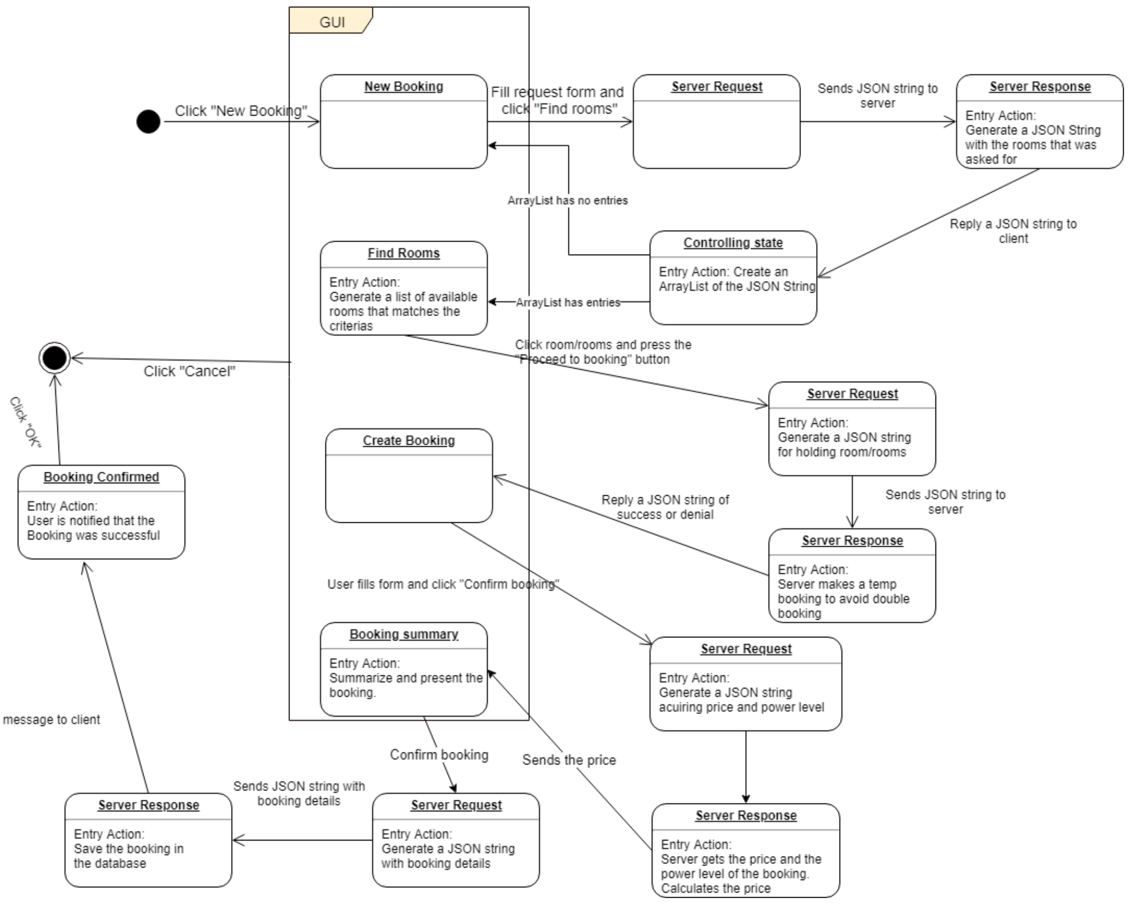
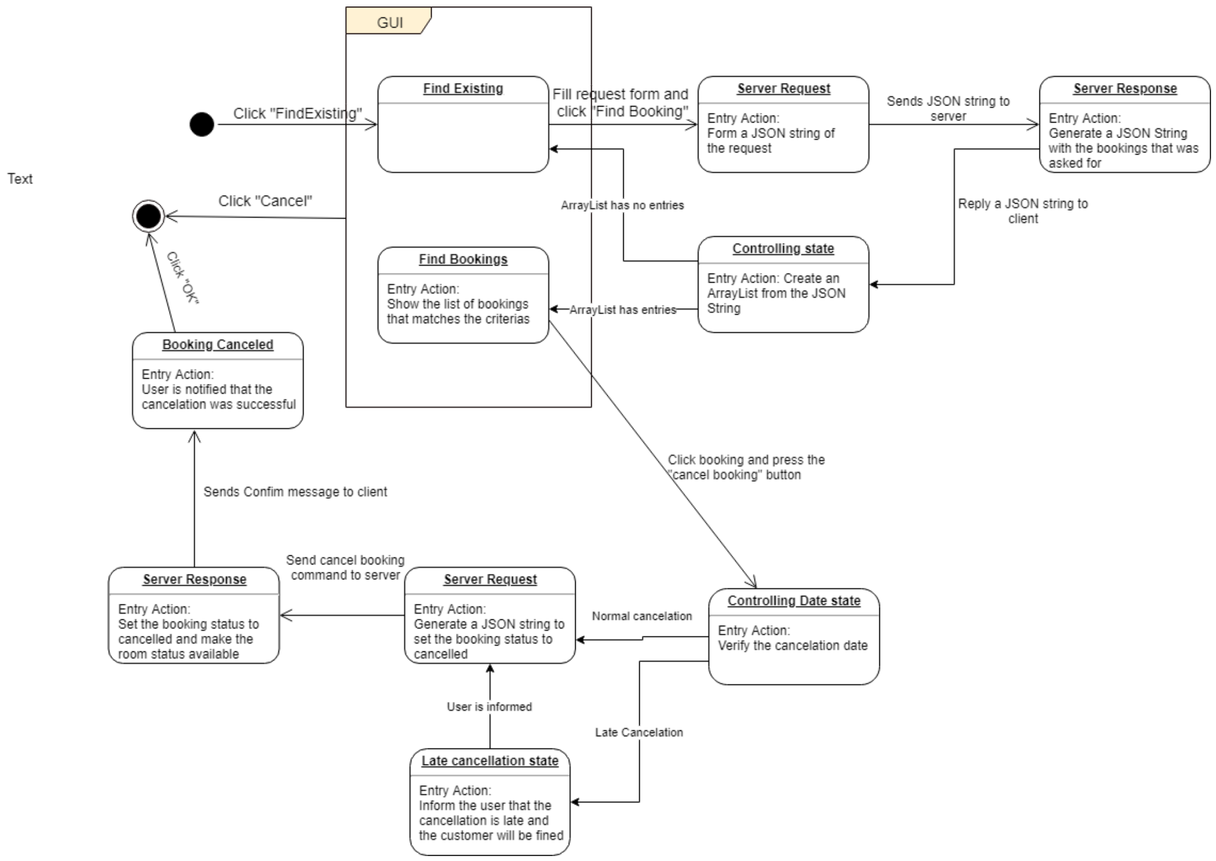

GUI
Click "New Booking"
New Booking
Fill request form and click "Find rooms"
Server Request
Sends JSON string to server
Server Response
Entry Action: Generate a JSON String with the rooms that was asked for
ArrayList has no entries
Reply a JSON string to client
Find Rooms
Entry Action: Generate a list of available rooms that matches the criterias
ArrayList has entries
Controlling state
Entry Action: Create an ArrayList of the JSON String
Click "Cancel"
Click room/rooms and press the "Proceed to booking" button
Click "OK"
Server Request
Entry Action: Generate a JSON string for holding room/rooms
Create Booking
Booking Confirmed
Entry Action: User is notified that the Booking was successful
Reply a JSON string of success or denial
Sends JSON string to server
Server Response
Entry Action: Server makes a temp booking to avoid double booking
User fills form and click "Confirm booking"
Booking summary
Entry Action: Summarize and present the booking.
Server Request
Entry Action: Generate a JSON string acuiring price and power level
Sends Confim message to client
Confirm booking
Sends the price
Sends JSON string with booking details
Server Response
Entry Action: Save the booking in the database
Server Request
Entry Action: Generate a JSON string with booking details
Server Response
Entry Action: Server gets the price and the power level of the booking. Calculates the price

Fig. 18: State machine diagram for use case 5 (book room/rooms).

This use case (figure 18) is an extension of Use case 4 (figure 17), and once we get the rooms displayed, we select one and press “proceed with booking”. Here we send a request to hold this room, which means that no one else can attempt to book this room until we released this room. After we get a confirmation that it is held, user is then given a form to fill in the customers information. Once filled and user presses “confirm booking” we send a server request in order to obtain the complete price with regards to power level. Once we have the price, we change to another screen where we summarize the booking, displaying the customer information, booking dates etc. When user presses “Confirm booking”, we send a request to the server to create and save this booking. Once we get a confirmation, we display a message saying that the booking has been successfully created.

5.6.6 - Use Case 6 - Cancel Reservation



GUI
Text
Click "FindExisting"
Find Existing
Fill request form and click "Find Booking"
Server Request
Entry Action: Form a JSON string of the request
Sends JSON string to server
Server Response
Entry Action: Generate a JSON String with the bookings that was asked for
Reply a JSON string to client
Click "Cancel"
ArrayList has no entries
Controlling state
Entry Action: Create an ArrayList from the JSON String
ArrayList has entries
Find Bookings
Entry Action: Show the list of bookings that matches the criterias
Click "OK"
Booking Canceled
Entry Action: User is notified that the cancelation was successful
Sends Confim message to client
Click booking and press the "cancel booking" button
Send cancel booking command to server
Server Response
Entry Action: Set the booking status to cancelled and make the room status available
Server Request
Entry Action: Generate a JSON string to set the booking status to cancelled
Normal cancelation
Controlling Date state
Entry Action: Verify the cancelation date
User is informed
Late Cancelation
Late cancellation state
Entry Action: Inform the user that the cancellation is late and the customer will be fined

Fig. 19: State machine diagram for use case 6 (cancel reservation).

This use case (figure 19) is an extension of use case 3 (figure 16), and the addition to use case 3 is that once we receive the booking we requested, selects it and presses “Cancel Booking”, we enter a controlling state. Here we verify the cancellation date (the current date) and the “check in” date. If they are 2 days or fewer apart, the user will be notified that this is a “late cancellation”, and the user will be fined for a part of the costs. Once user has been informed, we will create a server request to change the booking status to “canceled”. We will reach this state even if the “check in” date is farther than 2 days apart, but the customer will only be fined for a late cancellation. After the request has been created, we send it to the server, which in turn updates the database and gives us a confirmation message. Once received, the user will receive a message saying “cancellation was successful”.

5.6.7 - Use Case 7 - Pay Reservation

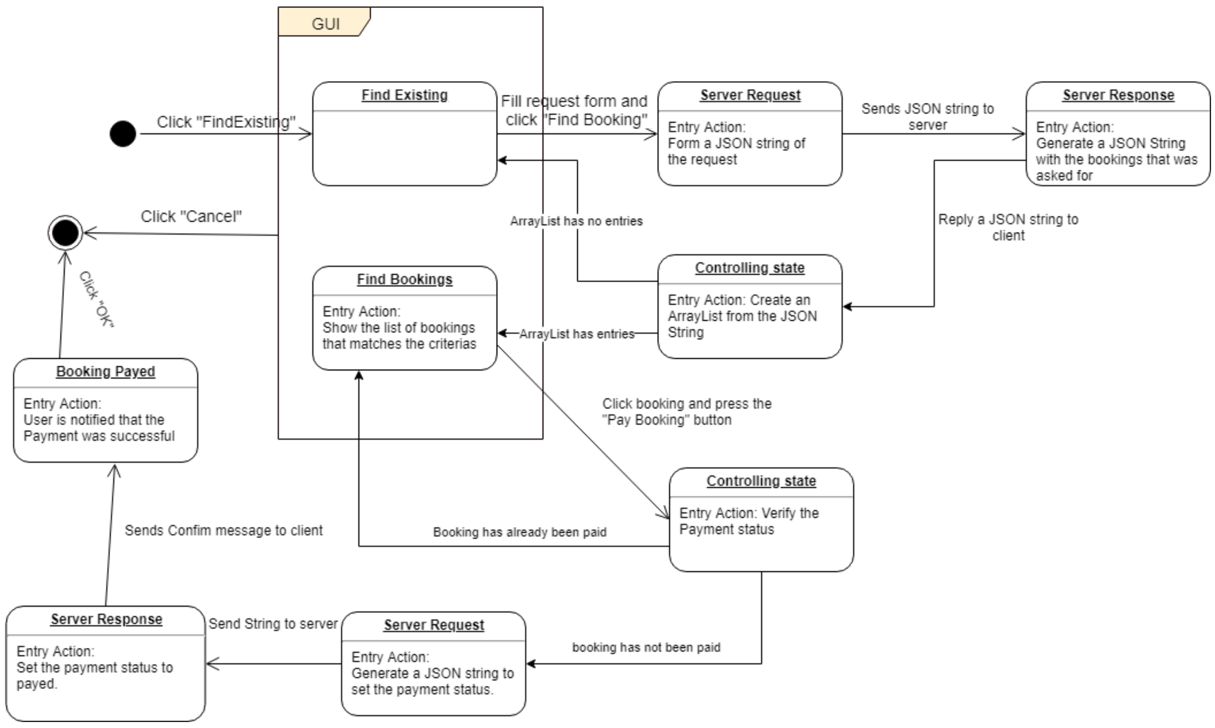

GUI
Click "FindExisting"
Find Existing
Fill request form and click "Find Booking"
Server Request
Entry Action: Form a JSON string of the request
Sends JSON string to server
Server Response
Entry Action: Generate a JSON String with the bookings that was asked for
Reply a JSON string to client
Click "Cancel"
ArrayList has no entries
Controlling state
Entry Action: Create an ArrayList from the JSON String
Find Bookings
Entry Action: Show the list of bookings that matches the criterias
ArrayList has entries
Click "OK"
Booking Payed
Entry Action: User is notified that the Payment was successful
Click booking and press the "Pay Booking" button
Controlling state
Entry Action: Verify the Payment status
Sends Confim message to client
Booking has already been paid
Server Response
Entry Action: Set the payment status to payed.
Send String to server
Server Request
Entry Action: Generate a JSON string to set the payment status.
booking has not been paid

Fig. 20: State machine diagram for use case 7 (pay reservation).

As this use case (figure 20) is an extension of the third use case (figure 16), we follow all the previous states. Once the booking has been selected and user presses “Pay Booking”, we enter a controlling state where we check if the booking has already been paid. If it’s already paid, we return to the “find bookings” state. Otherwise we generate a server request to change the payment status and send it to the server. Once the database has been updated, we send a confirmation message to the client. When the message has been received, the user is notified that the payment was successful.

6. Performance engineering

This section will shortly describe the performance engineering of the Linnaeus Hotel Server.

6.1 The performance engineering

We have a time of 4 ms in average on the application server, and 15 ms for the database. We also have a population of two, which is the number of front desks for the Linnaeus Hotel chain. Finally we have a requirement that it must never take longer than two seconds to find out if there is available rooms for any date. Since the application server is using a synchronized call the server can only handle one client at a time, and if two arrives at the same time one of them has to wait until the other get served, and then for the time it takes for the server to handle the request. We also have a requirement(NonFunc_01), which states that it must never take more than 2 seconds for a request to find available rooms when creating a booking. By study the diagram fig 21 we can see that for a single call the longest average time is in total 29 ms. However such times can change quite drastically during higher loads. So to evaluate this we will use the formula: μ = jobs/time = 1/0.015 = 66 jobs per second which is our teoretiska maximum throughput.And then we need to check how many request we must be able to handle and we use:
λ = arrival rate --> (5000/2)+((10+1)/2) = 2004. 5ms delay/request -----> λ = 0.4988... requests / sekund, which concludes that the system will have no problem handle the requirements.

6.2 The diagram

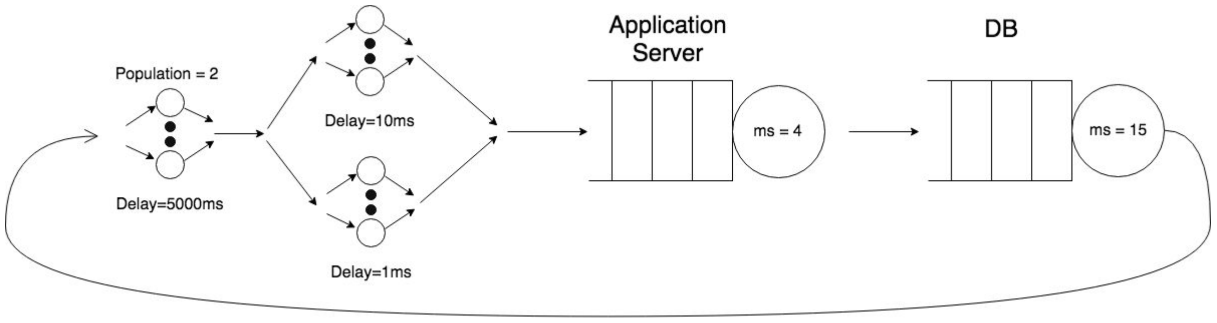


Fig. 21: This diagram explains the server calls and our measured average times. The 2 different paths is that one client is on the same network as the server where the other one will have a 9 ms longer network connection. One note about the Server and DB times is that the database isn’t enough populated to make us sure that the times won't raise over time, however these tests is is run on a raspberry pi version 3, so there is definitely much room for upgrading the server.

6.3 Conclusion

Our calculations show that in the current state the server has no problem handle requirements asked for. This is not a large surprise, as the relational databases are known for their speed, and our model is relatively simple. However, the tests we've done could be considered best case, because the database was more or less empty at the time (all the rooms were present, but only a handful of bookings and customers were added). If the hotel had been running for quite some time, the database would be in another state. We recommend filling the database with some random data, so that the server has some data to chew through, to better approximate a real life scenario.

Another note about the server in these measurements is the hardware used - a Raspberry PI 3, which offers a 1.2 GHz processor and 1 GB of RAM. These are very low specs, especially for a server used in a live environment. Our prediction is that even with a largely populated database, all requirements would still be met.