# 

# 

2DV603 - Design

By Johan Andersson (ja222um), David Larsson (dl222is) & Fredrik Norrman (fn222ep)

*Version 1*

###### 

[**1 Introduction**](#_ghsodkcyp2c) **2**

[1.1 Purpose](#_howtb1tznf7n) 2

[1.2 Requirements implemented by this design](#_fj7x8updxbff) 2

[1.3 General priorities](#_4bugagemq797) 2

[1.4 Outline of the design](#_oui6i3vh6h3z) 3

[1.4.1 Rewriting the system](#_73jvawaomx0e) 3

[1.4.2 System structure](#_adozov3tlqqz) 3

[**2 Major design issues**](#_16wb2qet4ygj) **4**

[2.1 Legacy design or not](#_pycmjxr3jrev) 4

[2.2 Graphics library](#_x5z9u5ue2d2) 4

[2.3 Database](#_5y2wexv1ndj2) 4

[2.3.1 E/R Diagram](#_75zl8f3eaj19) 5

[2.3.2 Database Schema](#_vpqcrm4dx7d6) 5

[2.4 Refactoring and design decisions](#_iy3hz9l9fyed) 6

[2.4.1 Multi-threading](#_e2lrwut5ct4) 6

[2.4.2 Reusability](#_bemdkt7c2qv8) 6

[2.4.3 Design principles](#_17w9oxqy3vwe) 6

[2.4.3.1 Defensive programming](#_qdds0b8pgans) 6

[2.4.3.2 Divide and conquer](#_2407dez45m4x) 7

[2.4.3.3 Libraries](#_tisocbyp32b4) 7

[**3 Design details**](#_m6syiw1vralf) **8**

[3.1 Architecture](#_hhi2zip2ui3) 8

[3.1.1 Component Diagram](#_88vsqhqc4v0k) 8

[3.1.2 Components](#_rweblxljtttc) 8

[3.1.2.1 Application](#_xya6mwokkth) 8

[3.1.2.2 Model](#_dywr2tlsjq29) 8

[3.1.2.3 View](#_x2c4jvjnxg7d) 8

[3.1.2.4 Controller](#_j4vbholey7hy) 8

[3.1.3 Architectural Pattern](#_4wgcy345gd3b) 9

[3.1.4 Deployment](#_x72lh9mo16k9) 9

[3.2 Modelling](#_y95oohv0fkhc) 10

[3.2.1 Class Diagram](#_q6ustlpf38su) 10

[3.2.2 Sequence Diagrams](#_34shwo84vhku) 11

[3.2.2.1 List All Guest](#_953myvjh9gdu) 11

[3.2.2.2 Add Guest](#_sk5cu2x4k2qo) 12

[3.2.2.3 Search for Guest](#_4jr8tjljuaj) 13

[3.2.2.4 Update Guest](#_9ume7rco11nv) 14

[3.2.2.5 Make reservation](#_1hemni712gi6) 15

[3.2.2.6 Cancel reservation](#_uuqfjiltqhvz) 16

[3.2.2.7 Check in](#_2khrqikubvw3) 17

[3.2.2.8 Check out](#_u0l9khc8gjov) 18

# 1 Introduction

## 1.1 Purpose

This document will be going through the general design of a application that is created for a hotel front-desk, specifically for the Linnaeus Hotel stationed in Kalmar and Växjö. We will go through the design process used for the building of the application, the architecture of the application and the flow of the application, with the use of diagrams. The general idea is that by reading this document, you will be able to understand how the system is built and be able to understand the application and its architecture as good as possible.

## 1.2 Requirements implemented by this design

Most of these requirements are covered by a sequence diagram that explains the flow of the implementation of the requirement. But all of the requirements have been designed and implemented.

* F1.1 Book a room
* F1.2 Book specific room
* F2.1 Check availability
* F2.2 Check specific room
* F3.1 Cancel booking
* F4.1 Check in
* F4.2 Check out
* F5.1 Save customer information
* F6.1 List customers
* F6.2 Search customers
* F7.1 Print bill

These requirements were implemented with having the quality requirements being fulfilled in mind also.

## 1.3 General priorities

When we started the design process it was important for us to make the outlines of the program so that we would be able to see the program as a whole without it being complete. That way we would know right away if we missed anything. So we prioritized creating the skeleton of the application by creating the UI using JavaFX. By doing so we got a good knowledge of what had to be done next. It gave us sort of a checklist of what had to be done. Since most of the buttons and design was there rather quickly, the things that was left was the functionality for those buttons.

When the UI was “completed”, we started to look at ways to extract data from our database, as well as manipulate this data to make it readable and import it into a table within the UI. At first dummy functions were used for this, to console the output before we put it in an actual table. This was important to start off with since it gave us an insight in the program and how we should get and put data from it.

When this was completed the path onward was straight. We checked items off one by one from the requirements specification and got the basics down of all requirements. So the basic functionality was there for most of the program. And then we focused on perfecting this functions. Looking at efficiency in both time and beauty.

## 1.4 Outline of the design

### 1.4.1 Rewriting the system

At the start of the design process we started out by looking at the legacy system, since why shouldn’t we use that as a guideline. But when we got into the code and the general design of that system, we were not only displeased with it, but also quite scared at the thought of trying to understand such poorly written code. The system was built up using Swing and no architectural pattern was used, such as MVC. So all the files were in the same package and there was no clear structure of it. We gave it some time and tried to understand the code, but it was confusing us more than it did us good. So we came to the decision to rewrite the entire program from scratch.

And the motivation to that is what was mentioned previously, as well as the fact that we felt that we could make a better program from scratch rather than re-engineering it. Since the time it would take to grasp the structure of the code we would be halfway done with an entire new program.

### 1.4.2 System structure

We built up the system using a Model-View-Controller architecture. To see what parts work together, refer to the class diagram (3.2.1).

We tried to make the structure of the system as simple and clear as possible. That way it would be easy to refactor and improve for the future. But also very simple to understand.

We use a main controller that handles the first window you see when you run the program and from that controllers, multiple other controllers are derived and used for popup-windows.

All the controllers are linked with its own fxml-file in the view-package. And the specific controller for the view handles all the functionality for that window. Such as getting and setting information for and from the view. The code itself is very self-explanatory and is easy to understand if need-be.

# 

# 2 Major design issues

## 2.1 Legacy design or not

When we started to design the system, we started off with having to decide if we should refactor, re-use or re-write with using the old system as a base. And to decide how to do this, we simply analyzed the legacy code and made the decision to rewrite. This was the first issue we had to solve, since if we would’ve decided to refactor we would inherit the previous systems problems and design issues.

## 2.2 Graphics library

A choice we had to do was what graphics library to use. Although this choice was pretty clear, I feel like it is worth mentioning. There are a few graphics libraries to choose from when designing apps in Java. For example the legacy system used Swing and then there is SWT, AWT. But we decided to go with JavaFX for two reasons. The other systems are deprecated and Oracle does not offer support for them anymore. And JavaFX uses a MVC-pattern and that is what we decided to use to have clear structure in the program. JavaFX use fxml-files as the “View” and every fxml-file have an associated “Controller” and all you have to do is design the model-structure, the rest is given to you by the library. Knowing this, there was no other choice.

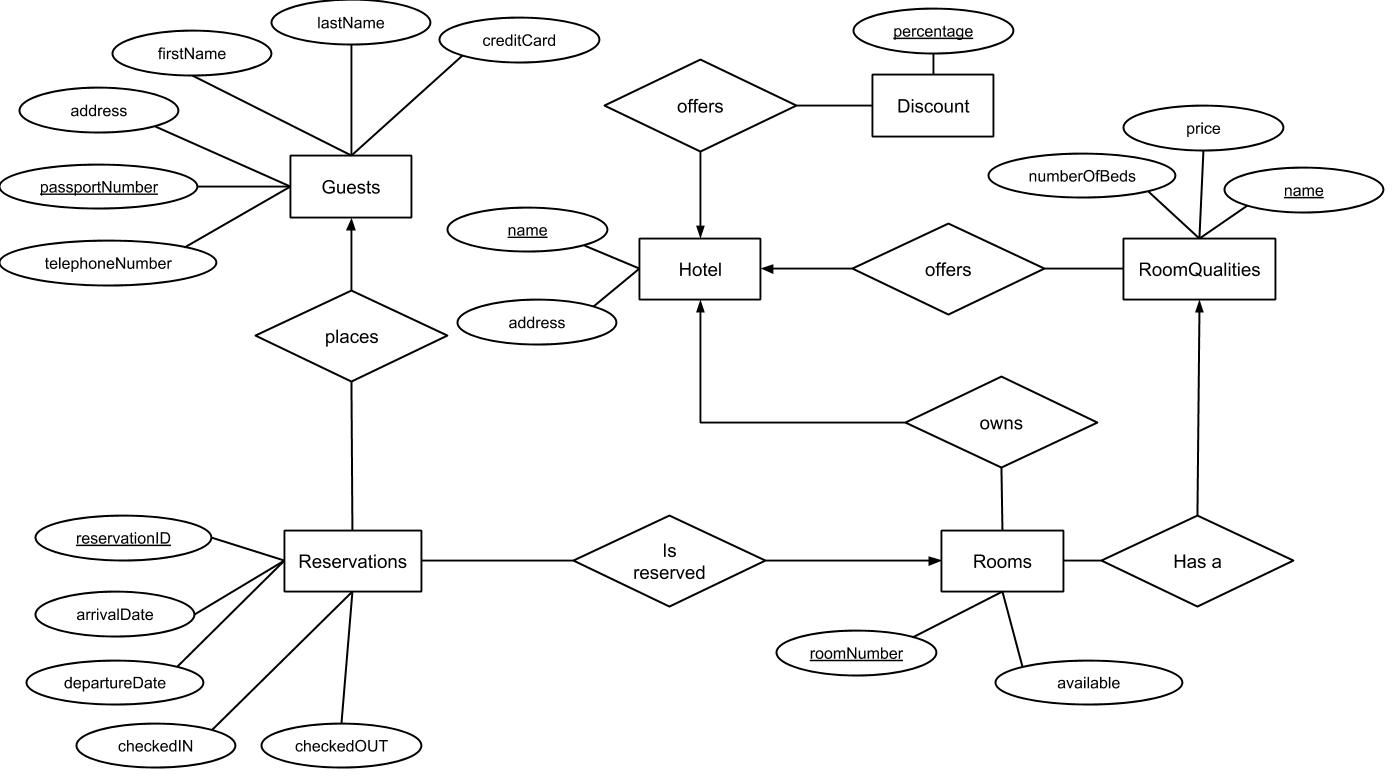
## 2.3 Database

We also had to design our data storage. The choice to use a database was clear, since that way it is dynamic across multiple front-desks, so both Kalmar and Växjö can see real-time data. And because of this, we also decided read, write and update to the database immediately in the program, rather than doing another solution, such as read all data at start and write & update all data at the end. The issue by doing that is in-case the program crashes or if multiple people use the application at the same time. It would mess up the data. So our solution to query right away seemed like the only proper solution.

### 

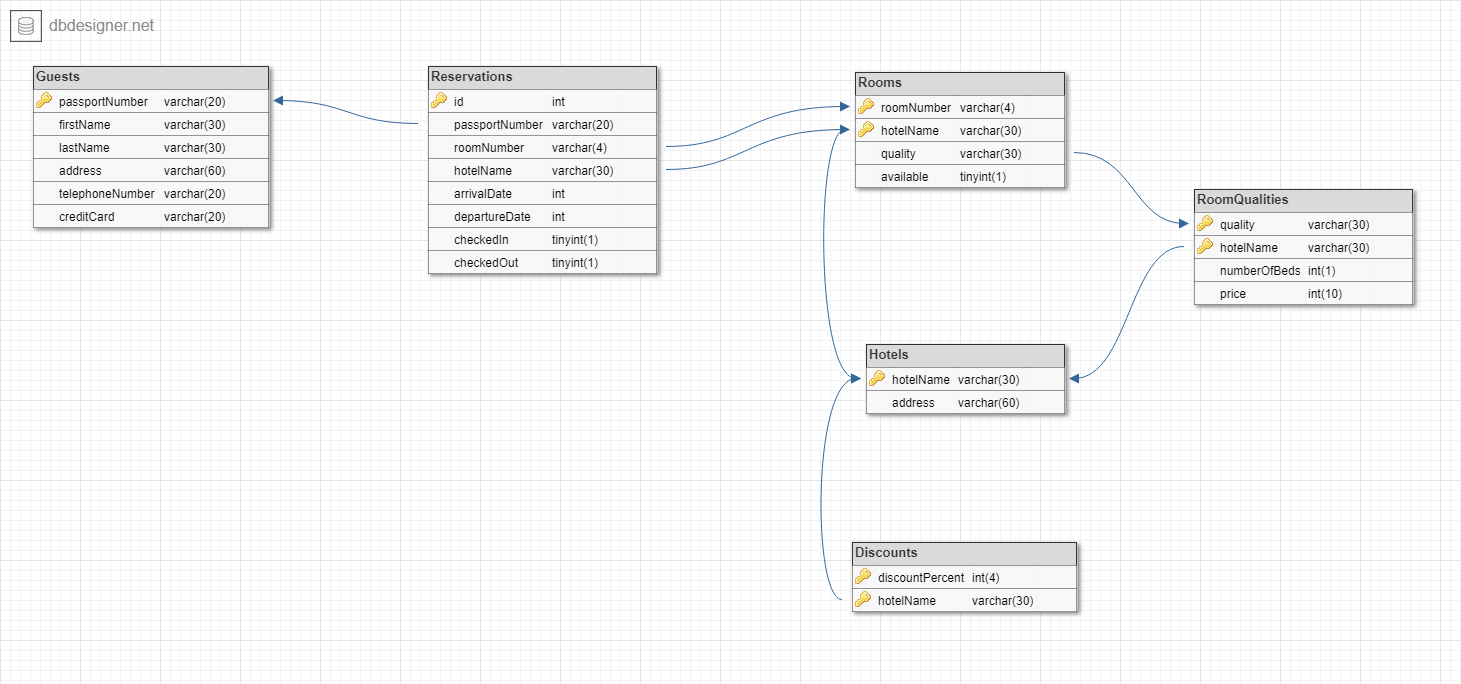
### 2.3.1 E/R Diagram

This E/R diagram discloses the structure we outlined for the database before the implementation of it. We then used this as a basis for the tables, columns and relations in the actual database.



### 2.3.2 Database Schema

This database schema shows the final structure of the database; displaying the tables and their columns, as well as primary and foreign keys.



## 

## 2.4 Refactoring and design decisions

### 2.4.1 Multi-threading

To make the software multi-threaded was a no-brainer from the start. Running everything on a single thread in a GUI-based, client side, software would lead to an unresponsive and slow experience - e.g. queries run on the database would lock up the program until completed.

Utilizing multiple threads, we can make sure that the software is still usable while database queries and the likes are run in the background. Adding to that, it also speeds up the tasks run in the background themselves.

To handle our threads, we took advantage of the ExecutorService - a thread handling service built in in Java.

### 2.4.2 Reusability

We wanted to make sure that the code base we left behind us after the project end was easily understandable, consisted of a good structure and was well documented/commented. Not only to enhance the understandability of the code base in itself, but also to increase the ease of eventual refactoring or reusability of both the whole project, as well as select parts of it. This also leads to better maintainability of the software’s code.

To fulfill these promises, we simply applied some basic principles when coding. Making sure naming conventions were followed (self-explanatory variable and method names), writing JavaDocs for all classes and methods, avoiding/removing redundancy and duplications, striving for low coupling and high cohesion, using the single responsibility principle as well as implementing the software with the MVC-pattern are some of all the strategies we used.  
  
This sums up to a well-structured and easy to understand software, where all or parts of it can be used for further development.

### 2.4.3 Design principles

#### 2.4.3.1 Defensive programming

Since this is a front desk-application meant to be used by people not necessarily possessing extensive knowledge in IT or computer science, we wanted to foolproof the software to the highest degree possible.

Instead of letting the end-users make mistakes and then providing for them fuzzy error messages, we opted to hinder them from doing mistakes from the very beginning.

Some easy examples of this can be:

* Input validation - for example making sure only numbers can be input into the passport field when adding or updated a guest in the system
* Button activation - only enabling buttons when sufficient and correct information has been provided to execute the action
  + A sub-example of this is not being able to press the button to add a new guest before all input fields have been filled in correctly
  + Another sub-example is not being able to press the button to check in a reservation that is already checked in

#### 2.4.3.2 Divide and conquer

The design principle divide and conquer talks about separating the code into smaller sections, as it is easier to understand and reuse for the future.  
We followed this principle throughout the entirety of our program. Rather than having one or two selected controllers we separated it into one controller for each FXML-file.

This made the program easy to build and kept the structure and clarity of the code intact. Rather than it getting more and more clustered and hard to understand as the code progressed. Wf

#### 2.4.3.3 Libraries

*Reinventing the wheel - “waste a great deal of time or effort in creating something that already exists or doing something that has already been done.”*

The above statement is often times true - especially in software development.

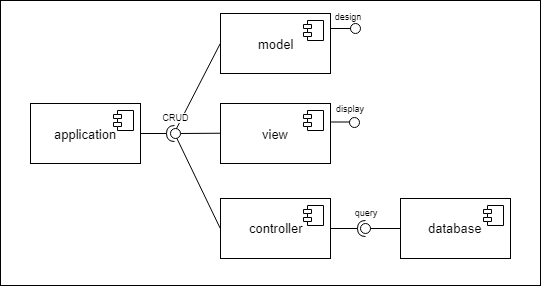
To not reinvent the wheel again - since there are so many fast ones out there already - we opted to make use of libraries for some of the functionalities needed that were not inside the scope of the project.

* As already mentioned in previous sections, we used JavaFX for the implementation of the GUI.
* Since the software is built with a database as a foundation, we needed a way to communicate with it in Java. For this purpose, we used the JDBC mysql connector library.
* We wanted the printing of the bills to result in a nice PDF file, and used the by Apache provided PDFBox library, coupled with FontBox and commons-logging (for adjusting fonts and logging respectively).
  + Adding to this, we created a template file for the bills, which is filled in and saved as a new file when creating a bill.
* To be able to pass the retrieved values from queries on to other methods from our DBParser class, we used the rowset library to access the CachedRowSet class. This enables “transforming” a ResultSet to a CachedRowSet that lives on after the PreparedStatement has been closed.

# 3 Design details

## 3.1 Architecture

### 3.1.1 Component Diagram



### 3.1.2 Components

#### 3.1.2.1 Application

This component is the application as a whole. This is the program you run on your computer to use the application. The application-component is linked to the other components and connects all of them together, making them work together as a whole.

#### 3.1.2.2 Model

This component is responsible for creating objects that will be used in the Controller-component. The objects are created using data from the database and then manipulated and displayed by the Controller and Model working together.

#### 3.1.2.3 View

The view part is responsible for displaying the stages and giving the user something to look at. It creates the so called user interface. The view is responsible for telling the other components what to do, via pressing buttons.

#### 3.1.2.4 Controller

The final component, the controller. It is used for doing the manipulation of the data. Using the objects from the model and displaying the data in the view for the user to see. The controller also works the other way. It gets the user input from the view and takes it and creates objects from the model and stores them into the database, after validating it.

### 3.1.3 Architectural Pattern

We used the Model-View-Controller-pattern also known as MVC. The reasoning behind this is that we could get a good program structure and a easy way to work with object oriented programming and simple database management. And we used JavaFX as a graphics library which directly supports and uses a View-Controller pattern which is perfect for a MVC.

Other patterns that we have also made use of in this project are the Observer- and Dependency Injection-patterns.

The reasoning behind using the Observer-pattern was simply that for the functionality we aspired for, it was basically the only viable solution. It is used in a lot places in our software, for example to update text fields and other elements dynamically as well as directly when changes occur in other elements. One more direct example of this, is to disable and enable buttons depending on the content in associated input fields.

Regarding our use of the Dependency Injection-pattern, it also turned out to be the only solution possible. With the design of the code, making use of the MVC-pattern, in concoction with the JavaFX library and FXML for creating the GUI, we opted for having a controller each for the different stages (windows) being used in the software (this is also the recommended solution). These controllers needed some kind of method to communicate with the main controller for certain functionalities, and here is where Dependency Injection came into the picture.

When creating the new stages (pop up windows), the main controller is injected into the pop up controller through so called injection methods. This enables the pop up controller to access public methods on the main controller, and call them when needed. This way, we could update the main window accordingly after operations had been performed in the pop up windows. The reason why this is the best, and only, solution is simply that we need access to the original main controller - creating a new instance would lead to nothing, as it then would not be connected to the GUI.

### 3.1.4 Deployment

The application is deployed as simply as running a jar-file that can be run everywhere.

The application can be run in multiple instances at multiple places and still sync the data and receive no errors or interference between the two.

The application data is synced and stored using a MySQL-database that is hosted on a VPS on digitalocean. The database uses transactions so there will not be issues where two instances of the application attempts to book the same room. It will be handled appropriately by the database instead of double-booking the room.

The reasoning behind this is clear. We run the software as a jar-file so that it is simple to use and we can include our dependencies along with it. And as for the database, the simplest and best solution is to have a DaaS, since that way you can sync the data properly and avoid confusion or errors.

## 3.2 Modelling

### 3.2.1 Class Diagram

File in original size is included with document. Zooming in CTRL+Scroll UP also works.

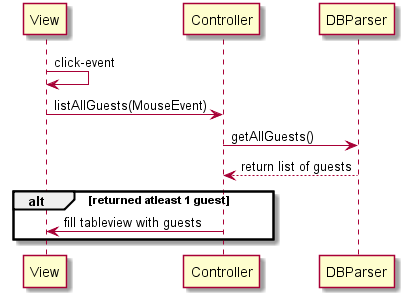
### 

### 3.2.2 Sequence Diagrams

Below here are a list of sequence diagrams showing the flow in some of the more important functions in the system.

#### 3.2.2.1 List All Guest

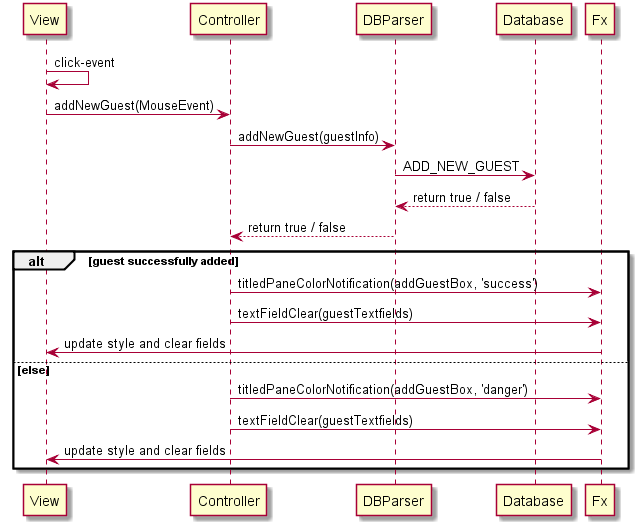
This diagram displays a function that is used in multiple places in the application. It shows the flow when the application requests a list of all guests. It is called by clicking a button from the View and that triggers a function in the Controller that fetches all the guests from the database using the DBParser. And then the Controller returns the guests to the View and the View presents it in a table.



#### 

#### 3.2.2.2 Add Guest

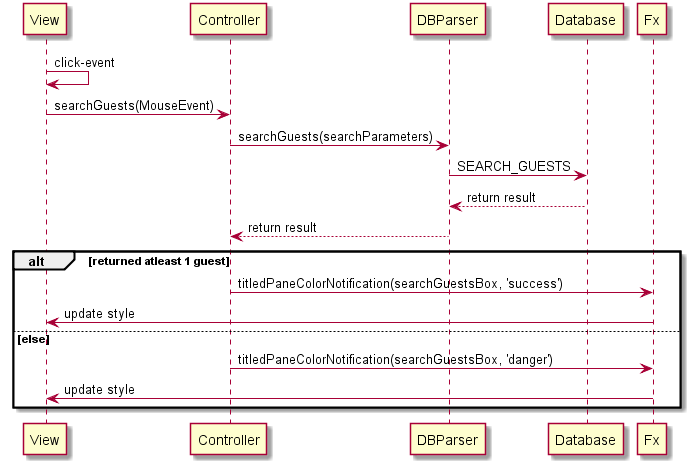
Here is the diagram for the flow of adding a guest to the database. It starts by the User inputting information into the View(UI) and then clicks a button to send the request. Then the View tells the Controller to add a new guest with the selected guest information. And the Controller then contacts the database using the DBParser and sends the information to the database and inserts it. The database then tells the Controller if it was successful or not. Depending on if it is, the UI is updated with either a green color for success or red for error.



#### 

#### 3.2.2.3 Search for Guest

Here you can see the flow for how to search for specific guest(s). It starts with the User clicking on a button after possibly inserting some search parameters. Then the Controller fetches these search parameters if they are there and use it to select guests from the database using the DBParser. It then returns a list of these guests and depending on if it returned 1 or more guests the UI is updated with green if it did, and red if the result was empty.

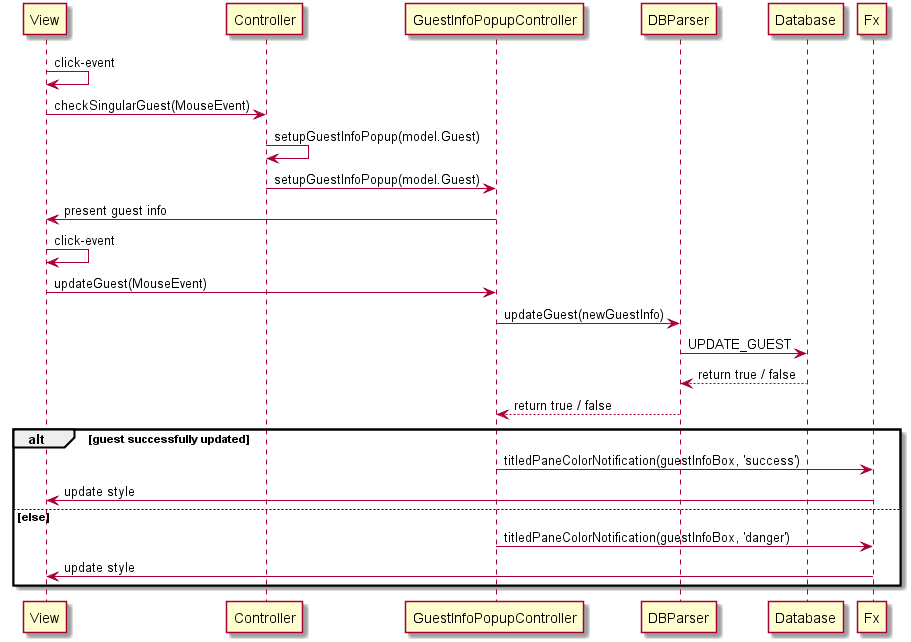


#### 3.2.2.4 Update Guest

Prerequisite: 3.2.2.3 Search for Guest

This is the flow of how to update a currently existing user in the database. It starts by the User opening a specific guests info by double-clicking the result from searching for guest(s).

Then the User updates the specific guests information in the UI and clicks the button to save that guest. When that is done the View contacts the GuestInfoPopupController which then sends the new guests info to the database after validating it. And the database returns true or false depending on if the update was successful. And then the UI is updated with green if it was updated or red if it failed for some reason.

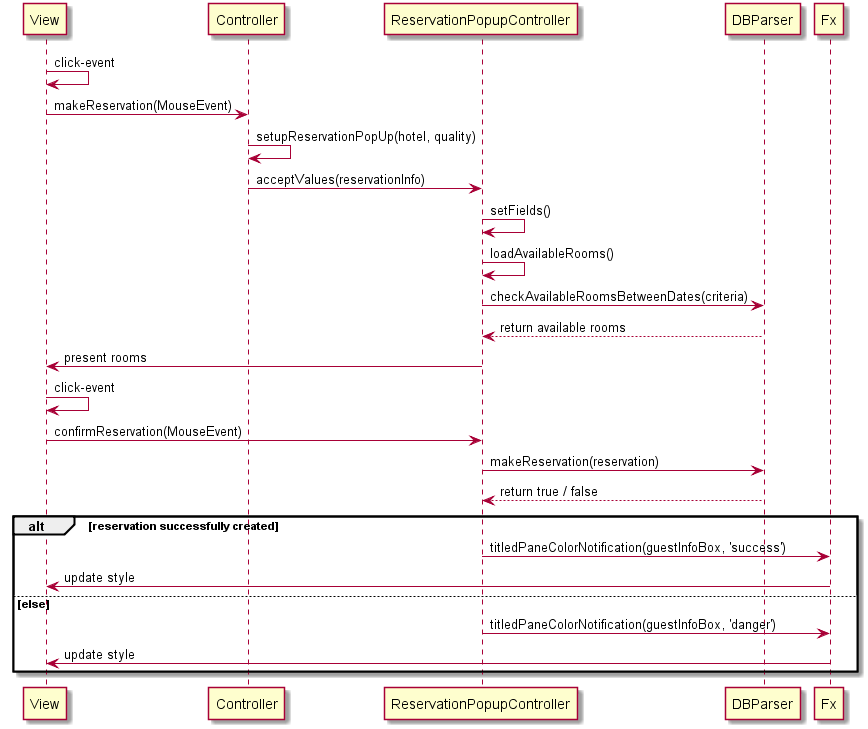


#### 

#### 

#### 3.2.2.5 Make reservation

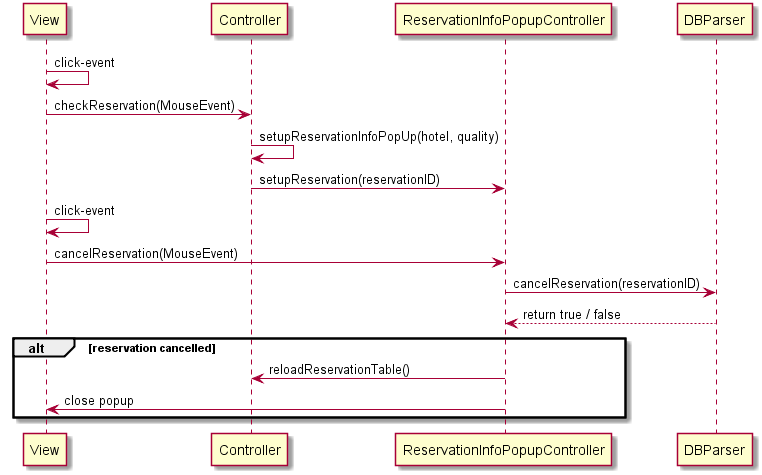
Here is the flow for making a reservation. It starts by the user selecting a hotel or quality, or not. Then clicks a button to proceed with the reservation. And that prompts a new window where the user gets to select a room from all available rooms that have been fetched from the database. When the user have selected a room a button for creating the reservation will be made available. And when clicked it will create the reservation in the database with the selected room & quality & hotel. And then changes the UI to green if it was successful or red if it failed.



#### 

#### 3.2.2.6 Cancel reservation

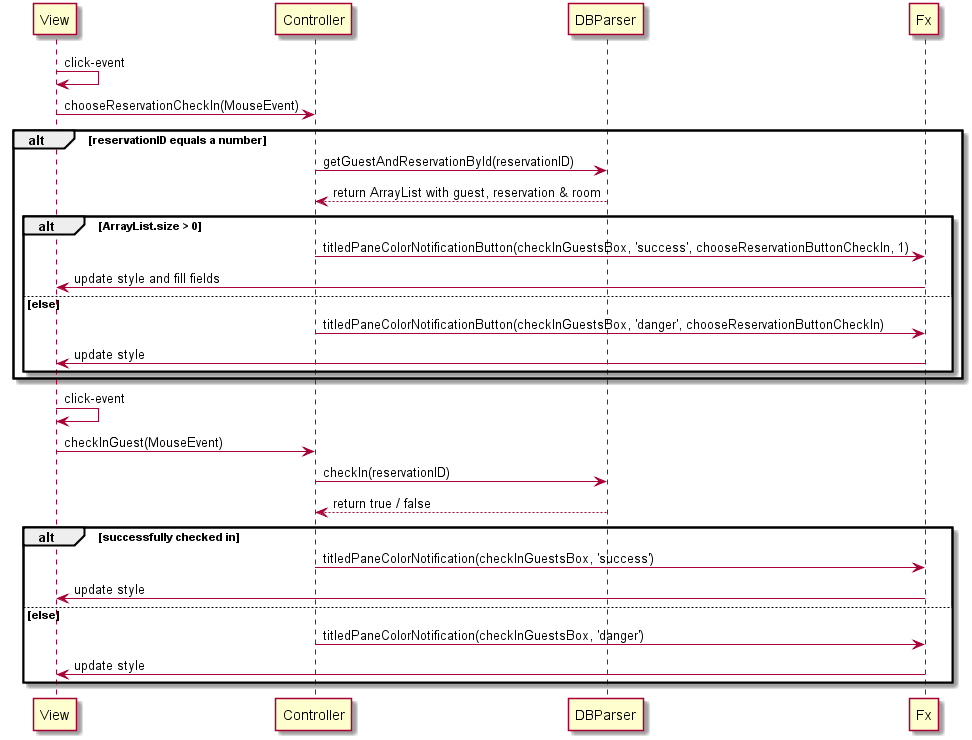
The flow for this shows how to cancel a reservation. It is done by first the User checking for a specific guests reservations and then selects that reservation by double clicking the result. Then the User is presented by the reservation information and the option to cancel the reservation. If the button for cancelling is pressed, a request is sent from the popup Controller to the database to remove the reservation. And if the reservation is cancelled, it closes the popup and removes it from the table.



#### 

#### 3.2.2.7 Check in

Here is the flow of how to check a guest in. It starts by the User writing in the reservation id and then the User clicks the View-button to choose reservation. By doing that the Controller is contacted to get the reservation info and the guests info that the reservation belongs to. If the reservation is returned it fills the textfields with the information. Then the User got the option to check the guest in by clicking the check in button. Then an event is fired to the Controller and it contacts the database via the DBParser to update the reservation as checked in.



#### 

#### 3.2.2.8 Check out

Similar flow as the check-in diagram. The key difference here is that if a guest have successfully been checked out, create a bill in PDF format and generate it using the fields from the checkOutGuestsBox (Guest, reservation and room info) and then it opens the bill for the User and the User has the option to print the bill or not.

