

1.0 Preface

This document was written by Morten Drescher Salling, Mikkel Thomsen & Ivan Naumovski, students at ITU in Copenhagen for their course basic programming. The main reason was to document the design and idea generating phase, coding phase and our group dynamics during our collaboration. We especially wrote it to cover the software product we ended up delivering. This was the central area of this assignment, to document the entire process from start to end.

We would like to thank our professors Dan Witzner and Lars Birkedal in introductory programming, as well as Anne Hatting and Mathias Grüttner in project work and communication.

Table of contents

1.0 Preface.....	1
2.0 Introduction.....	3
3.0 Tasks.....	4
4.0 User manual.....	5
Getting started	6
The button bar.....	7
New Customer:	7
New Reservation:	7
Show Customers:	8
Search Vehicles:.....	8
The navigation bar	8
The calendar frame.....	9
The info panel.....	9
The console panel:.....	9
Troubleshooting:	10
5.0 Analysis.....	11
5.1 Design pattern	11
5.2 SQL info.....	11
5.2.1 Sql Database Design.....	11
5.3 Controller	13
6.0 Technical description.....	14

6.1 Classes	14
6.1.1 Controller	14
6.1.2 Sql	16
6.1.3 View	19
7.0 Testing	23
7.1 Customers	23
7.2 Reservations	24
7.3 Controller	24
7.4 Gui testing	25
8.0 Conclusion	25
9.0 Work process and reflections	26
10.0 Appendix	28
10.1 Expectancy tables	28
10.1.1 Customer tables	28
10.1.2 Reservation tables	28
10.2 Gui test documentation	29
10.3 Log book	31

2.0 Introduction

In this report we will explain what decisions we made during the period where we programmed our software. We will cover what tasks we expect the user to have based on a regular working day, and show a description of what each task contains, and what must be done internally in the software. Tasks also cover what limitations our software has. Most of the limitations are caused by time issues, but some of them are made to simplify the program. We are aware that we have some critical limitations which shouldn't be in a commercial application, but we felt as this was a testing program, used by programmers it wouldn't matter, as the target group wouldn't try to destroy it.

The next thing you will read about is the manual of the software.

The manual consists of a front page and a table of contents which should not be counted as regular pages in the report. The manual shows the user how to use the program covering all aspects for how to start it, how to shut it down, how to debug, create a reservation, create a customer and so on. Everything is divided into chapters to make it easier to read and in the end there's a trouble-shooting guide if you should run into problems.

As you move deeper into the report you can see what design pattern we chose. You can read about why we chose a specific one. There were certain advantages for MVC, which is the one we chose, at first it consisted of 3 parts allow us to easy split up the work in 3 branches allowing us to work much faster.

We also explain what mistakes we made with the design, and how we solved them. In the end we explain about CRC cards and what we used them for.

Next coverage is about our SQL information. Here we explain why we chose MySQL, what advantages it had, such as freeware, no need for servers, and previous experience.

Furthermore there was a premade model in the course we could use.

In this section you will also read about how we retrieve and update information and how it can be done in different ways, and advantages and disadvantages it had.

As a sub chapter you can read about how the database is designed. Each table and row is carefully described regarding what type it has, and why it has that type. Attributes for the fields are also described and nothing in the database has been made by coincidence.

As you proceed you will reach the next part which is the controller. We did many considerations with the controller, especially if it fills the role it was assigned or not. Our deepest concern was about the action listeners in the class. We talked only a very little about it, and it ended up that the Gui held all the action listeners.

We still think the controller does its job and fits into our design, but improvement is possible.

Next on the list is our Gui, due to it having a major role in what the user perceives regarding the software its design, look and feel was crucial to our group. We did not discuss the matter very thorough, but the gestalt laws are considered when making the panels and borders. The technical aspect is a bit messy though but that is due to time issues and the code freeze we were forced to enforce, not to get behind our schedule.

The next you will read about is our technical description of all the classes and the description. We start out with the controller which was our link between the gui and the sql. It played an important role, when passing, receiving and validating information between the two classes (sql and gui).

Here we describe all the methods in the class, what they are used for, when they are used, and what they return.

Next part is the Sql. The Sql class is responsible for providing all the methods that the controller class uses to pass it to the gui. Then it's responsible for open a database connection, and that's about it. It works closely with CustomerDb and ReservationDb and their goals are to retrieve and update information on the different tables. All of their methods, return types, and when they are used are described here.

The last part in the technical description is the Gui. The Gui works close with 4 classes, namely Console panel, window panel, month panel and calendar frame.

Its responsibility is to display the gui, so that the user can get a graphical overview of the data, and manipulate it, as they see fit.

As described earlier it also has action listeners so that it also does something whenever an action is performed.

Then we move on to tests. Here we test on all three branches; however the tests included are only the final ones, so most of the expected outcomes match our expectances; however there are a few results that don't. We are testing both negative and positive tests, and also some that we actually didn't know what was going to happen, and we just assumed something, which turned out to be wrong.

We made expectancy tables over most of it, which will be referred to in the appendix when you read the different parts of our tests.

In the end we move on to a conclusion and reflections where we have notes about how the entire period has been, and what we could have been done better.

3.0 Tasks

Our main task in this project is to create a car rental system that allows the user to rent cars as well as change reservations. We assume that the only user is an employee at the car rental company and that it is only accessible from one computer. In our group we have chosen to focus on the following tasks:

1. **Create a customer** - the user is able to create a customer and store it directly in the database.
2. **Change a customer** - the user is able to change the details of an existing customer.
3. **Delete a customer** - the user is able to delete a customer from the database.
4. **Create a reservation** - the user is able to create a new reservation and store it in the database.

5. **Change a reservation** - the user is able to change the details of an existing reservation.
6. **Delete a reservation** - the user is able to delete a reservation from the database.

The graphical user interface has the current month as a row of days and the cars as columns. All reservations are shown as colored boxes covering the days the specific car is reserved. We have decided that the company are not able to add, change or delete cars from the database, because we think it is not ideal that an employee has access to editing the cars. The user is not able to create a reservation that overlaps from one month to another. It can be done, but it will not show up on the calendar view. If a reservation overlaps it has to be made in as many separate reservations as the reservation runs. When a customer picks up a car it is only possible during opening hours and returning of the car is possible at all hours, as long as it is before opening hours the day after the reservation ends. The program is not able to generate rental contracts or bills; the company either does it by hand or have a bookkeeping program.

4.0 User manual

Our software is designed to rent out cars for a company.

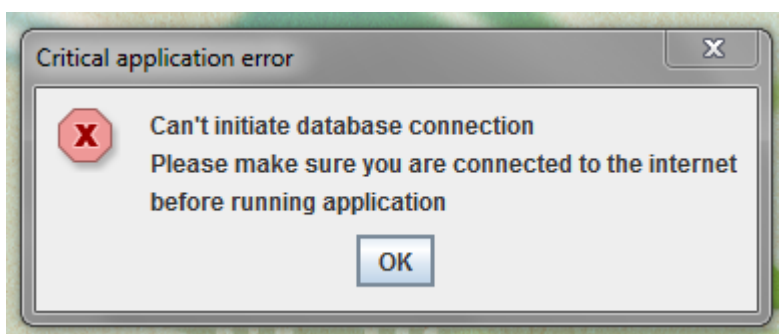
The software doesn't test for user inputs but expect those to be logical and correct. We have not been able to account for all type of inputs due to the deadline we had, though some of the fields have character type checks.

RentIT v.1.3

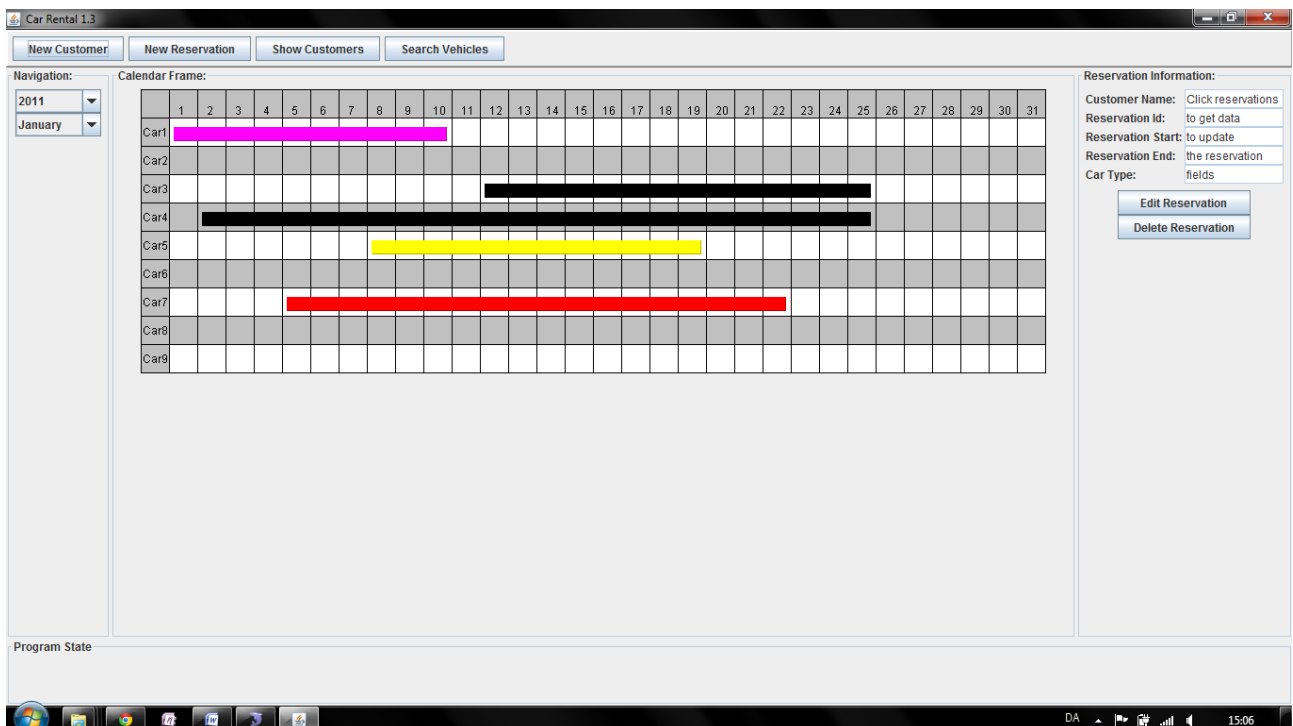
Getting started

This program is run using a JAR extension, all data is stored externally which is why an internet connection is required to run the application.

Before running the application, make sure you are connected to the internet. Else you will encounter an error window looking like this:



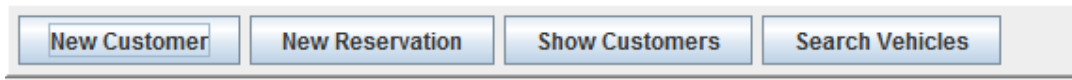
If you have an active connection your main window ought to look something like this:



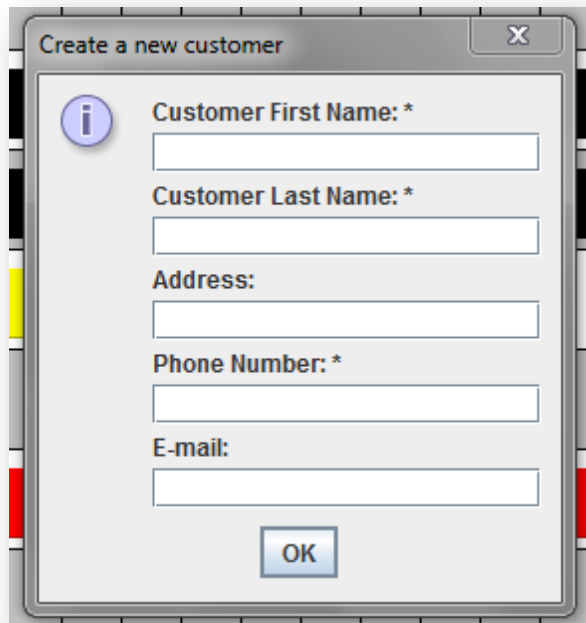
The main frame consists of 5 sub components, which we will cover on the next page.

The button bar

The top left corner of the main frame has a bar looking like the picture underneath:



This bar gives the user 4 options to interact with the 'RentIT' software.

A screenshot of a dialog box titled 'Create a new customer'. It contains five input fields: 'Customer First Name: *', 'Customer Last Name: *', 'Address:', 'Phone Number: *', and 'E-mail:'. Each field has a corresponding text input box. There is an 'OK' button at the bottom right.

New Customer:

This function opens a window that requires following inputs: a *First name*, a *last name*, an *address*, a *phone number*

Example inputs:

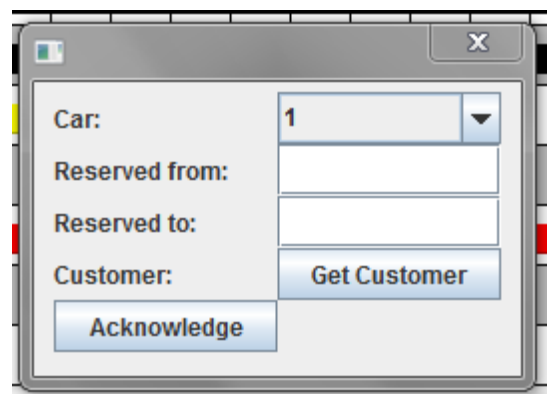
John
Doe
25252525
Rued Langaards vej 7
J_doe@itu.dk

New Reservation:

This window places a reservation on the chosen car.

Car ranges from 1 to 9, the different cars are colored based on the car class (and state) on the calendar frame.

Reserved from and to, are the dates which the reservation is active. Due to restrictions in our design, input ought to be restricted to dates in the same month. If the duration overlaps months, a new reservation for the next month should be created by the user.

A screenshot of a dialog box for making a reservation. It includes a 'Car:' dropdown menu with '1' selected, 'Reserved from:' and 'Reserved to:' text input fields, a 'Customer:' label with a 'Get Customer' button, and an 'Acknowledge' button at the bottom.

The 'Show Customers' window displays a grid of customer information, ordered by customer name. The grid contains 6 entries per row. Each entry includes fields for Name, Address, Phone, and E-mail, along with Edit, Delete, and Get Id buttons.

Customer ID	Name	Address	Phone	E-mail
0	0-MAINTENANCE	Inhouse	00000000	thisCarRental@here.com
G	Gert Knudsen	Frederikssundsvej 44, 8.tv.	61221337	gert_knudsen@gmail.com
I	Ivan Naumovski	Sibbernsvej 08, 4.tv	25376749	inau@itu.dk
P	Paul Richard	Rosevej 64 2660 Brøndby Strand	47574738	none@domain.com
S	Santa Claus	North Pole	69 69 69 69	iEatCookies@NorthPole.N

Show Customers:

The 'Show Customers' window consists of a grid with 6 customer entries per row. These are ordered alphabetically (note that maintenance is a standard entry)

Every grid has information showing the name, address, phone number and email of the customer.

It gives the user a good overview of all the entries in the customer database.

The 'Search for a vehicle type' dialog box allows users to search for a vehicle based on class, start date, and end date. It includes an information icon, input fields for 'Vehicle class:', 'Start date:', and 'End date:', and an 'OK' button.

Search Vehicles:

This button gives the user the option to search for a vehicle of a certain class in a specified time interval. This will return all the vehicles of the corresponding class that aren't booked.

The user is advised to remember the returned car id's for booking the vehicle through the 'New Reservation' dialogue.

The navigation bar

This part of the main frame is used to navigate the calendar frame, which will be covered in the next part.

It consists of two combo boxes which represent the current year and current month. This ensures that the user has a good overview of an entire month's reservations.

The 'Navigation' bar contains two dropdown menus for selecting the year and month. The current selection is 2011 and January.

The calendar frame

This part of the screen is where the application updates all the current reservations for the specific month. It is clickable, which works well with the next part covered in the manual, *the info panel*. Once you click a reservation the information is parsed to the right panel.

Calendar Frame:

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
Car1																															
Car2																															
Car3																															
Car4																															
Car5																															
Car6																															
Car7																															
Car8																															
Car9																															

This gives the user an easy way to edit and delete current reservations, as well as the possibility to restructure for maintenance purposes.

Reservation Information:

Customer Name:	71
Reservation Id:	18
Reservation Start:	2011-01-08
Reservation End:	2011-01-19
Car Type:	5

The info panel

This panel shows information about the current selected reservation, parsed from the calendar frame. The main function that separates the two windows is the 'edit' and 'delete' functions. This panel gives the user tools to alter the current data stored in the database.

When clicking edit reservation, a dialogue window appears, taking parameters such as new dates for the reservation or whether it should be changed for a maintenance job.

Car:

Reserved from:

Reserved to:

☒ Maintenance

The console panel:

On the bottom part of the main frame the console panel is placed. This panel is the user's main source of information for when errors occur in the software. It also reflects which job was recently initiated. It notifies the user of what

Program State

Search cars initiated

Troubleshooting:

“My program refuses to update the calendar frame and the customer window”

This is a common issue this is usually caused by loss of internet connectivity after the application is run.

To resolve this, reconnect to the internet and try to access the same function. If this doesn't resolve the issue try restarting the application.

“I cannot update any information through the user interface”

To fix this restart the application.

“I accidentally deleted the maintenance entry, now the cars for maintenance will not change to the right color.”

You can fix this by adding a new customer entry in the database with a fixed id of 0.

If this isn't possible call our customer hotline (The hotline rates are determined by your region).

5.0 Analysis

5.1 Design pattern

We started working on what design pattern we would like to use, and we only considered Model View Control. We figured that this model suited our needs very good. We could easily split up our project in three “branches” namely Model (Sql class), Controller (Controller class), and View (Gui class). During the coding phase we made a huge mistake implementing an Init class that initiated all of the three classes, and in that way we lost reference on the classes, and nothing worked the way we expected it to when we put all the pieces together.

This resulted in that we had to remove the Init class, and make the Gui our “main class”. The Gui then starts the controller, which starts the Sql, so we at all times had references on everything. The Sql wouldn’t need any references as it just executes queries to the database and then return true/false depending on what happens. It can also return a result set when getting information from the database.

We used the iterative design pattern as well during the development of the program. We tested the current code and thought of alternatives to it. We weighed it against each other and chose the design we found best. We also used CRC cards to give us a general idea of how to structure and implement the program.

5.2 SQL info

This section contains all the info and thoughts for our MySQL database.

We chose MySQL, as it’s a freeware, and something that ITU supported, so we didn’t need our own server. One from our group had previous experience, so he already knew the structure language.

In the start we pulled all info at a time, making an object for each row in the specific table such as customers and reservations. We then stored the objects in an array list for easy access. With this way, we ran into a lot of troubles. Among them were that we now had 2 places with data, instead of one so each time we had to update a piece of data, we needed to do it two places. This created a heavy amount of extra unnecessary work, and it wasn’t very flexible to select all customers or all reservations at the same time. Any way, we decided to go on with this at first, but when we ended up with an array list we didn’t know what to do with. This was mainly because of our bad class design, which didn’t allow get-methods on all of the fields. Now when our project is done we could have implemented this way of doing it, but even if we had the time for redoing it, we don’t think we would, due that it’s a lot of work, and it doesn’t provide any benefits.

5.2.1 Sql Database Design

We started making a whole lot of tables in our database. Those included car, car_type, car_model, type, reservation, and customer. We decided to hardcode most of all this, as another limitation was that the end-user wasn’t able to add/edit/delete cars, and car types etc.

We eventually ended up with only customer & reservations.

Customer design:

- Row: customer_id - ID (int) is a unique Identifier for a specific customer. ID is not used on the front-end (gui), but is used internally. ID makes it possible to have multiple customers which are exactly alike, in case that it arrives.
Attributes:
 - PK (This field is already unique for each reservation).
 - Not NULL. This field is Unique & PK, so it can never be NULL.
 - Auto-increase This field should be unique, so putting autoincr on, made it unique at all times.
- Row: name - the name of the customer (varchar). This is not a unique field but is required (cannot be null) at all times.
Attributes:
 - Not NULL. We don't want the name to be left-out. After all, everyone got a name.
- Row: Address - the address of the customer (varchar). This is not a unique field but is required (cannot be null) at all times.
Attributes:
 - Not NULL. We don't want the name to be left-out. We expect that you won't rent cars to a homeless.
- Row: phone - the phone of the customer (varchar). This is not a unique field but is required (cannot be null) at all times. It has been made varchar if you want to separate the phone number using spaces or using characters like in the US.
Attributes:
 - Not NULL. We don't want the name to be left-out. Those who rent a car will most likely have a phone, and you will most likely need to contact them at some point.
- Row: email - the email of the customer (varchar). This field is neither unique nor required. It's just an optional value you can use if you want to save the email on a customer. It has not been made required, as not everybody has an email.

Reservations design:

- Row: reservation_id - ID (int) is a unique Identifier for a specific reservation. ID is not used on the front-end (gui), but is used internally. ID makes it possible to have multiple reservations which are exactly alike, in case that it arrives.
Attributes:
 - PK (This field is already unique for each reservation).
 - Not NULL. This field is Unique & PK, so it can never be NULL.
 - Auto-increase This field should be unique, so putting autoincr on, made it unique at all times.

- Row: car - the specific car stored (int). This is not a unique field but is required (cannot be null) at all times.
Attributes:
 - Not NULL. We don't want the car to be left-out. You cannot make a reservation without knowing what car-type you want.
- Row: Date_from - when does the reservation start (date). This is not a unique field but is required (cannot be null) at all times.
Attributes:
 - Not NULL. We don't want the start-date to be left-out. If you don't know when it starts, you can't create one
- Row: Date_to - when does the reservation end (date). This is not a unique field but is required (cannot be null) at all times.
Attributes:
 - Not NULL. We don't want the end-date to be left-out. If you don't know when it starts, you can't create one.
- Row: customer - the id of the customer (int). This is not a unique field but is required (cannot be null) at all times If you don't know customer you are going to reserve the car for, you can create a new customer with some dummy data.
Attributes:
 - Not NULL. We don't want the car to be left-out. You cannot make a reservation without knowing what car-type you want.

Car design:

- Row: car_ID - ID (int) is a unique Identifier for a specific car. ID is not used on the front-end (gui), but is used internally. ID makes it possible to have multiple cars of the same type.
Attributes:
 - PK (This field is already unique for each reservation).
 - Not NULL. This field is Unique & PK, so it can never be NULL.
 - Auto-increase This field should be unique, so putting autoincr on, made it unique at all times.
- Row: type - the type of the car (int). This is not a unique field but is required (cannot be null) at all times.
Attributes:
 - Not NULL. We don't want the type to be left-out.

5.3 Controller

We chose to make the controller without any action listeners, the design worked for us, since many of the methods did not change a lot while the gui and where and how the action listeners worked changed a lot more during the development. The action listeners are spread out in the different gui classes.

We did not consider whether it was good or bad class design before quite late in the process.

The reason why we chose the design we have now is that it works as intended, however we should have considered where we wanted the action listeners during the design process. We still think we kept to our overall design pattern, even though we would have preferred to have more loose coupled class design.

6.0 Technical description

6.1 Classes

6.1.1 Controller

The controller class is the “controller” of our program. It is the link between the model and the view. All the action listeners in the Gui that need information, add information or change information from the database are parsed through the controller, where the format of the information is changed and checked before it is parsed to the Sql class. The action listeners themselves are in the View and not in the Controller class, because the Gui is our main class. The controller contains a couple of methods that just returns a result set from the sql without any changing of the data format. It is to ensure that we keep the Gui from interacting with the Sql directly to keep it as loosely coupled as possible.

The rest of the methods return either a boolean or a result set.

The booleans are returned to tell the user that action they performed has actually taken place or not. This was the original idea of the booleans, we kept them in the methods, but they are in the current version not used. That is done on the methods to add, change or delete a customer or reservation and it checks first that the Sql class return true or false before it returns the same to the Gui. We chose to use booleans primarily in the Controller class, because we think that the exceptions that can be thrown from this class are very unlikely and requires the user does something quite against what she is told by the Gui. For instance the date the user inputs can only be ints, but it was changed from the user could write anything, in the field where the user writes the date, the date is checked if it is a real date before it is parsed onto the Sql.

The result sets are returned when the user or the Gui does a search of either customers or reservations. The search parameters are checked before parsed onto the Sql, therefore any return is accepted even if it is empty.

It parses the console panel to the Sql class, to display the exception to the user, when thrown.

showCustomers

This method calls the method to return all customers in the Sql class. It was written early in process, because it is an essential method for the program. The method is used by both creating a reservation and when editing the customers. It is a very simple method that just returns what it gets from the Sql class. Not a lot of consideration went into the writing of the class, because of its simplicity, it doesn't have any parameters and it will just return an empty result set if there are no customers in the database.

showCustomer

This method was created as one of the last method in the Controller class. It used only by the Gui class to obtain information on the customer when a reservation is shown. The method is very simple in the Controller class and does not have any catch blocks, since it is used internally only and the ID parsed by the Gui should match a customer in the database.

showReservations

This method calls the method to return all reservations in the Sql class, it was written early in the process, but it had no practical use for the program and was left unused. As with the *showCustomers* method, the method is very simple and just returns whatever the database contains of reservations.

showReservationsMonth

This method came late in the process of the program, because we determine what and how we wanted the search for the reservations when we wanted them to be displayed on the Gui. The parameters it takes is giving by the Gui, and not the user, when a year and a month is chosen on the Gui. Therefore it does not have a try-catch, because the Gui will only parse a correct month and year.

searchCars

This method was as *showReservationsMonth* written late in the process, when we were sure of how the search would be done in the database and what parameters the user would input in the Gui. From there the method was created to parse the information from the Gui to the Sql.

addCustomers

It was created early in the process, we knew what and how the customers would be stored in the database. The method has not been changed from when it first were created, since the design of how the customers are stored in the database was one of the first things we discussed. The return type boolean was decided to give the user a feedback on whether the method succeeded or not.

changeCustomer

This method was created early and wasn't changed from creation, as with *addCustomer* this method is a simple method and the design of how the customers are stored in the database. The boolean return is to give the user a feedback of whether the method succeeded or not.

delCustomer

The parameter of this method is parsed from the Gui when the delete button is pushed, therefore it is not checked since the Gui can only parse a valid ID. The boolean return gives the user feedback on whether the deleting has been done.

newReservation

This method was created as one of the first methods in the class, but was changed when we went from parsing the dates to the database as ints to Date objects. The parameters were not

changed, since the input could still be used to create what had to be parsed to the Sql. The boolean return has the same purpose as with the other methods, to inform the user of whether the action has been completed.

changeReservation

This method went through the same changes as *newReservation*, the parameters were the same as from the beginning, but the dates parsed to the Sql were changed. The boolean return gives tells the user whether the changing was successful.

deleteReservation

This method was along the other create, change and delete methods created early on in the process and as the *delCustomer* method it is very simple and hasn't changed during the developing of the program. The parameter it takes is used internally only from the Gui class, when the user presses a delete button on the gui.

dateCheck

This method is a private method used by the *newReservation* method to check the input dates, that they are actual dates. It was intended to be used by the *changeReservation* as well, why it was created as a separate method.

dateChange

This method was created when we decided to go from int to Date objects. it is used by both the *newReservation* and *changeReservation* so it is in a method both to avoid code duplication and to make it easier to read the code.

dayToInt

monthToInt

yearToInt

These methods are used by the *newReservation* and *changeReservation* methods to parse the strings received from the Gui to ints used by the *changeDate*. They don't have try catch block, because the input can only be ints when the user types in the dates in the Gui.

6.1.2 Sql

The Sql class is our "model" in our design. Its task is simply to provide an easy access to our real database classes namely CustomerDb and ReservationDb.

The Sql class holds methods for opening a connection and closing it as well. The connection gets established upon creation of the class.

6.1.2.1 CustomerDb

The CustomerDb is responsible for actually querying the database for everything related to customers. This include showing, adding, editing, and deleting information.

The CustomerDb can show all customers at once (for the show customers button), and showing a single customer (to edit a customer).

Methods:

getCustomers:

Not much was considered while coding this method. It just had to show all customers, and the easiest thing to do, was to query for every piece of data, in the table. We order by name as that is usually what you are after.

Return-type is result set. This is due to that we want to return all the data of the customer.

Boolean would be the best so we would know if the sql query was actually successful, yet again, we wouldn't know if the result set was empty.

Another approach to this, could be to store each customer in an object, or all customers in objects in an array. We did this at first, but found it very clumsy with a lot of hassle and extra work. Based on this, we decided to work with result sets.

getCustomer:

This method is pretty alike the above method. It however finds a customer based on what ID you are searching for.

The return-type is result set, as we want to have the details on the customer we just got.

The parameter it takes could also have been name or phone number, but then we would run into issues if two customers had the same name, or the same number.

changeCustomer:

This method changes a customer based on the parameters it receives. Again there wasn't much trouble getting this working. We Decided that it would be the easiest to have one method changing it all, rather than one method to change one thing. If you for some reason don't want to change all fields, all you have to do is to put the original data, or check if the input is "" or null.

The return-type here is boolean. The reason for this, is that we don't want the new details of the customer, we just want to know if the query was successful or not. If it wasn't we show an error and return false.

delCustomer:

Simply deletes a customer in the database. This method isn't really optimized, as old reservations doesn't get deleted, and the database will contain information that is no longer of use, and no longer valid as the client doesn't exist. It might even crash the application.

This method wouldn't be implemented in a real piece of software without deleting reservations as well. Most likely an "active/inactive" attribute would be used.

The return type here is also boolean as we just want to know if we actually deleted the client.

6.1.2.2 ReservationDb

The ReservationDb is responsible for querying the database for everything related to reservations. It can show/add/edit/delete information. Furthermore it can search and check/validate different things. It can for example check a reservation on a specific car, in a specific period. Then it can get reservations between one date and another, and even get it in an interval [start_date_1 ; end_date_1] & [start_date_2 ; end_date_2].

methods:

getReservations:

Gets all reservations. It orders by car due to we have hard-coded a specific position in the gui. This method is no longer used, as we never get ALL the reservations at one time. The return-type is result set as we wanted to get all the results.

getReservationBetween:

Gets all reservations between two dates. It orders by date_from as this is most likely the field you want to sort at.

This method isn't used either. It wasn't flexible enough, and was replaced with Expand, which turned out to be a bit more complicated than we first expected it to be.

This method could have worked with the newest design, but not at the time where we made the decision.

The return-type is result set, to get all the reservations between one date and another.

getReservationBetweenExpand:

This is an expanded version accepting intervals. It is required in order for overlapping months to display correctly on the calendar.

At a later time we switched so that a reservation cannot be longer than a month. This removed the need for this method it but was already implemented, so we decided to go on with this.

The return-type is result set to get all dates between the 2 intervals.

checkReservation:

Checks if a specific car is reserved in a specific period. This could also include the intervals, but that is something we have chosen to leave out, due to time limit.

The return-type is boolean which returns true if the car isn't reserved.

searchReservation:

This returns a result set with what cars are available in a specific period of a specific car type. It returns a result set with all the cars that are not available. The Gui then translate it, to display which cars are then available, as cars are hard-coded into the application.

It could easily have been expanded to allow an unlimited number of cars, but due to time pressure, we decided to leave that out.

Add/change/del reservation are the same as in the CustomerDb class.

6.1.3 View

The view consist of multiple classes, this is done to give each class a main focus for which they are responsible.

- Gui
- CalendarFrame
- MonthPanel
- InfoPanel
- ConsolePanel
- CustomerWindow

6.1.3.1 Gui class:

The Gui has the main string which creates one instance of the Gui class, the constructor of the Gui class consists of two methods - one which initiates the collaborators for this class and one which creates the main frame for the application.

The collaborators are sub components of the gui and an instance of the controller class, which is what we use to parse data from and to the sql, which is instantiated in the controller.

The idea with making so many different classes for the gui was to keep it loosely coupled, but the end result - due to a lot of fixes to make it work rather than sticking with the pattern - has made it coupled to a higher degree than first planned.

Methods:

initiateVariables

Tries to instantiate the collaborators to make the program run - if it fails it returns a popup window with an error.

makeFrame

The sub components are sorted and allocated to a frame. The action listeners for the quick access bar are constructed in the method as well. The action listeners and mouse listeners created in this method also access methods in the controller to generate data for the different panels, especially the mouse event has multiple calls, first it calls the calendar frame and gets an reservation from the two-dimensional array, this array is then parsed to the information panel covered later and used to update the information panel fields.

Please consult the source (Gui - 70 to 178) to get an elaborated explanation on functionality.

createReservation

This method is called through an action listener, it pops up a dialogue window which then waits for an confirmation. Once confirmed it parses the inputs to the next method that finalizes the reservation.

finalizeReservation

This method is only called by create reservation, it sends an request to the controller to create a new reservation data entry in the database.

createCustomer

is similar to create reservation, this method on the other hand does not have a finalize version, but calls the controller from within the method.

searchVehicles

similar to create reservation, once initiated it generates a window that then parses the input data over to the search vehicles result method.

searchVehiclesResult

This method queries the database through the controller based on the parsed inputs from search vehicles. Then it generates a window with the number of cars corresponding the class searched for.

quickAccessBar

This method generates a set of buttons, which are used to generate the main frame. Called by the make frame method.

6.1.3.2 Calendar frame class:

This class consist of an constructor which takes a instance of an controller and an instance of a console panel as parameters. This is due to the class itself needs to update parts of the UI based on the returned values from the controller.

Methods:*populateReservations*

This method calls the showReservationsMonth in the controller, based on the current month and year parsed to the controller from the calendar frame - this results in an Result Set. Then it uses the class Reservation to populate a 2 dimensional array with reservations. This is done so we can click on the calendar frame and update the information pane(covered a tad later) based on the input. Afterwards this calls a repaint on the calendar frame.

updateCalendarYear

This method updates the year corresponding to the input parsed from the combo box. It also calls update calendar month, to ensure that the last inputted month also is the one used for repainting the canvas.

updateCalendarMonth

This updates the month shown on the calendar frame. A check ensures that we don't have a leap year, since this would result in an alternate length for the month February. Afterwards it changes the number of columns based on the length of the month to reassure that we have

enough room to generate the graphical representation of the reservations - which is done in the paint method.

paint

Does all that has something to do with graphical representation of the data needed and its predesigned calendar look. Before we actually merged the 3 branches, two dummy arrays were used with a start date and an length, this worked well to show the bars, and the same method was kept when we finished the new method with the 2 dimensional array of reservation data, since the methods in the reservation class, has methods to get the two integers needed.

isClicked

This method is called from the gui, in the form of a mouse listener attached to the calendar frame. This parses the mouse event to isClicked, isClicked gets the coordinates and checks the reservation array for data entries - then it returns a Reservation and calls information panel.

6.1.3.3 Month panel class:

This class does not need too much covering, since it creates a panel with 2 combo boxes - action listeners for these are generated in the Gui class. It could have been built in the gui class itself - but the whole idea was to separate every panel for a loose coupling.

6.1.3.4 Info panel class:

Generates the east panel, this holds fields with information about the last clicked reservation. And has two buttons that can change the current active id. Design wise we altered the constructor to be able to call methods from the constructor and the calendar frame - we did this by parsing the references to static fields in the class.

Methods:

editReservation

Does the same as most of the other edit methods - creates a dialogue with input fields, once activated it parses the input data to the finalizeReservation method which finalizes the edit action.

finalizeReservation

Calls the method changeCustomer in the controller and updates the current reservation id, with the new input data from edit reservation.

deleteReservation

Deletes the current reservation id, gotten from the last reservation clicked in the calendar frame.

setInfo

Updates the info based on the reservation parsed from the Gui. The reservation class has the methods needed to get the right information to fit in the text fields of the info panel.

6.1.3.5 Console panel class:

This is a very small class, its sole purpose is to show what the program state is. If any of the modules get an error they parse it out to this part of the GUI through its updateBotConsole method.

Methods:*updateBotConsole*

The method basically just takes a string and sets it as the text for the text area in the bottom of the main frame.

6.1.3.6 Customer frame class:

This is the last part of the Gui, the window is created through the 'initiateVariables' method call in the Gui and takes two parameters of type controller and console panel and calls the build method.

Methods:*build*

Generates the frame that is shown when the triggering button is clicked. The frame is set to hidden as default every time the update is called its set to shown - the default close operation is "hide on close" which makes it go in the background while the application is running.

updateCustomerWindow

The list is purged and populated with new data from the controller showCustomers method each time this method is called. This ensures that the list always is populated with the newest data.

Due to the window having three buttons which need to refer to different id's we have hash maps with button references as keys and int values for customer id's to support different features.

getLastID

This method was made to support the feature with being able to get customer id's when creating a new reservation. It basically returns the corresponding id to the customer for use in the reservation dialog..

editCustomer

This method generates an dialog as well, it takes an action event as an parameter to be able to find the customer id in the corresponding hash map. The reason we chose to design the method this way was the population of the list. Since we used a result set, the data was not stored for direct access but instead we had to query the database every time. In the action

listeners we kept getting problems with being forced to input static variables instead of dynamic variables. The whole workaround for this was to create the hash maps, with the “event trigger”(button) as key and the id as value. This way we could track the source, and get the corresponding value from the hash map. After reflecting on this choice a stronger design pattern would have been to stick with the reservation and customer classes and create array lists or any similar structure to support graphical representation in a different and better structured way.

getCustomerId

Is only used to fetch the customer id of the clicked customer on the list.

deleteCustomer

deletes the customer with the id corresponding the clicked button.

7.0 Testing

We have chosen not to test all the showing methods in the SQL class, and just in the controller. After all, when we test it in the controller, it automatically tests it in the SQL class. We have done it previously, however we ran into time issues, and therefore we have chosen not to include further tests on the sql class.

Test we made on the controller were made on all the methods used in the final version of the program. The methods not used by the final version were never used by the Gui class and therefore not tested.

7.1 Customers

Note that our expectancy tables are located in appendix 10.1.1

Some of the tests failed earlier on, due to the SQL syntax was wrong, but when we got a rid of that the code worked as expected.

We see that when we input all fields with valid data, we get a return of true. We expected that, as this is what the method does.

If we create one without a telephone number we expect the query to fail, as in our database we have that phone cannot be null. We saw that we were right.

If we move on to deleting customers, we see that when the id is not existing or null it will return false. When the ID is missing, it sends a null value. No entries got a null-value as the key is not null. When we send an empty string, it will also fail, as ID is the type of int, and not string. A negative number is an int, however as ID is auto-increasing, we wouldn't reach a negative number at any time (as long as the first ID is 1). This means that if we had a customer with ID -1, then the result would be true.

If we look at edit customer, we see that it works pretty much same as add customer, except that it also includes an ID, which is the same as with deleting customers.

Therefore we expect the values to be exactly as earlier, and we see that they are exactly as earlier.

7.2 Reservations

Note that our expectancy tables are located in appendix 10.1.2

Some of the tests failed earlier on, due to the SQL syntax was wrong, but when we got a rid of that the code worked as expected.

If we take a look at creating reservations, we see that we expect the first one to fail. We expected that when it was an int with auto increase, it wouldn't be possible to create a reservation with id -1. This appeared to be possible. This is most likely because an int also can be -1, so technically this is possible but would most likely never happen.

The next issue with creating is because we use 'abc' as a string instead of a date. The SQL expects to receive a date, and when it doesn't it will not create the record, leading to a false. In the next test we put all the right parameters and expect a true which we also get.

The last one, is much like the one described earlier. We can insert a customer with -1 as it's an int, but we don't have a customer with id -1.

The edit functions, works much like with customers.

When we edit the first one, we used a non-existing ID, which is why it fails. If -1 was an actual reservation it would work.

We see on the second line, that yet again we didn't think everything through. An associated customer can be assigned, even if that customer doesn't exists. It would never happen as long as the user didn't have access to the database, as the gui picks the ID in a read-only format for the user, but technically it is possible.

7.3 Controller

We made a few manual test before the test class was made, one test showed that we could make a reservation in 2011, but not change a reservation in the year of 2011. We found the mistake, it was a condition where it said higher than and it should have been equals to or higher than. We can't explain why it was able to create a new reservation, since it is the same checks performed on the parameters as when we change a reservation. We only know when we changed it, it works correctly on both new reservations and changing reservation.

The controller test repeated some of the test on the Sql we already made, but the tests with the result sets were meant to cover both the Sql and the Controller since the Sql tests were made before and didn't test the returns from the database. That was also the only place the actual result were not what we expected. We expected the database to throw an exception on a negative car number, but it didn't it returned an empty result set instead. That was actually a good thing, because it meant that we did not have to ensure that the search parameter input were only positive integers. It will then tell the user that a car.

7.4 Gui testing

Due to the gui mainly consisting of graphical representation, and calls to the controller. The way we tested was with usability test, by allowing different people do some tasks with our system. By giving them tasks like creating customers, reservations and editing them as well - while we monitored the database for alterations in our data we had a good overview of whether it worked as intended. This also helped to find a lot of the flaws, which we fixed before the newest version which still has a lot of bugs. More of this is covered in the appendix 10.3.

8.0 Conclusion

We started by outlining the tasks and limitations of the project we created. We made the tasks list to ensure our final program would live up to the requirements of the course and our own expectations. The limitations we set up had a similar purpose, we knew we had limited time to complete the project and we did not want to use a lot of time making a lot of features and smart hacks that were not a part of the requirements.

In the user manual, we covered the steps a user takes of the everyday use. Our limitations are also covered by telling the user how the program should be used and what steps to take if the user makes a mistake. Every step is complemented by a picture of what the user should see, that is to give a user who is a complete stranger to our program an easy introduction to the program. It also gives the reader some idea of how the program works and leads on to the analysis.

In the analysis we went through the thought process in the beginning of the project. We showed that we wanted to design our program according to the Model-View-Controller design pattern and how we used it. We did not consider other design patterns, we felt confident in the M-V-C pattern. We used the iterative design pattern continuously on at least the major parts of the program. We found that some parts of the application was not designed completely as intended, but we only realized it when we started to evaluate the process. The Gui part of the program is the largest part and that is one of the places most of our initial design process was allocated. It was also where most of the limitations had an impact.

The change from a local representation(database data being generated in lists) of the database to result sets was one of the larger changes we made and it removed a large part of the model we had at that time.

From there we described the important methods of the essential classes of the program. We found that the classes could have been optimized, the controller could have had the action listeners and the main method. The way we store dates of a reservation could also have been optimized as a start date and length of the reservation, making searching the reservations easier and make it possible to solve the problem of reservations overlapping months better.

We then ended up discussing our tests and the changes they had on the program. We were quite confident in what input/output had what response by the time we made the tests, therefore what we expected were almost in all cases what we got. During our manual testing we encountered several issues that we had not expected, primarily in the Gui and the Sql.

The program lives up to the requirements and our limitations, therefore we feel we have lived up to what was expected of us. We discussed and evaluated our decisions and we feel we have a solid and good program we can vouch for. We say we have fulfilled the requirements of the project and what we have obtained through the course and projects shine through in both our program and in our report.

9.0 Work process and reflections

The way we have worked is to split up our project in 3 main tasks, the Gui, the Sql, and the Controller. We divided it and wrote javadoc and documentation, so that we can learn about each other's code, however we haven't sat down and talked about it. We plan to do this sometime after we turn in the project, as we by then have a much larger amount of time, so that we don't need to waste the one that we got to write the report and the software.

Tools that we have used in the process:

Eclipse

GoogleDocs

DropBox

GitHub

Expectancy tables

Model/View/controller design

MindView (mind map)

Logbook

We chose the Eclipse editor as we found it easy to use, integrate with git, it supported a lot of shortcuts. Netbeans was among one of our alternative editors, but we had issues integrating it with Git.

We used GoogleDocs to write the report, allowing all three of us to write in the document at the same time, at the end we would eventually put this into word, and convert images, make

table of contents etc, that GoogleDocs either doesn't support, or doesn't support very well. We primary used DropBox to share files in the group. We converted to DropBox after using GitHub, which didn't turn out very well, read more about that below. DropBox was a good way to share files, so we had them locally on our machines, and then we had to confirm with everyone, before uploading to the shared folder.

GitHub was our first way of sharing files. It is a powerful system allowing to go back in earlier versions, and restore the code. We also had different branches, and can track what changes we made. GitHub wasn't very user friendly at all, and we spend many hours getting it to work. Sadly it turned out that we ran into some issues regarding push/pull and in the end we couldn't do anything to our master branch. We eventually moved to DropBox. Expectancy tables was a good idea, and it helped us debug many of our functions. With the sql, part it was quite easy to foresee if a query would fail/succeed as the mysql server is coded by another company. It's used by many people and companies worldwide, and expected to work, just like office/word is.

We designed our software after the model/view/control design, so we always had the required references, but for some reasons we navigated away from it, and created an init class to start the application, which was a huge mistake, that we had to revert later on.

We used MindView to create CRC cards, and see what classes was responsible for what, and keeping a clear and understandable structure in our project. By time we stopped maintaining it, and keeping it in our heads, which was a bad idea in the long run. Luckily the project wasn't a huge one, and not many members were involved.

Everyday we kept logs of what we had done in the group, and how far we were, even though we forgot some days, however it was usually added the next. The entire log can be read in appendix 10.2

There are surely more tools that we could have used, such as a bugtracker, Gantt chart, collaboration agreement and so on. We chose not to use these tools, as we didn't think we had the time or the need for them. But it appeared from time to other that it would be nice to have a place where we could put group information, such as what time we were going to meet next time.

10.0 Appendix

10.1 Expectancy tables

10.1.1 Customer tables

Customer tests	Tasks	Data	Expected	Actual
		String name; String address; String Phone; String email		
	New customer	'Paul Richard', 'Rosevej 64 2660 Brøndby strand', '47574738', 'none@domain.com'	TRUE	TRUE
	New customer	'Paul Richard', 'Rosevej 64 2660 Brøndby strand', '', 'none@domain.com'	FALSE	FALSE
	New customer	'Paul Richard', 'Rosevej 64 2660 Brøndby strand', '87878745874387434387', ''	TRUE	TRUE
	New customer	'', 'Rosevej 64 2660 Brøndby strand', '47574738', 'nonedomain.com'	FALSE	FALSE
		Int ID;		
	Del customer	'1'	TRUE	TRUE
	Del customer	'0'	FALSE	FALSE
	Del customer	'-1'	FALSE	FALSE
	Del customer	''	FALSE	FALSE
		Int ID; String name; String address; String Phone; String email		
	Edit customer	'-1', '', 'Rosevej 64 2660 Brøndby Strand', '47574738', 'none@none.com'	FALSE	FALSE
	Edit customer	'0', 'Paul Richard', 'Rosevej 64 2660 Brøndby Strand', '47574738', 'none@none.com'	FALSE	FALSE
	Edit customer	'1', '', 'Rosevej 64 2660 Brøndby Strand', '47574738', 'none@none.com'	FALSE	FALSE
	Edit customer	'1', 'Paul Richard', 'Rosevej 64 2660 Brøndby Strand', '47574738', 'none@none.com'	TRUE	TRUE
	Edit customer	'1', 'Paul Richard', 'Rosevej 64 2660 Brøndby Strand', '47574738', ''	TRUE	TRUE

10.1.2 Reservation tables

Reservations test	Tasks	Data	Expected	Actual
		int car; Date reserved_from; Date reserved_to; int customer;		
	New reservation	'-1', '2011-12-06', '2010-12-06', '1'	FALSE	TRUE
	New reservation	'1', 'abc', '2011-12-04', '1'	FALSE	FALSE
	New reservation	'1', '2011-12-04', '2011-12-06', '1'	TRUE	TRUE
	New reservation	'1', '2011-12-04', '2011-12-06', '-1'	FALSE	TRUE
		Int ID;		
	Del Reservation	'1'	TRUE	TRUE
	Del Reservation	'0'	FALSE	FALSE
	Del Reservation	'-1'	FALSE	FALSE
	Del Reservation	''	FALSE	FALSE
		Int ID; int car; Date reserved_from; Date reserved_to; int customer;		
	Edit Reservation	'-1' '-1', '2011-12-06', '2010-12-06', '1'	FALSE	FALSE
	Edit Reservation	'1' '1', 'abc', '2011-12-04', '1'	FALSE	FALSE
	Edit Reservation	'1' '1', '2011-12-04', '2011-12-06', '1'	TRUE	TRUE
	Edit Reservation	'1' '1', '2011-12-04', '2011-12-06', '-1'	FALSE	TRUE
	Edit Reservation	'1' '1', '2011-12-04', '2011-12-06', '1'	TRUE	TRUE

10.1.3 Controller test tables

CustomerTest	Tasks	Data	Expected	Actual
		String name; String address; String Phone; String email		
	New customer	"Paul Richard", "", "47574738", "none@domain.com"	FALSE	FALSE
	New customer	"", "Rosevej 64 2660 Brøndby Strand", "47574738", "none@domain.com"	FALSE	FALSE
	New customer	"Paul Richard", "Rosevej 64 2660 Brøndby Strand", "47574738", ""	TRUE	TRUE
	New customer	"Paul Richard", "Rosevej 64 2660 Brøndby Strand", "", "none@domain.com"	FALSE	FALSE
	New customer	"Paul Richard", "Rosevej 64 2660 Brøndby Strand", "47574738", "none@domain.com"	TRUE	TRUE
		Int ID		
	Del customer	"1"	FALSE	FALSE
	Del customer	"1"	TRUE	TRUE
	Del customer	"0"	FALSE	FALSE
		Int ID; String name; String address; String Phone; String email		
	Edit customer	"0", "Paul Richard", "Rosevej 64 2660 Brøndby Strand", "47574738", "none@domain.com"	FALSE	FALSE
	Edit customer	"-1", "Paul Richard", "Rosevej 64 2660 Brøndby Strand", "47574738", "none@domain.com"	FALSE	FALSE
	Edit customer	"1", "Paul Richard", "Rosevej 64 2660 Brøndby Strand", "47574738", "none@domain.com"	TRUE	TRUE
	Edit customer	"1", "", "Rosevej 64 2660 Brøndby Strand", "47574738", "none@domain.com"	FALSE	FALSE
	Edit customer	"1", "Paul Richard", "Rosevej 64 2660 Brøndby Strand", "47574738", ""	TRUE	TRUE
ReservationTest		int car; Date reserved_from; Date reserved_to; int customer;		
	New reservation	"-1", "17122011", "19122011", "1"	FALSE	FALSE
	New reservation	"1", "20122011", "19122011", "1"	TRUE	TRUE
	New reservation	"1", "17122011", "20122011", "1"	TRUE	TRUE
	New reservation	"1", "17122011", "19122011", "-1"	FALSE	FALSE
		int ID		
	Del Reservation	"1"	TRUE	TRUE
	Del Reservation	"-1"	FALSE	FALSE
		int ID; int car; Date reserved_from; Date reserved_to; int customer;		
	Edit Reservation	"1" "1", "17122011", "19122011", "1"	TRUE	TRUE
	Edit Reservation	"-1" "1", "17122011", "19122011", "1"	FALSE	FALSE
	Edit Reservation	"1" "-1", "17122011", "19122011", "1"	FALSE	FALSE
	Edit Reservation	"1" "1", "20122011", "", "1"	FALSE	FALSE
	Edit Reservation	"1" "1", "20122011", "19122011", "1"	TRUE	TRUE
		Int month, int year		
	Show reservation	2, 2011	NULL	NULL
	Show reservation	0, 2011	NOT NULL	NOT NULL
	Show reservation	13, 2011	OutOfBounds Exception	OutOfBounds Exception
		int car.Type; String reservedFrom; String reservedTo;		
	Search cars	-1,"01122011", "31122011"	OutOfBounds Exception	NULL
	Search cars	1, "01122011", "31122011"	NOT NULL	NOT NULL

10.2 Gui test documentation

The plan with this test is to confirm that the functions implemented through the gui work as intended and to catch any flaws which we can fix. It consists of a few tasks that the user is supposed to carry out while we cross reference with the database.

- 1) Create a new customer.
- 2) Try searching for cars of class 1 in January '11
- 3) Create 2 reservations with the new customer on one of the cars available
- 4) Change one of the reservations to be in the same time interval just as maintenance instead

Results:

While monitoring the user his inputs were as follows:

1) *Creating a new customer:*

Inputs:	Results:
Joe	Joe
Doe	Doe
Paradisæblevej 162	Paradisæblevej 162
23435373	23435373
jd@google.com	jd@google.com

First tests shows that the input was matching (right column is from the database)

2) *search for cars in January '11*

Inputs are class and dates:

1 01012011 10012011

The returned result was that the cars of class 1 (1, 2 and 3) had 2 free cars for the duration - car 2 and 3.

3) *Create 2 reservations on one of the free cars with the created customer*

Inputs:

car 2 01012011 04012011
car 2 05012011 08012011

By looking at the graphical representation - we can deduct that the creation is functioning as needed.

4) *Change 1 of the reservations for maintenance*

Chose the first reservation (car 2 , 01 to 04)

Inputs:

2 01012011 04012011 maintenance = true

The graphic representation changed to black, this means that the general design is working as intended. Looking at the database the customer id has been changed to 0 which means the customer is the maintenance default customer.

10.3 Log book

22-11-2011

We started out by making a mind map over what features classes (and methods) we felt we would need at this present time

After that we started to code some of the gui, so that we knew what the gui would look like, and thereby make it easier to adjust the methods in the application. Short after this we started to create the database layout.

24-11-2011

We started working on a project analysis. We then made a class diagram and CRC cards to determine what classes we needed and how they relate to each other.

After that we began creating the classes and some of their interfaces.

28-11-2011

Today only Mikkel and Ivan met at ITU, and they started to get the SQL class going so that we could actually pull some information from the database. It took quite a lot of time, but they succeeded in the end and briefed Morten the following day.

29-11-2011

All connections to the database was finished, it's now possible to insert/edit/delete data for both reservations and customers.

01-12-2011

We switched over to dropbox as github was way too difficult to figure out with the amount of time that we had.

As we gathered all the classes we now tried to make the different parts: Model, View, and Controller working together.

05-12-2011

We did some thoughts about the results that use from the mysql, and decided that it would be easier to use Result Set instead of first converting the result set to objects and arrays. Then we started to make some expediency tables and try off some tests.

06-12-2011

We implemented all functions regarding the customers into the gui, by now it was possible to get a list with all customers, edit, delete and add new customers.

08-12-2011

We decided to put a new limitation due to time issues. It's not possible to reserve a car for more than one a month in a row. If you want to, you have to create multiple reservations, for the same customer. This could eventually be implemented as an automatic process.

9-12-2011

Today we finished the software. We agreed that Ivan would improve the gui at home, and show the reworked version when we were to meet at Monday. We left of early to celebrate weekend and that our software was finally done, and all we had to do now was to write the report.

12-12-2011

Today we were finally finished with the project, or so it seemed. Ivan kept running into issues while trying to make the gui slightly nicer. Morten and Mikkel spent time with the report, and by the end of the day it was around 11 pages.

13-12-2011

The report is nearly finished. We finished technical description, the analysis and the user manual. The only things left are the introduction, preface and conclusion. This will be done tomorrow, and then we will fix spelling mistakes etc in the weekend.

14-12-2011

Today we wrote the introduction, conclusion, and it seemed we also missed an analysis, testing and the technical description of the gui class. All of those has been finished and we are only missing the preface.

Tomorrow we take a day off, and at Friday we will finish the preface, read the report through, and most likely turn it in, if we make it before the office closes.

Further more we should meet and discuss the different classes so that we can all understand them.

16-12-2011

Today we finished reading the report, and correcting the major mistakes.

We can hereby conclude that the report is almost finished and ready to turn in, but we still need to burn down the application, formatting the report so it looks nice etc. We will do all this at Monday before turning everything in.