Vamsi Lingamaneni
214 724 058
EECS User: vamsi

EECS 3311: Final Project Report

DEMO Video link :

https://drive.google.com/drive/folders/1GMJRy5qBGM4z1lCKavtPBa3MlrjP6eTu?usp=sharing

I have explained in this video the visual aspects of how I fulfilled the requirements stated in the SRS.

Design pattern

For my project I have used the factory pattern for the overall design of the system. There are three main users involved in my project, parking enforcement officers, parking customers and system administrator. Customers, enforcement officers and admin have common attributes, like name, email, username, password, only thing that differs between both of them is permission level. All of these users can inherit these common attributes from the User class. Although the system administrator already has a master login and exists in the system which was stated in the SRS. The system administrator still needs username and password to login to the system. In this case, the admin will go with the username “admin” and password “admin”. Once the user inputs their login information (username and password), the system will go to the user database (user.csv) and check for the user’s record and let the factory method (jButtonLoginActionPerformed method in LoginFormGUI class) know which type of user they are, based off their id number. For Customers the id is set to 1, when they register; for enforcement officers when they are added to the user database by the admin, they are set to 2. The id number determines which subclass the user heads to, as their subclasses are pre-determined with their respective restricted actions. So based off the login information, the factory method can classify which user is a parking customer, administrator or parking enforcement officer. There are differences in the dashboard and tasks of customers, administrator and enforcement officers, as expected based off the SRS. For example, the administrator is the one updating payment status and adding or removing enforcement officers compared to the customer who books, cancels or pay for parking. Which is why factory pattern was used for the overall system as the attributes needed to login for each of customers, administrators and enforcement officers, are the same. However, once they are logged in, their actions can defer, but they are using the attributes from the User class.

Finally, another design pattern which I used for certain parts of the system is the observer pattern which is a behavioural pattern. Observer pattern is basically when one object changes state, all its dependents are notified and update automatically. For this system, when the administrator is updating the validation for payment status of the customer, this design pattern can be used. As once they update the validation for payment status, the customer will know if their parking spot is booked or not. In addition, the enforcement officer will also know

if the customer has paid or not, when they search up the parking lot and the license plate of the car parked at that lot. This is just one of the scenarios where observer pattern was used for me, more will be explained later on or in the demo video. Therefore, since a change in one object has an effect on two other user's and their actions, this design pattern is important for parts of system like this scenario.

Design Implementation: Overview

As mentioned above, the system I designed and implemented uses a factory pattern. The user of this system lands on the LandingGUI class, this is an GUI which gives user option to either to login or register. If the user has already registered and decided to login, they got to LoginFormGUI, which is dependent on the MaintainUser class which is dependent on User class for the attributes like email, password. LoginFormGUI class is also the factory component of this system. Based off the login information, the MaintainUser class takes the information and looks up the information in the user database (user.csv). Once the information that the user provided is matched with the one in the user database, then the LoginFormGUI class inside the method jButtonLoginActionPerformed determines where the user goes depending on their permission level or the attribute known as "id" in the user database. In my design, customer has permission level 1, enforcement officer has permission level 2 and administrator has permission level 3. The levels are based off, who has the most authority in the system. The permission level is set automatically when the user is registered. This allows for dynamic system, as you can add more permission levels as the system gets more roles in the system. Once the jButtonLoginActionPerformed method in the factory class LoginFormGUI, gets the user's "id" or their permission level, they will then locate the user to their respective dashboard. All my users have different options and different UI, which is easier to navigate. Once the user lands in their respective GUI class based upon their respective role, the user now have number of options they can do based of what is given in SRS.

More information about User class is it holds the attributes that will be used by MaintainUser class which is needed for both LoginFormGUI and RegisterFormGUI. Basically, for both login and registration of the user. These attributes are firstname, lastname, email and password, which are all of String type. The MaintainUser class will either take these attributes and store them in database after it gets the input from RegisterFormGUI class when registering or it will verify the existing data for these attributes after it gets the input from LoginFormGUI class when logging in. User class also has attributes loginStatus, officeID and id. loginStatus is of type Boolean and both officeID and id are of type Integer. What the significance of these last three mentioned attributes and how they are used to solve requirements will be explained later on.

Design Implementation, SRS Functional Requirements: 4.1.3 Manage Parking Officer

As mentioned above, officeID will be one of the attributes user from User class and this is one of the fields in the user database (user.csv). This will be generated when the Admin in the AddEOffGUI class or when Admin selects the add enforcement officer option. It will be

generated by using the Random class instance “rand” and peforming the action rand.nextInt(). The value of this will be set to the field officeID, which is of int type. This field will be passed to the User class constructor when new instance of that class is created inside the JButtonAddOfficerActionPerfomed method in the AddEOffGUI class. The user class will store the Offcer ID into the user database field officeID. So, this fulfils the requirement 4.1.3-REQ-1, where the system must assign a unique ID to each new parking officer added to system or user database.

Next, we have 4.1.3-REQ-2,3,4, which are if the officer exists in the system before removing them, and if their ID doesn’t exist in system and finally the system must store parking officer’s registrant information. The officer will be checked if they exists or not before being removed in the JButtonRemoveOfficerActionPerfomed method in RemoveEOffGUI class. I checked if their email and password exists in the user database, if not it will throw an error saying “Officer doesn’t exist”. So that solves 4.1.3-REQ-2. 4.1.3-REQ-3,4 are done in the JButtonAddOfficerActionPerfomed method in the AddEOffGUI class. When the admin is adding officer, the officer’s information is stored in User Database and their officer ID is unique.

Design Implementation, SRS Functional Requirements: 4.2.3 Customer Regsitration

As mentioned above, I will be taking the fields email, password, firstname and last name form the user class, when user is registering or logging in. When Customer registers they are required to input all the fields I mentioned above and in the jButtonRegisterActionPerformed method in the RegisterFormGUI class, checks if the user exists already or not by checking if email inputted exists in the User database. If it doesn’t exist, then only the customer is registered. When the customer is registered it says, “successfully registered”, this means the customer information is stored in the user database (user.csv). I only check if the email inputted exists before in the database not the first or last name, so basically met all the requirements of 4.2.3- REQ1,2,3. More about this will be explained and shown in the video.

Design Implementation, SRS Functional Requirements: 4.3.3 User Login

I have mentioned previously that my system will first have the user land on the LandingGUI class first, which is just a GUI with two options, either they register or login. Other than the admin, who already has username and password set up by me, both the customer and parking officer will need to register. The admin only registers the parking officer for them whereas the customer can do it themselves. In my jButtonLoginActionPerformed method in the LoginFormGUI class, I have made it sure that the email and password that the customer or office inputs is matched in the user database. If not, then they can’t access the system. More about this will be explained in the video. However, what I mentioned just above shows that I got 4.3.3-REQ-1,2 done.

Design Implementation, SRS Functional Requirements: 4.4.3 Book a Parking Space

For this I have made a new database, called database.csv. This basically the main database which stores all the parking information. I have used Database class for all the attributes it presents into MaintainDatabase class and that MaintainDatabase class, will be used in all of the parking related classes. Thus making database.csv the main database and MaintainDatabase one of the most significant parts of my system. Book a parking space is one of the most important requirements and with a lot of requirements. First off, 4.3.3-REQ-1, I have done this from the LoginFormGUI class itself. As mentioned above, I have an attribute in user class called the loginStatus, which tracks the customer loginStatus. Once the customer is logged in, until they log out, their login status will be true. Next off, for the Database class I have attributes, email, parkingLot, license, duration, paid, bookID, validated and time. Everytime a customer books a parking spot, all of these attributes need to be filed by the constructor of Database class. In the jButtonBookParkingActionPeformed method inside the BookParkingGUI class, where the booking happens by the customer. Inside that method, I will create a new instance of the Database class, so when I do, all those attributes listed above have to be passed. One of them is email, I am passing the email through a static method I made in MaintainUser class, where I set the email of Customer to the email they are logged in with in the LoginFormGUI class. That email is being sent to the emailString method in MaintainUser class, so when I call that method, I will get the email in which the customer is logged in with. If the customer isn't logged in, I will get a null, which basically means their parking booking won't go through. So therefore, 4.4.3-REQ-1 is fulfilled right there.

For 4.4.3-REQ-2, I used a drop-down box of parking spots upto 15, as I imagined this system as system located in one parking lot in Toronto, even though it is fully functional and capable of handling more than 15 parking spots. I just added 15 spaces by default, however the Admin and officer can add spaces if they wish to upto 7500, as required in the SRS. Anyways, the customer can chose one out of the 15 spots, once they chose a spot. The value of this is going to be stored in the instance jComboBoxParkingSpot in the BookParkingGUI. So this fulfils the requirement for that 4.4.3-REQ-2.

I created a method in MaintainDatabase class called unique, which basically returns takes in a parking spot as a parameter and it returns true if the parking spot is taken. So if this method returns true, an error is thrown in the BookParkingGUI class to the customer saying the parking spot is taken up. This fulfils the 4.4.3-REQ-3 requirement. This will also fulfil the 4.4.3-REQ-7 requirement as its basically asking to do same thing as REQ 3.

As mentioned above, the user is required to input their desired parking spot, they will also have to input the duration as well as their license plate. I added the license plate as in the real life, license plate is used by enforcement officer to ticket the person and send it to their home based of their license registration. In addition, I asked the customer to input the duration of their stay at the parking spot in minutes, which will be stored to the main database along with other information of the booking. This fulfills the 4.4.3-REQ-4 requirement and 4.4.3-REQ-5 requirement.

Finally, in the Database class one of the attributes that is there is the bookId, which is used in this method of jButtonBookParkingActionPeformed method inside the BookParkingGUI class. Like officer id I used the same Random class instance rand and did rand.nextInt(7500), since there are 7500 parking spots potentially, to generate a random id. Which will be associated with this specific booking by the customer. Since the bookId is attribute of Database class, it will be passed into that class's constructor when doing booking along with other attributes metioned above like, email, parkingSpot, license, duration, etc. So this fulfils the 4.4.3-REQ-6 requirement, meaning all the requirements in this section are now complete.

Design Implementation, SRS Functional Requirements: 4.5.3 Cancel a Parking Space

This is fairly straight forward requirement similar to how I add to the database using MaintainDatabase class and the main database (database.csv), I can remove the booking the same way. The implementation of this function in the system is done same way as if it would for booking. But removing is obviously easy as you don't have to create a new instance of Database class which has all the attributes you need. So, this function for the customer is done in the CancelBookGUI class and in the method jButtonCancelBookingActionPerfomed method in that class. So for this function to work, the customer must input their booking id, which fulfill the 4.5.3-REQ-2 requirement. They will also have to include their parking spot and their license. Their email is associated in this case, the booking id, license, and parking spot as well as the email has to be matched with the database records. If not, the booking can't be cancelled. So this fulfils the 4.5.3-REQ-1 requirement. The 4.5.3-REQ-3 requirement is also fulfilled here as the expiry time will also be taken in to consideration before cancelling the booking.

Design Implementation, SRS Functional Requirements: 4.6.3 Payment

Payment is one of the most important aspects of the system as customers can book but if they don't pay they are basically owning the parking spot for free. This is not how the world works, you have to pay to use someone else's property. So the implementation of the payment system is important. Payment will be done in the PaySystemGUI class in the jButtonPayActionPerformed method. Where the email of the user when logged in, is passed to this method and class using the static method emailString in MaintainUser class. When the user is logged in their email gets passed to this emailString method and saved there until they are logged out. During this logged in time their loginStatus is true, this is also set in the user.csv database as mentioned above. The email which they logged in with and the one that is associated with their booking in the database.csv database, is the email which the method checks to see is the same. The user has to also input their email in the payment system gui, so that email input will also be checked to verify if they are part of the system and logged in. So 4.6.3-REQ-1 requirement is fulfilled as the customer needs to be logged in before they pay.

I have made the GUI in a way it has multiple payment methods the user can select visa, mastercard, credit/debit or paypal. This will fulfil the requirement 4.6.3-REQ-3. Customer is also

required to input their parking space for sure, and they will have to input their license as verification to cross check their license and parking spot as well as their as mentioned above matches that booking field in the database.csv database. So this fulfils the requirement 4.6.3-REQ-2.

I have added a timestamp of payment for the customer using the java class LocalTime(), which gets the current local time. And I saved this time to the customer's booking field in the database.csv, in the column "time". This fulfils the requirement 4.6.3-REQ-4, as the timestamp is exactly when the customer enters details and presses pay.

It will also start countdown till expire in the backend the countdown can be viewed in the ViewBookingGUI class. This fulfills the requirement 4.6.3-REQ-5. 4.6.3-REQ-6, 7 are also taken care of those requirements have been fulfilled.

Design Implementation, SRS Functional Requirements: 4.7.3 View Bookings

Customers can view their bookings in the ViewBookingGUI class, when they are logged in they are brought to CustomerGUI, which has a button called view booking. After clicking that button the customer can now go to the class ViewBookingGUI. In that class I have allowed the customer to view their booking details that are stored in the main database (database.csv), which is associated with their email that they logged in with. Their details include, email, parking spot, license, pay time stamp, payment status, confirmation status, booking id, duration. So 4.7.3-REQ-2,3 requirement is fulfilled.

Parking enforcement officers can enter the license and parking spot of the customer and they can see all the details like payment status, pay time stamp, their booking Id and duration. They can then grant or cancel access if they didn't pay based of that, this will be explained shortly. But this basically fulfill the 4.1.3-REQ-1 requirement. The admin can also view customer's details like their payment details but not full details like duration, booking id, etc, like the parking officer.

Design Implementation, SRS Functional Requirements: 4.8.3 Manage Parking Spaces

The parking officer as well as the system admin can both add or remove spaces in the system. AddSpaceGUI and RemoveSpaceGUI classes both have code in the back end which check if the parking spot is vacant then only it will add the space or remove the space if the user is already there. This fulfils the 4.8.3-REQ-1,2,3,4. As the system will have at least one parking spot also, this is designed in the MaintainDatabase class itself. The parking officer can also grant request to the customer for their booking if they have paid and are in right spot or else cancel the request if they haven't paid yet. Which will delete the booking from the database.

Design Implementation, SRS Functional Requirements: 4.9.3 Change Payment Status

For these requirements, I made a class called ValidPaymentGUI, which has two methods, one of the lets the system admin to view the customer's booking details like whether the customer has paid already or not. It will also display the customer's booking id and license. The admin will need to input the parking spot and the customer's email to view the details first. Then if the customer hasn't paid, the system will advise the admin that the customer didn't pay and it isn't a valid payment. Which then the customer can do nothing, as the default value when the customer books their parking space for the field paid in the Main database (database.csv) is false. However, if they have paid, then the admin will have to press on the validate and this will validate the customer's payment and will confirm the booking of the parking spot for the customer. In future implementation, I can send an email to customer once this button is pressed. So, for this basically only admin can do this, although parking officer can also grant access based of if they paid or not. But for validation of the customer's payment status, that is only admin's job. This fulfils the 4.9.3-REQ-1 requirement. For the customer's existence, well the system admin enter the email of the customer so that it self confirms the existence of the customer. So 4.9.3-REQ-2 is also fulfilled. As far as the occupancy and the payment, as mentioned above the admin can see customer's bookID and payment status. Booking id is only assigned to an customer if they have booked already and payment status is updated when user paid as mentioned above, if they paid it says true. Therefore, both 4.9.3-REQ-3,4 requirements are both fulfilled.

Design Changes from Midterm:

The design for this is quite similar to what was done in the midterm, for example the design pattern I used is exactly the same. Where I mentioned I would use a factory design pattern in the midterm, and I had used the same here also. However, one of the most visible differences is the fact that I made a lot more classes and 90% of those classes being GUI. The reason why I did way more classes and more GUI classes specifically is because of organization. I could have made some non-GUI classes like Database or User, but I chose not to because I have a lot of GUI classes and almost of all of them are unique in their own way. Mainly because we are dealing with 3 different users, their dashboard is different, and their tasks are different. One thing maybe I can do different next time is the fact that I did perform the same tasks in some of the action performer methods in classes like registering users or viewing parking spaces that the user has booked. Or even cancelling parking spaces. I have used the same functions for these tasks and all three users are allowed to most of these things. So, I could have built some more factory classes and implemented them to GUIs. I have tried this but couldn't perfect it due to most of the methods and classes being task specific. Meaning if the customer is cancelling a booking, I have a created a class called cancel a booking, so the customer will head to that specific GUI. This is easier and more organized from both programmer perspective and user perspective, which is why I stuck with my way of building the system and the design I redid for the one I implemented. My system would be used in an **Agile** software development life cycle as that is also the life cycle in which I used. It was fast for me to

implement this and I made a lot of changes to my design along the way, and it was easier to fix as I implemented stuff. If I had stuck with my midterm design it would have been waterfall SDLC, which would have not allowed me to complete this fully functional system on time. The demo of this project is once again will be shown in my video, the link is at the top of the report.