

# PARKING SYSTEM

AMANPREET DHIRAJ

EECS: AMAND25 | STUDENT #: 215-080-716

## Video Walkthrough:

Here I have went over all the functionality meeting all the requirements the application must do. Along with this video I have also added details on each screen in the report below.

Link : <https://drive.google.com/drive/folders/1BQ49OclDKY3mtlEysU83m5A81ofg8uC5?usp=sharing>

## Introduction

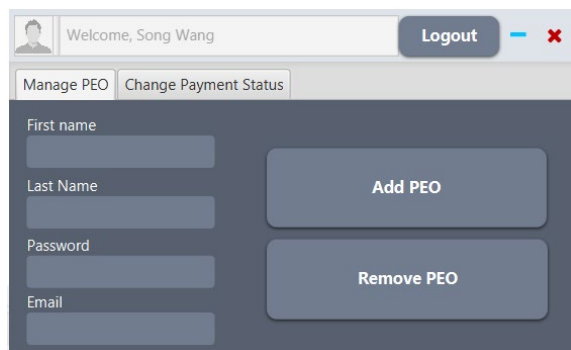
Toronto Parking Authority needs a new system in which a customer and police service enforcements can distinctively use the system. As designer and developer, it must pass the SRS specifications. Using design patterns; a standard solution to a common software problem in context; we can use many of the design patterns to construct the over all functionality and accessibility of this project. Furthermore, in this report you will see some major and some minor changes to the over all design and functionality when compared to the midterm.

The creational pattern has many patterns such as Singleton, Builder, Prototype, Factory, etc. One of the patterns that would be useful in the implementation of the program/software would be Factory Pattern. Using factory pattern returns an instance of one of several possible classes depending on the data that is given or provided. With the SRS, there must exist a user where it can have subclasses such as customer and PEO. With this in mind, we can use the factory method to decide which of these subclasses to return depending on the arguments that are specified. Furthermore, the Abstract Factory pattern can be used to support multiple user interfaces, such as customer, admin, and PEO graphical user interface. This way we can utilize the abstract factory pattern to tell the factor to return the GUI factors that are needed to be rendered.

Other pattern that was used in this project were iterator, state, and observable design patterns. These patterned allowed the program to have information at its hand and will be further discussed when it's used to fulfil certain requirements.

### 4.1 Manage Parking Enforcement Officer

This feature allows a *system administrator* to add or remove *parking enforcement officers* from the system. When an officer is added, a unique ID is assigned. To tackle this requirement, an admin user information is in the system at default with permission level 3

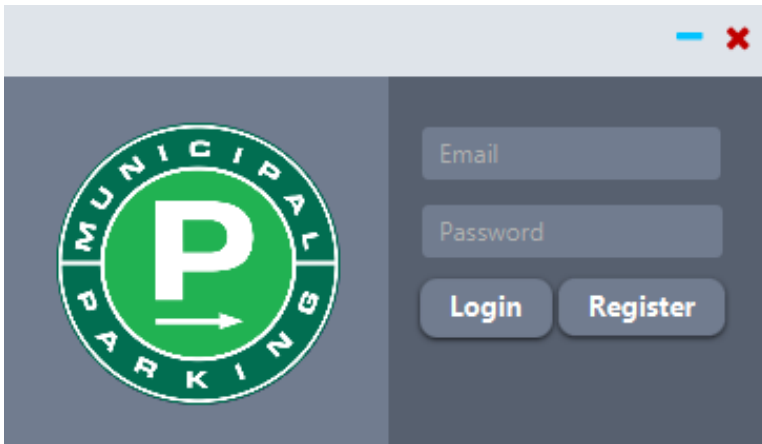
The screenshot shows a web application interface for managing parking enforcement officers. At the top, there is a user profile section with a placeholder icon, the text "Welcome, Song Wang", and a "Logout" button. Below this, there are two tabs: "Manage PEO" (which is active) and "Change Payment Status". The main content area is divided into two columns. The left column contains four input fields labeled "First name", "Last Name", "Password", and "Email". The right column contains two large buttons: "Add PEO" and "Remove PEO".

(admin level). This information is stored in the "user.csv" file where all the user critical data lives. The Unique ID that each parking enforcement officer has when added; is the same the as the email. In other words, the email id plays as the primary key to each customer and PEO. The admin proceeds to add PEO information such as first name, last name, password, and email. One of the additional fields that was added as a design choice was password field. Although a default password of "password" can be set, I felt it would be much informative

for the admin to make the password for the PEO and cause less confusion. When the admin, clicked the “Add Peo” button, a method is invoked called “AdminAddPeo()”. This function checks in the system if the officer’s ID is not already existing in the database “!db.checkRegister”. If presented when adding PEO, the system alerts the admin of existing user. Furthermore, once the validation is passed the system does a database call to add the PEO by “db.RegisterPeo”. There were not many changes made from my initial design when tackling this requirement. Although there were some additional variables such as, email used to represent the PEO’s Unique ID and along with UUID for other functions.

#### 4.2 Customer Registration

This feature allows *customers* to register a new account in order to use this application. Implementation of this requirement were strictly implemented as the initial design. With one change of not having an “username” but rather “email” option for unique ID. This way the



system can have multiple users with the same first and last name. As seen in the picture, the program presents the user to a login where they can click “register” for the first time. One thing to emphasize is that, when the user registers it is given a default permission level of 1. This is the level for customers, whereas, level 2 would be for parking

enforcement officer and level 3 would be for admin. Although, PEO can not register from this page and is highly protected with the permissions. Furthermore, the system also checks for existing users in the database when the “Register()” method is called inside the RegisterMain class.

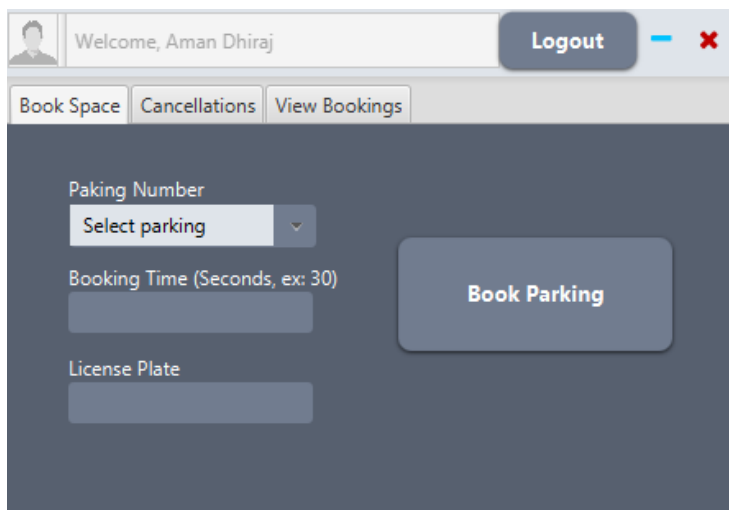
#### 4.3 User Login

This feature allows the user to sign in, to access the application. The implementation of this design and requirement were also trick followed as initially presented in the midterm. The customer, police enforcement officer and admin can all login from this same login panel. As mentioned before the permission level plays an important role when rendering the GUI for each user and since each permission level decides on the render, a Factory Pattern was used to achieve this. The system validates the login and registration to see if the user is in the database by invoking the method “db.checkLogin()”. Further detailed in the “LoginMain” class, you can see how each GUI is rendered accordingly to the permission level. Once the user logs in, their “loginStatus” is changed to “true” for further use. This information is constantly updated by the program and sets the user’s login status to false once they either logout or close the program itself. This information is stored in the “users.csv” file.

#### 4.4 Book a Parking Space

This feature allows a *customer* to select which parking space, and how long they want to book it for. With this Implementation there were some major changes to my initial design. With my initial design I designed the UML in such way that the parking information was also part of the User class. However, this was changed in the current additions to make relations of the data easier to access and set. This new class was named Parking which hold majority of the program's attributes. This includes, uuid, parkingNumber1, parkingNumber2, parkingNumber3, licenseNumber1, licenseNumber2, licenseNumber3, parkingOne, parking1PayStatus, parking1Timestamp, parkingOneTime, parkingTwo, parking2PayStatus, parking2Timestamp, parkingTwoTime, parkingThree, parking3PayStatus, parking3Timestamp, parkingThreeTime, count. All these attributes build up the functionality of the parking system and essentially the backbone for the data. Located in the "parking.csv" a further detailed usage can be seen when the user is using the program.

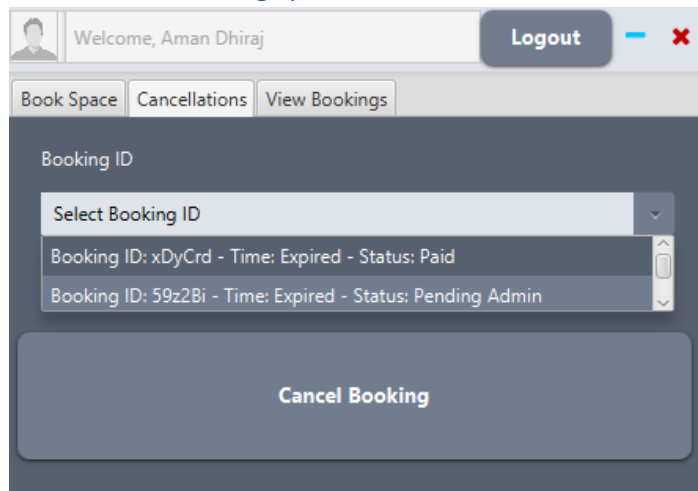
As mentioned before the login status of a user is kept along each GUI which ensures that the user is registered and logged-in before booking a parking space.



The user can choose a parking spot by selecting the drop-down menu. The parking spots are dynamically changing and hold at minimum of 1 parking spot. An observable pattern was used to keep the list up to date when new changes are made by the PEO and invoked by the method "updateList()". As per requirements, the customer is presented with an error message when the parking spot selected is occupied. For the sake of this program and overall realistic

expectation the booking time was recorded in seconds to mimic a real time scenario. The system keeps a record of each occupied parking space along with its important information. The user can only book up to 3 parking spots and if the customer exceeds this limit, an error message is presented to the user. Moreover, the user is also presented with an error message if the parking spot is currently occupied. Each of these errors are presented when the user clicks "Book parking" and the method "customerBookspace()" checks each of these requirements before executing the method "parkDB.addNewParking()". Furthermore, once the database method is invoked the system enters the information accordingly to what the customer inputs. "genNums(int n)" method is used to generate a unique booking id of n size. In this program a length of 6 chars was used to generate a bookingID along with status of the booking as "not paid".

#### 4.5 Cancel a Parking space

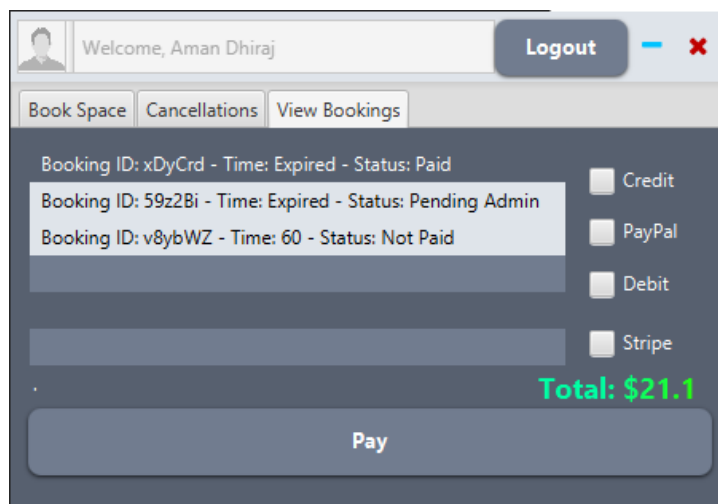


This implementation of the requirement was same as the initial design. The customer must be logged in and register and this is validated thought of the application with a state of login status. As seen in the image on the side, the customer can see the current booked parking where the customer can select any of the booked parking and cancel the parking by clicking the cancel booking button. Once the button is clicked,

“customerCancelSpace()” is invoked and updates the database and removed the current booked parking from the user’s data. This only happens if the time status of the booking is not expired otherwise the user is presented with an error, they cannot cancel the parking.

#### 4.6 Payment & View Bookings

This feature shows a *customer’s* current parking booking. Information such as expiry time and payment status can be viewed and allows the *customer* to pay for their currently booked parking space. The overall implementation was heavily changed from what initial stated in the midterm. One of the differences is that I added observable list which keeps track of the selected



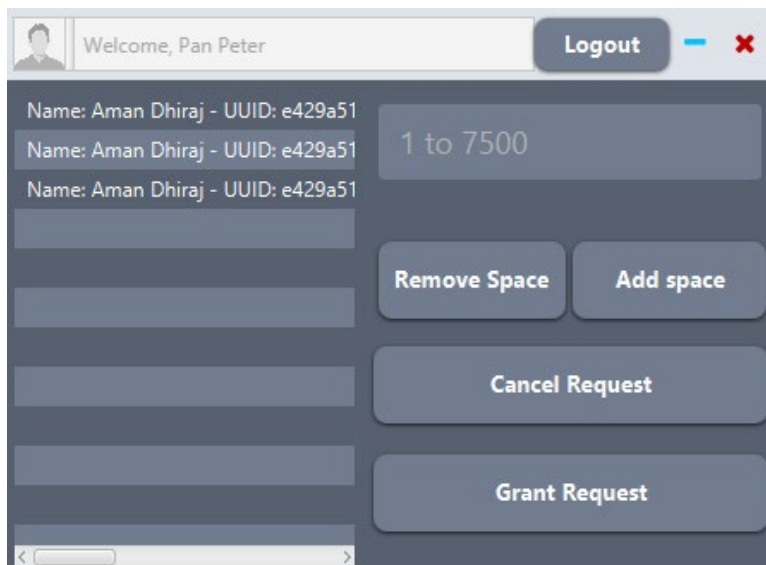
items and gives the users of the total amount the customer has to pay. This calculation is done the method “initData()” which initializes all the data this GUI needs along with the observer. One of the design choices I made is to have the list display all the information that is needed for the user to view their bookings but also when the user clicks pay, the list is scrapped for the data, so the user does not have to enter he is parking space number. It is also assumed once the user clicks

any of the payment methods, they do not have to enter their banking information and its is already assumed to exist in the system. As always, the system keeps a state of the user’s login status. The user is not presented with any list of data if they have not yet made any bookings, so they cannot just randomly pay for a parking. Once the user selects the spaces to pay (can be multiple) and clicks the “pay” button, the system automatically assigns a timestamp to each of the purchase along with the system automatically opening a new thread to each parking time.

The “countDown(int time, String bookingID, String UUID, int interval)” is invoked to start a count down till the user's booking time. Once the timer has reached the expiry time (in seconds), the system alerts the user that one of their parking has expired and the observer list updates this live on the customer's view booking tab.

#### 4.8 Manage Parking Spaces

This feature allows the *parking enforcement officer* to manage parking spaces by adding or removing them. In this requirement there were not many changes made from the initial design in the midterm. In fact, there were slight additions to what I have visualized for the program, such as using additional pattern, “iterator pattern”, to integrate through the Users object to present the data in a list.

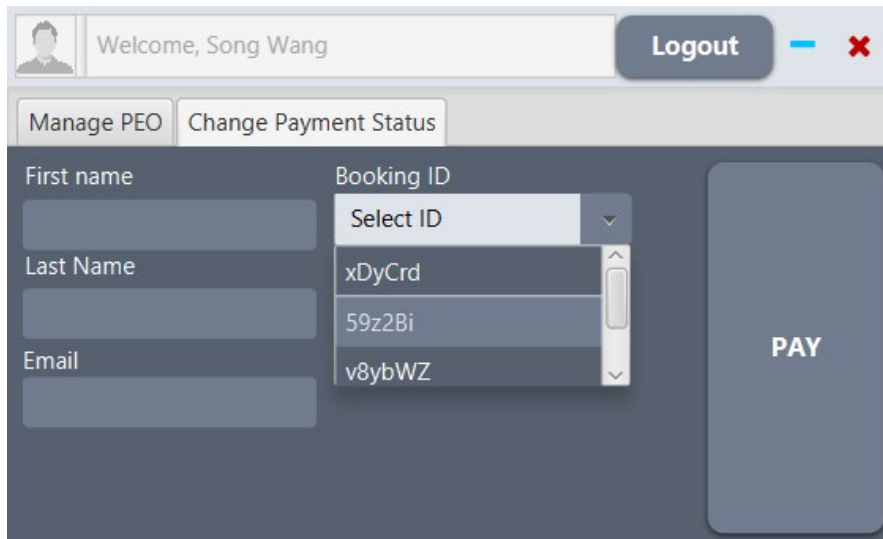


As seen in the image, the PEO can see all the current booked parking spaces along with additional information, such as payment status, parking time etc. One of the design choices that I made in this GUI is to simulate the “manage parking” button as the initial starting page. This way there is no button that leads to this page. The PEO can add spaces either in increments of 1 or choose to enter for example 23 which will add 23 additional

spaces to the system along with the system keeping track to have maximum of 7500 parking spaces. If this number is exceeded the PEO is presented with an error message and an error message is displayed if the parking space is lower than 1. The “parkingspaces.csv” keeps track of the parking spaces for the system to use accordingly.

The parking officer can also remove spaces by add a valid parking number in the textbox. The system verifies that this parking spot is vacant before the removing the space from the system which is handled by the method `removeParking()`. Furthermore, the PEO can grant and cancel request by selecting a customer from the list and each of the functions will parse the data from the list and make requests accordingly to the database.

#### 4.9 Change Payment Status



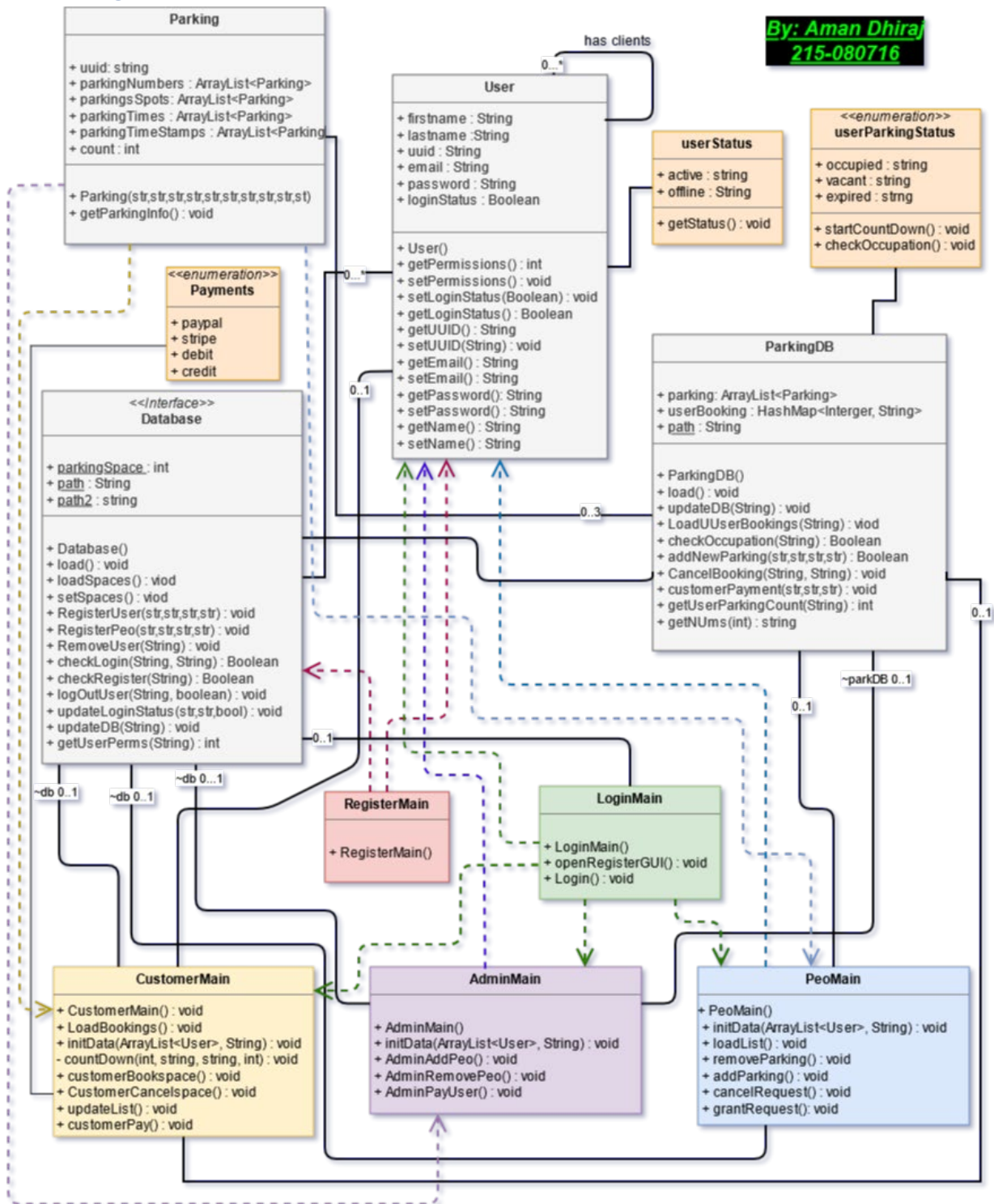
This feature allows the *systems administrator* to update *customers'* payment status. This implementation was strictly followed as my initial design and there were no differences. The system always has a user in the `“user.csv”` file where it holds a system admin login detail. The user had a permission level of 3 which allows

the system to render the admin GUI once authenticated. In this demonstration the admin email is `“songwang@yorku.ca”` and password is `“admin”`. One of the design changes that I made to this requirement is that instead of validating the users existing in the system, the system only lists the booked parking of the users that are already part of the system. This also allows for the system to keep track of only occupied parking space and change status accordingly. The system admin can enter the persons details such as first name, last name, and email. The system also checks if the email/customer exists if the system before making any changes. The admin must select the booking id for that user whose email relates to each of the booking id. This is assumed that the admin knows what they are doing and have the right details when changing the status. `“AdminPayUser( )”` is invoked when the admin clicks `“pay”`, and the system updated the customer’s payments accordingly.



# UML Diagram

By: Aman Dhiraj  
215-080716





## Testcases & Coverage Results

**Test Cases (Left Pane):**

- ParkingTest [Runner: JUnit 5] (0.261 s)
  - test\_AddPeo() (0.158 s)
  - test\_SetSpaces() (0.007 s)
  - test\_user\_Exists() (0.003 s)
  - test\_AddUser() (0.007 s)
  - test\_removeUser() (0.007 s)
  - test\_pdb\_addParking() (0.041 s)
  - test\_removePeo() (0.011 s)
  - test\_registration() (0.002 s)
  - test\_login() (0.004 s)
  - test\_perms() (0.020 s)

**Coverage Table (Middle Pane):**

Element	Coverage	Covered Instruction...	Missed Instructions	Total
3311_Final	33.1 %	1,552	3,132	
src	33.1 %	1,552	3,132	
Customer	0.0 %	0	948	
CustomerMain.java	0.0 %	0	948	
CustomerMain	0.0 %	0	795	
PEO	0.0 %	0	687	
peoMain.java	0.0 %	0	687	
peoMain	0.0 %	0	687	
application	73.9 %	1,325	468	
ParkingDB.java	76.5 %	675	207	
ParkingDB	76.5 %	675	207	
Parking.java	69.4 %	206	91	
Parking	69.4 %	206	91	
Database.java	81.1 %	368	86	
Database	81.1 %	368	86	
Main.java	0.0 %	0	43	
Main	0.0 %	0	43	
User.java	65.0 %	76	41	
User	65.0 %	76	41	
Admin	0.0 %	0	461	
AdminMain.java	0.0 %	0	461	
AdminMain	0.0 %	0	461	
Auth.Login	0.0 %	0	311	
LoginMain.java	0.0 %	0	311	
LoginMain	0.0 %	0	311	
Auth.Register	0.0 %	0	257	
RegisterMain.java	0.0 %	0	257	
RegisterMain	0.0 %	0	257	
(default package)	100.0 %	227	0	
ParkingTest.java	100.0 %	227	0	
ParkingTest	100.0 %	227	0	

**Console Output (Right Pane):**

```
<terminated> ParkingTest [JUnit 5] C
in space=1
adding first entry
Cancelling Parking 1
```

The above testcases and coverage results are test with all the possible methods that can be accessed. Other 66.9% of the test cases require user's input or/and multiple entries, which cannot be tested. I hope this coverage shows the main features.