CSE 564: Software Design

Spring 2022

PARKER: AUTOMATIC PARKING ASSISTANT

Name 1	Name 2	Name 3	Name 4
Parvakumar Patel	Parth Miyani	Prayag Patel	Sudhanva Rao
Program of Study	Program of Study	Program of Study	Program of Study
MS in CS	MS in SE	MS in SE	MS in SE
School of Computing and Augmented Intelligence			
Arizona State University, Tempe, AZ, USA, 85281			

	Parvakumar Patel	Parth Miyani	Prayag Patel	Sudhanva Rao		
Parts	**% Effort**	**% Effort**	**% Effort**	**% Effort**	**Points**	**Earned Points**
Problem description	25	25	25	25	5	
SRC design specifications with description	30	20	30	20	25	
UML design specifications with description	25	25	25	25	25	
Implementation	30	30	20	20	10	
Experiments & results	35	25	25	15	10	
Conclusions	25	25	35	15	5	
Demonstration	25	25	25	25	10	
Presentation	25	25	25	25	10	
Code quality	30	30	20	20	5	
Report quality	25	25	25	25	5	
Total					110	

Table of Contents

1 PROBLEM DESCRIPTION 4

2 DESIGN 7

2.1 SRC DESIGN SPECIFICATIONS WITH DESCRIPTIONS 7

2.1.1 Application Component 7

2.1.1.1 Login Component 7

2.1.1.2 Signup Component 10

2.1.1.3 QR-Generator 12

2.1.1.4 Scanner Component 14

2.1.2 Server Component 16

2.1.3 Parking Spot Component 20

2.2 UML DESIGN SPECIFICATIONS WITH DESCRIPTIONS 22

2.2.1 Class diagrams 22

2.2.1.1 Parking 23

2.2.1.2 ParkingSpot 23

2.2.1.3 User 23

2.2.1.4 Client 24

2.2.1.5 Server 24

2.2.1.6 QR request 25

2.2.1.7 Sensor 25

2.2.1.8 Scanner 26

2.2.1.9 Card 26

2.2.1.10 Visit 26

2.2.2 Use-case diagram 27

2.2.2.1 The User: 27

2.2.2.2 The Scanner: 28

2.2.2.3 The Server: 29

2.2.3 Activity Diagrams 30

2.2.3.1 For the user entity 30

2.2.3.2 For the Scanner entity 31

2.2.3.3 For the Server entity 32

3 IMPLEMENTATION 33

3.1 PARKER PROTOTYPE 33

3.2 ALGORITHM AND WORKING OF THE PROJECT 35

3.2.1 Algorithm 1: Algorithm of System Operations. 36

3.2.2 Algorithm 2: Update spot table 36

3.3 Application Component 37
3.3.1 qr_scanner [Lines of code : 256] *37*
3.3.2 api_call [Lines of code : 30] *37*
3.3.3 networking [Lines of code : 31] *38*
3.3.4 authorization [Lines of code : 90] *39*
3.3.5 user_home [Lines of code : 200] *40*
3.4 Server Component [Lines of code : 505] 40

4 EXPERIMENTS AND RESULTS 43

4.1 User is not signed up. 43
4.2 User enters the wrong password. 43
4.3 User successfully signs up or successfully logs in. 44
4.4 User generates QR code. 44
4.5 Admin/scanner logs in. 45
4.5.1 Entry scanner *45*
4.5.2 Exit scanner *45*
4.6 User can enter. 45
4.7 User can exit. 46
4.8 User cannot enter/exit. 46
4.9 Parking is full. 46

5 FRAMEWORKS AND SOFTWARE TOOLS 47

5.1 Software Tools 47
5.2 Frameworks 47

6 CONCLUSION 49

6.1 Code Quality 49
6.2 Report Quality 50
6.3 Evaluations 50

7 REFERENCES 51

A APPENDICES 51

1 PROBLEM DESCRIPTION

Smart Parking Assistant (SPA) is a Smart Parking System that provides a parking strategy that combines technology and human ingenuity to utilize as few resources as possible—such as fuel, time, and space—to enable faster, easier, and denser parking of automobiles during the majority of the time they remain idle.

Figure 1. Man searching for parking the traditional way

Congestion and car circles increase vehicle emissions and damage air quality. Consider the amount of time spent circling, driving, and waiting for a parking spot each day, as well as the environmental cost. This has a knock-on effect in the surrounding areas, contributing to urban traffic congestion. As a result, it's vital that your car parking system be as efficient as possible, allowing vehicles to find a spot quickly and easily.

In our Smart Parking Assistant system, every parking spot in the parking structure will be equipped with a sensor that will provide information about whether the parking spot is vacant or occupied. This data will be provided to the system in a continuous loop. The system consists of a mobile app (Android/iOS) that will feature user information such as personal details, automotive details, payment information, past visits and provides an option to produce a QR code. The user will have to scan this QR code with a barcode reader located near the building's entrance and exit gates.

When a car approaches a parking structure, the user can now use the smartphone app and login to the account. Once logged in, the user needs to add a card for payments. At Least one card is necessary for the application to provide parking to the user. Once done, the user can go ahead and generate the unique QR code. Showing this QR code to the placed scanners at the entry and exit gates of the parking, the user's QR code gets validated, a quick and efficient algorithm is run in the backend for fetching the nearest available parking spot from the data received from the proximity sensors placed at each and every parking

spot. Once validated and calculated, the user is provided with the exact location of the parking along with the route to the same location. The user can easily follow the guidelines provided by the application and reach the desired parking spot which is guaranteed to be the nearest one available. The user can now go ahead without any concerns of generating a parking ticket or getting in line in order to get any kind of token. The process of entry was as quick as parking the car in a personal parking area. Without any hassle of standing in a line for getting a spot or for generating a token, the user easily reached the parking area, conveniently opened an application by just sitting inside the car, scanning the QR code and just reaching the parking area smoothly. This process of letting vehicles enter the parking is so quick and hassle-free that there is almost null possibility of queue generation or vehicles getting stuck for entering the parking or getting spots.

Not just the process of entering, the process of exiting is also smooth as butter. The user does not need to provide any kind of pre-defined return time or exit time. They can take as much time as needed, run their errands and complete every possible task. On returning, the user again needs to open the application, login to the account if already logged out, go to the exit gate and just scan the QR code again. Our application will instantaneously determine the situation of the user, validate the user, will fetch the entry time of the user from the database and calculate the fare that is needed to be paid according to the predefined rates and without any kind of intervention from the user. The user will be prompted with the fare that is needed to be paid and once the payment is processed successfully, the exit is opened and the user is free to go. Once again, the process was so quick and hassle-free that users didn't even have to wait in any queue and didn't even have to get out of the car for any payment. Everything was handled by just sitting inside the car and performing a couple of steps in the application. This is the power of our application as there are a lot of people who face problems finding parking and have to wait and wait at every entry and exit gates in order to grab a parking spot. **Every parking problem of a common man is solved by just installing one application, Parker!**

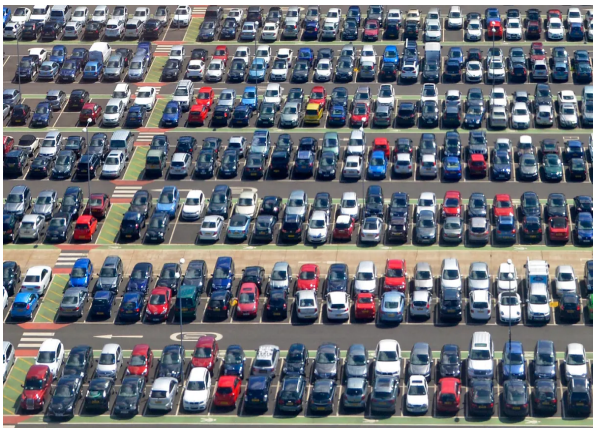


Figure 2. A systematically organized parking space

When the user is provided the access to enter the parking, the data for the parking spot, such as the level number or spot number, or both, will be shown to the user through the mobile application. The app will also have a navigation system that will guide users to their assigned parking spots. Being a cyber-physical system, we can also develop a technology that will communicate with a motor or actuator, making it easier

to open the barriers at the gates. During departure, the user will have to scan the QR code with the scanner at the exit. At this point, the fare will be calculated, and the customer will be charged using the credit/debit card information supplied on the app. After the payment is completed, the system will send another signal to the actuator at the exit gate, which will open the barrier and allow the car to leave the location.

One of SPA's aims is to reduce the amount of time and effort necessary to find a parking place. The ability to accurately steer an automobile to an accessible location has various environmental benefits; it reduces CO_2 emissions, noise, and other pollutants. Smart Parking and Smart Environment can be used in conjunction to monitor air quality and parking availability. Second, driving around the parking lot looking for available parking spaces may be inconvenient, especially during rush hour. If a consumer cannot find a parking place, they may lose their jobs or customers may decide to shop elsewhere and hence affecting businesses. The capacity of a customer or visitor to quickly identify a location reduces friction and enhances the overall experience. The convenience factor is critical for disabled drivers, public service vehicles, and emergency vehicles. Also, there are many additional benefits like real-time insights and data. Smart Parking provides broad data sets that may be used to uncover trends, peak hours, and other metrics for forecasting and reporting by a local government, car park operator, or business. Using specialized software, the data and sensors may be integrated into city administration systems or MI reports. Such unique and new business concepts like smart parking bring up the possibility of new business models that would otherwise be unattainable due to technical improvements. Reward systems, app-based payments, and dynamic parking costs are just a few examples. The overhead would decrease to a certain extent. Previously, on-street parking may have involved the purchase of parking meters or the hire of parking inspectors. Smart parking technology can reduce these expenses by automating operations and providing targeted enforcement efforts.

Thus, when a driver knows exactly where they need to go and where they need to park, it reduces idling and unnecessary driving – therefore optimizing traffic flows in built-up parking areas, eventually reducing CO_2 emissions, saving fuel, reducing congestion during peak hours, and benefiting businesses by increasing the customer base.

Figure 3. Proposed solution for the parking problem with the help of sensors and interactive UI application

2 DESIGN

2.1 SRC design specifications with descriptions

Parker is a cyber-physical system that would comprise numerous components that would combine their functionalities together and create a highly efficient and easy-to-use application that would definitely ease the parking process of customers and help them quicken the process. Parker, being a cyber-physical system, can be divided into 3 major synchronous reactive components. Combined, these reactive components create the whole functionality of the application and help us understand the flows and functionalities of the project. The synchronous reactive components that will constitute the behaviour of this application are:

a. Application component
b. Server component
c. Parking spot

These synchronous reactive components will be containing many components inside them as sub-components for easing the understanding process of this product.

2.1.1 Application Component

This component depicts the complete application that we have created. All the functionalities of the application will be covered in the description and the SRC specifications of this component. To achieve the goals of this project, we have created an application that helps the client and scanner to communicate with each other and the server. As shown in Figure 5 below, there will be 4 sub-components for the application component of our project. These subcomponents would contribute to the maintenance of the main SRC by performing their individual routines in order to authenticate the user and move ahead with generating the QR code or scanning the QR code.

When the clients want to park a vehicle, they need to open our application and there will be a prompt that asks them to log in or signup on the basis of their past transactions. If the user has used our parking facility previously, then they must have logged in earlier and have created an instance in our database. So, the next time they come for parking, they don't need to sign up again. Just providing their email and password would be sufficient for us to recognize the user.

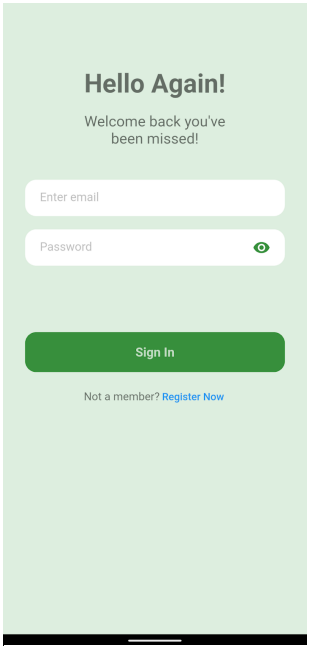


Figure 4. In-app screenshot of login page of the Parker Application

2.1.1.1 Login Component

It is necessary for the user to log in to our application using a valid email address and password. The **login component** takes two input strings i.e. **email** and **password** from the user as input. The user can enter the email and password using a highly interactive UI created by us in our application. These strings will be provided as an input to the login component. The login component will validate the user's email and password by fetching the details from the firebase database and checking if the user's email already exists in our database. The screenshot of the UI of login page is shown in Figure 4.

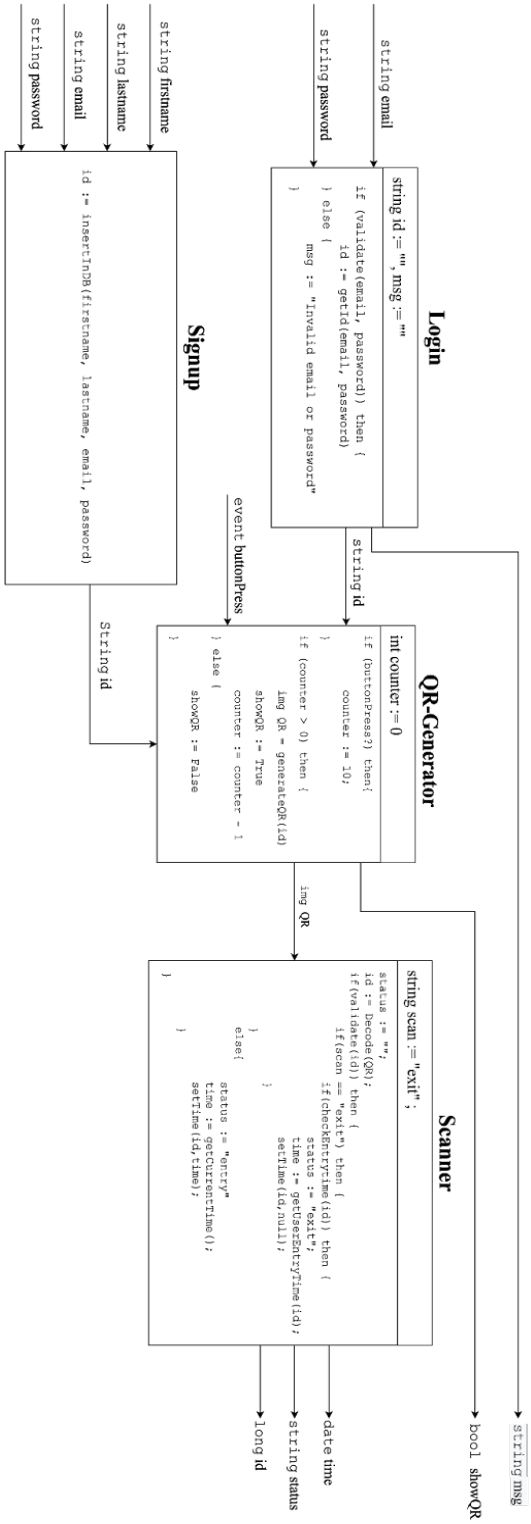


Figure 5. SRC for Application Component

a. If it exists, that means the user already signup earlier and doesn't need to do that again. We can fetch the user's details from the database. After fetching the details about the user from the database, we check the password provided by the user as input.
    - If the password of the database and the user-provided password exactly match, the user is considered to be valid and we redirect the application to the dashboard.
    - If the passwords don't match, the user is invalid and we can't let the user enter. In that case, we prompt the user with a message saying the password is invalid.

b. If it doesn't exist, the user must not have signed up earlier in our application. In that case, the user will have to sign up and the app will redirect to the signup page with a prompt that says that the user does not exist in the database.

Input, I = {string *email*, string *password*}.
Output, O = {string *id*, string *msg*}.
State variables, S = {string *id*, string *msg*}.
Init = {*id* := "", *msg* := ""}.

Inputs of the Login			
Name	Range of values	Initial value	Unit
email	[a-z]+@[a-z]+.[a-z]+	N/A	N/A
password	[a-z A-Z 0-9]+ | {6,}	N/A	N/A

Outputs of the Login			
Name	Range of values	Initial value	Unit
id	[a-z A-Z 0-9]*28	N/A	N/A
msg	{"Invalid email or password"}	N/A	N/A

State variables of the Login			
Name	Range of values	Initial value	Unit
id	[a-z A-Z 0-9]*28	""	N/A
msg	{"Invalid email or password"}	""	N/A

React:

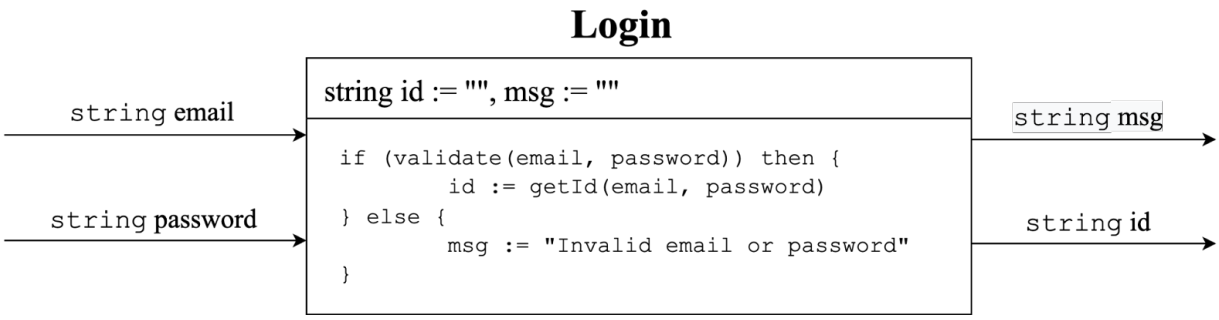


Figure 6. SRC for Login Component

2.1.1.2 Signup Component

The **signup component** handles any new user who wants to register for an account for our application. Our application will validate the syntax of all the strings i.e. whether the user entered the email address of valid format (*abc@xyz.com*). After validation of all the strings, the signup component will take 4 inputs i.e. **first name**, **last name**, **email,** and **password**. Figure 7 shows the signup page of our application. The component checks for the existence of the email address provided by the user in the database. If the email already exists, the user can't signup using an already existing email address. Otherwise, the 4 inputs will be sent to the database to insert a new user, and if the user is inserted successfully. The UI for the signup page is shown in the Figure 7.

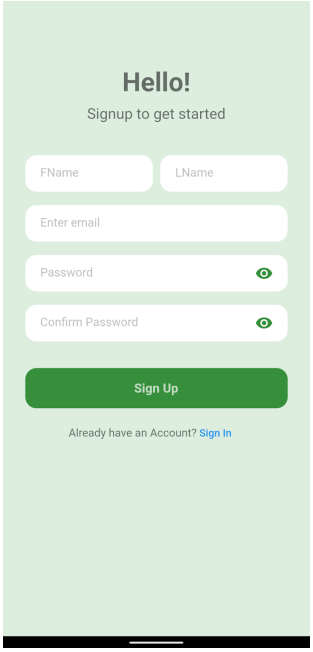


Figure 7. In-app screenshot of signup page of the Parker Application

Input, I = {string *firstname*, string *lastname*, string *email*, string *password*}.
Output, O = {string *id*}.
State variables, S = {}.
Init = {}.

Inputs of the Scanner			
Name	Range of values	Initial value	Unit

firstname	[A-Z a-z]	N/A	N/A
lastname	[A-Z a-z]	N/A	N/A
email	[a-z]+@[a-z]+.[a-z]+	N/A	N/A
password	[a-z A-Z 0-9]+ | {6,}	N/A	N/A

Outputs of the Scanner			
Name	Range of values	Initial value	Unit
id	[a-z A-Z 0-9]*28	N/A	N/A

React:

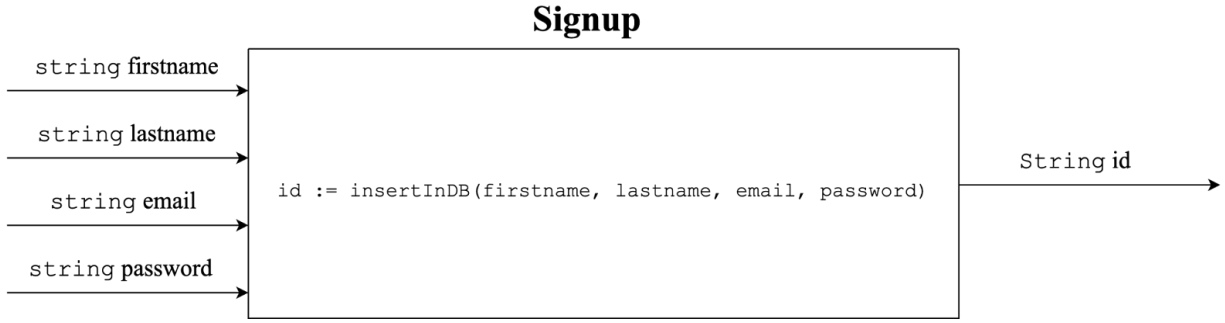


Figure 8. SRC for Signup Component

Both the login and signup components will be having 2 outputs i.e. a string containing the id of the user acquired from the database and another string containing the error message. If there is a success in authentication and the user is authenticated properly, then the components will assign the value of the id of the user to the output variable ***id*** and provide that output to the further component. If there is an error that is occurred during the validation and authentication of the user, then the corresponding output string will be sent as an output in the ***msg*** output variable.

Once the user is signed up or logged in to the application, we will show the user dashboard which displays options like (a.) account settings
(b.) past visits
(c.) add a card
(d.) logout, or
(e.) generate QR code.

2.1.1.3 QR-Generator

If the user wants to park a vehicle and is already signed in to the app, he can generate a QR code using the "Generate QR code" button on the dashboard. On clicking the button, our application will take the id of the user and encode it in a QR code and show it on the screen as shown in the Figure 9.
This process is for the validation of the user as the QR should be a unique entity as per any specific user. The **QR-Generator component** handles all these functionalities for our application. At the click of the button, the event *buttonPress* gets triggered and it sets the counter to 10. This counter will keep on decreasing every round till it reaches zero. This counter will keep the QR code visible on the screen till the scanner will scan the code and the application will receive a response from the server saying "you can enter", "you can't enter", "you can exit" or "you can't exit". Figure 5 depicts the react part of the QR-Generator synchronous reactive component. If the counter is positive, the component will generate a QR code and peek on showing it on the screen. The output variable *showQR* contains the boolean value of whether to show the QR on screen or not to show. Output variable *QR* contains the image that is generated when we need to show the QR on screen.

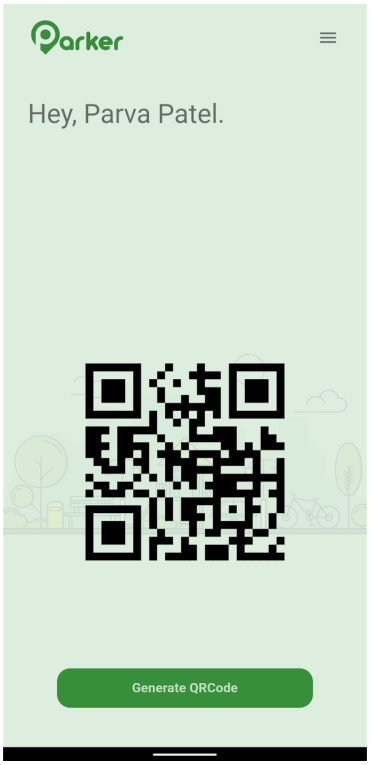


Figure 9. In-app screenshot of QR code generated dashboard

Input, I = {string *id*, event *buttonPress*}.
Output, O = {bool *showQR*, img *QR*}.
State variables, S = {int *counter*}.
Init = {*counter* := 0}.

Inputs of the QR-Generator			
Name	Range of values	Initial value	Unit
id	[a-z A-Z 0-9]*28	N/A	N/A
buttonPress	{⊥, ⊤}	N/A	N/A

Outputs of the QR-Generator			
Name	Range of values	Initial value	Unit

showQR	{0, 1}	N/A	N/A
QR	Image Object	N/A	N/A

State variables of the QR-Generator			
Name	Range of values	Initial value	Unit
counter	[-+]?[0-9]+	0	N/A

React:

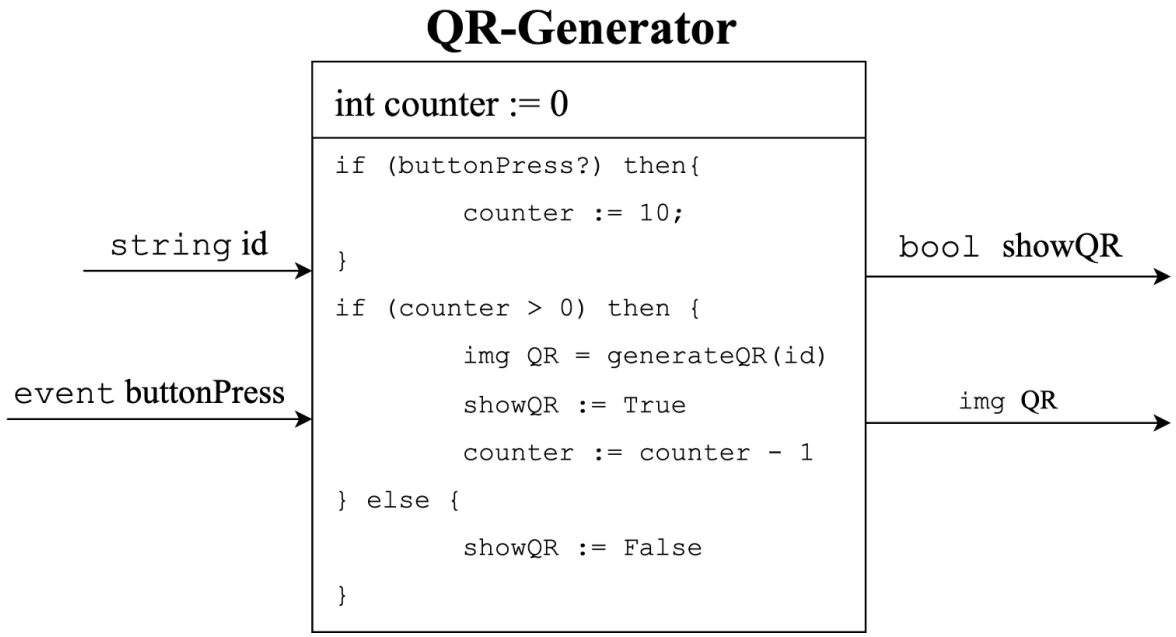


Figure 10. SRC for QR-Generator Component

2.1.1.4 Scanner Component

As mentioned earlier, we have 2 interfaces in the application: client mode and scanner mode. The client mode will be used by all the users or customers of our parking. The scanner mode is a kind of admin mode that will be used by the system. It will open a UI that will be used by the scanner on the gates. So, if in the login page, we enter the credentials of the admin user, then the app will open the scanner page as shown in the figure here. Our **scanner component** handles this part of the project. On the scanner page, we have 2 buttons that allow the scanner to open the camera. The "Entry QRScanner" button is for the scanner that stays on the entry gate of the parking lot. The "Exit QRScanner" button is for the scanner that stays on the exit gate of the parking. We have kept the type of scanner (entry or exit) as a state variable *scan*. During the scanning process, the scanner component will check if it is an entry or exit, and based on the type, the output will change. The scanner will first decode the QR-code image and fetch the unique ID of the user from it. This fetched ID is properly validated from the firebase database, and it will check if it is a valid user and a valid QR code. Now, **if it is an entry scanner,** we will store the entry time of the user in the database and send an API request (POST request) to the server. The UI of the scanner page is shown in the Figure 11.

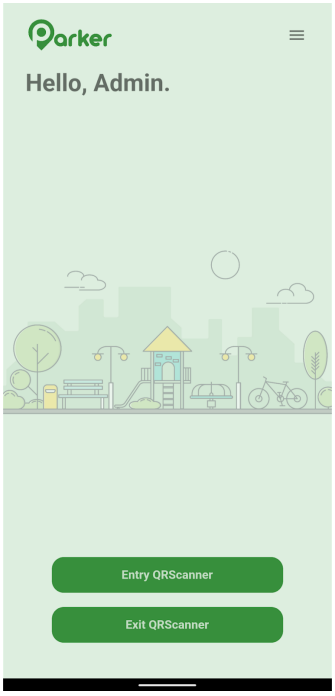


Figure 11. In-app screenshot of Scanner

This API call will contain 2 important things for the server to note.

a. the ID of the user
b. type of the gate i.e. "entry"

The curl command of our API POST request is given below:

```
curl -X POST "192.168.0.203:8080/entry?id=3iH3nC65uJMYF2b88SUsDRqgLi1"
```

In the above request, `/entry` tells the server if it is an entry gate or an exit gate. Given below is the SRC of the scanner component.

Input, I = {QR}.
Output, O = {time, status, id}.
State variables, S = {string scan}.
Init = {scan := "exit"}.

Inputs of the Scanner			
Name	Range of values	Initial value	Unit

QR	image Object	N/A	N/A

Outputs of the Scanner			
Name	Range of values	Initial value	Unit
time	dataTime object	N/A	N/A
status	{“exit”, “entry”}	N/A	N/A
id	[a-z A-Z 0-9]*28	N/A	N/A

State variables of the Scanner			
Name	Range of values	Initial value	Unit
scan	{“exit”, “entry”}	“exit”	N/A

React:

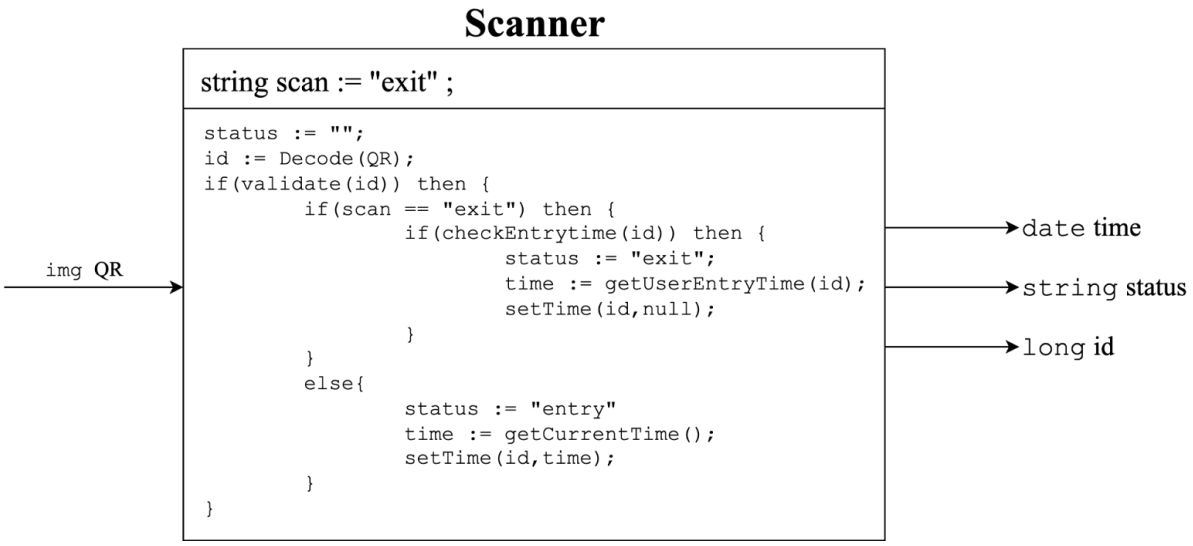


Figure 12. SRC for Scanner Component

If it is an exit scanner, we will fetch the entry time of the user from the database, and similar to the POST request sent during entry, we will send a POST request to the server, but this time, we also sent the time of entry in the API request so that the server can calculate the fare. This time, we also nullify the value of entry time in the database corresponding to the current user.

This was the whole application component of our project. It would completely handle the user and scanner side of the project and provide the necessary data to the server and in return, the server will respond to the application with corresponding outcomes.

2.1.2 Server Component

The server component will be the bridge between the user and the scanner and it will be handling all the calculations and heavy algorithms to find the nearest parking spot in a parking lot of huge size.

Input, I = {event *scannerEntry*, event *scannerExit*, event *check*, date *time*, string *id*, string *status*}.
Output, O = {event *canEnter*, event *canExit*, double *fare*, date *time*, string *spotID*, string *status*}.
State variables, S = {QRRequests *requests*, double *rate*}.
Local variables, L = {bool *flag*, int *counter*}.
Init = {*requests* := [], *rate* := 4, *flag* := True, *counter* := 0}.

Inputs of the Server			
Name	Range of values	Initial value	Unit
scannerEntry	{⊥, ⊤}	N/A	N/A
scannerExit	{⊥, ⊤}	N/A	N/A
check	{⊥, ⊤}	N/A	N/A
time	dateTime object	N/A	N/A
id	[a-z A-Z 0-9]*28	N/A	N/A
status	{“entry”, “exit”}	N/A	N/A

Outputs of the Server			
Name	Range of values	Initial value	Unit
canEnter	{⊥, ⊤}	N/A	N/A
canExit	{⊥, ⊤}	N/A	N/A
fare	{-∞, ∞}	N/A	dollar ($)
time	dateTime object	N/A	N/A
spotID	[A-Z 0-9]+	N/A	N/A
status	{“entry”, “exit”}	N/A	N/A

State variables of the Server			
Name	Range of values	Initial value	Unit
requests	List of QRRequest object	[]	N/A
rate	{-∞, ∞}	4	dollar ($)

Local variables of the Server			
Name	Range of values	Initial value	Unit
flag	{True, False}	True	N/A
counter	[-+]?[0-9]+	0	N/A

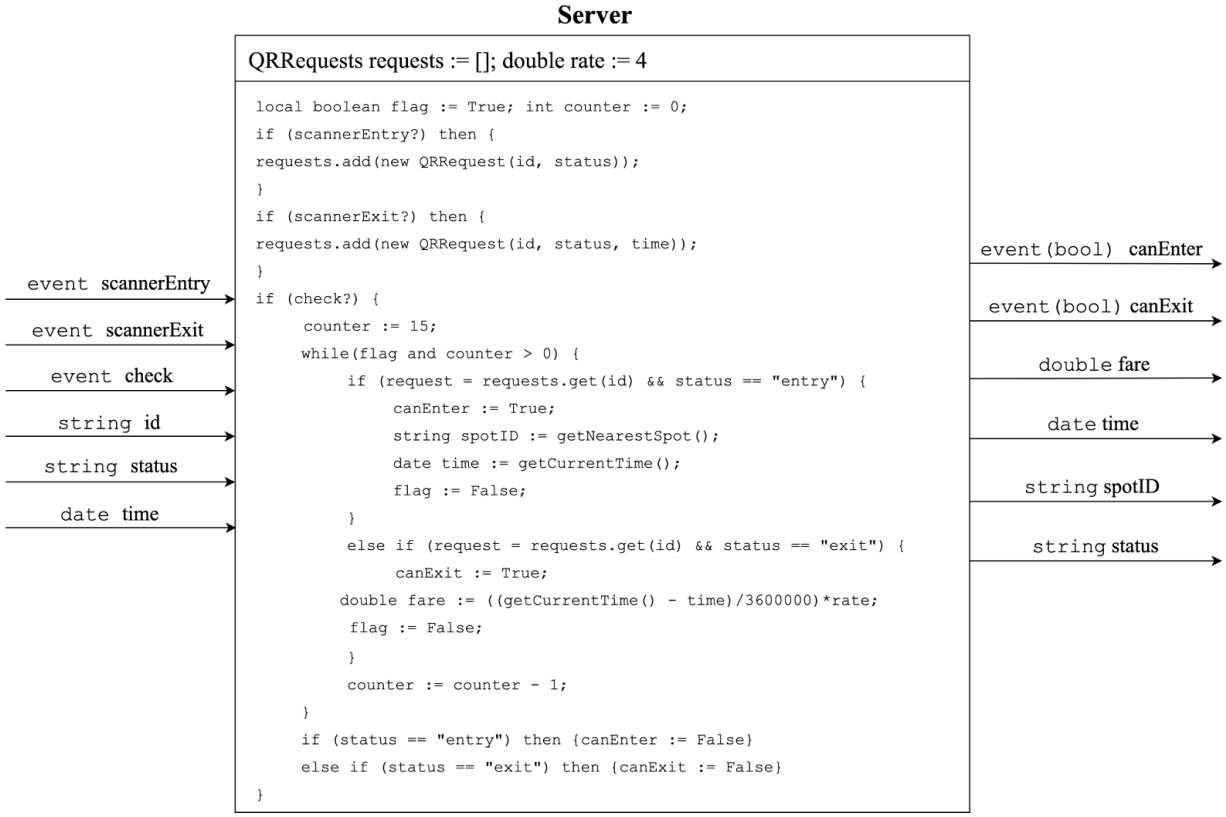


Figure 13. SRC for Server Component

As shown in the figure above, our server component will be taking 3 events as inputs along with other inputs:

- ***scannerEntry*** will trigger when the scanner present at the entry gate sends a POST request to the server. Along with this trigger, the scanner will also send ***id*** containing the ID of the user and ***status*** containing the type of the scanner i.e. “entry”.
- ***scannerExit*** will trigger when the scanner present at the exit gate sends a POST request to the server. Along with this trigger, the scanner will also send ***id*** containing the ID of the user, ***status*** containing the type of the scanner i.e. “exit” and ***time*** containing the entry time of the user.
- ***check*** will trigger when the user’s application sends a GET request to the server. Along with this trigger, the client will also send ***id*** containing the ID of the user so that the server can know which user wants to enter or exit the parking.

In the server component, we have handled a state variable *requests* that maintains the past requests that are sent to the server from the client or the scanner. Based on these past requests, the server can check the validity of any client request and provide them access to the parking or let them exit.

Whenever a trigger is received at *scannerEntry*, the server pushes a new request to the requests array that

would contain the *id* and the *status* in it. Whenever a trigger is received at *scannerExit*, the server pushes a new request to the requests array that would contain the *id*, *status* and *time* (entry time of the user) in it. Now, when the client clicks on the generate QR button, a trigger will be sent at the input variable *check* to the server. This trigger will start a counter for 15 seconds wherein, the server will wait for a response from the scanner that will allow any user with a particular ID to enter or exit. If in the 15 seconds timer, the server didn't receive any response from the server or if there was an invalid QR shown to the scanner, it will update the client with an error that means either there was no response from the scanner or it was an invalid QR code as shown in the screenshot.

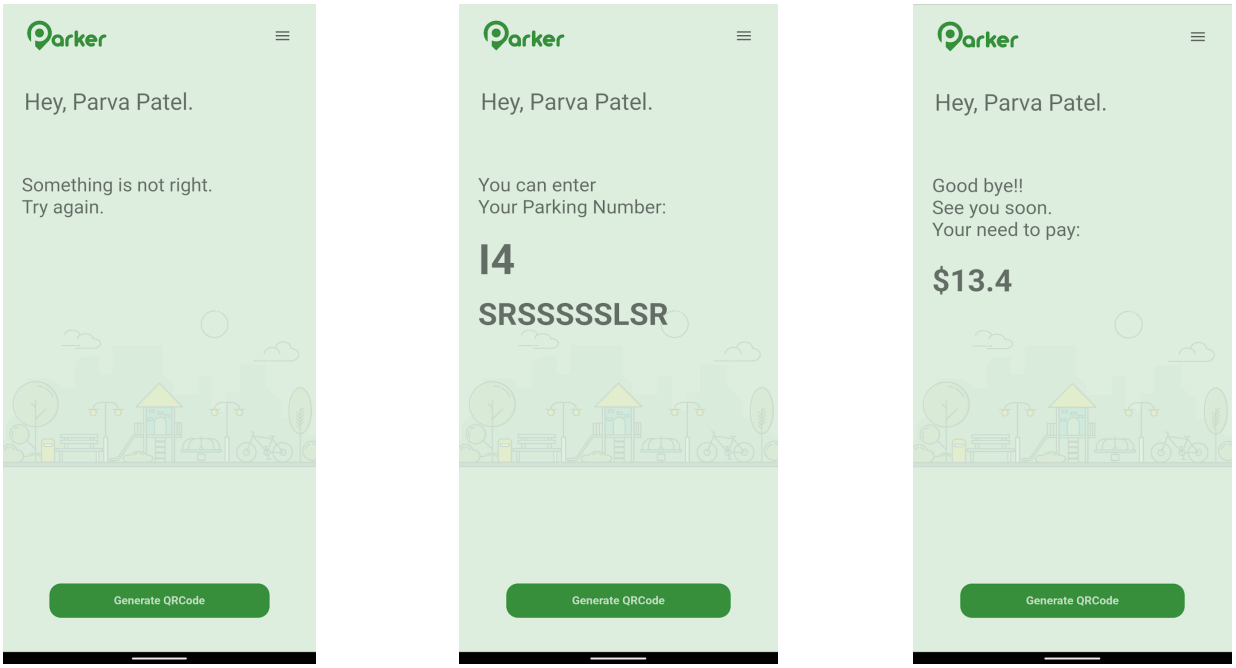


Figure 14. In-app screenshot of (a.) when request is rejected (b.) when the user can enter (c.) when the user can exit

If the QR code was valid and the server picks up a request which matches a request in the *requests* array, then that particular request would be returned with a valid response.

- If the request is for entry, the server fetches for the nearest parking spot from the 2-D array of parkings. The *getNearestSpot()* function runs a breadth first search algorithm in the parking array and searches for the nearest available parking for the current user. Once found, the server will return the event *canEnter* with a True value along with the *spotID* which would contain that id of the parking (for e.g. A5) and the path to that particular parking by displaying them on the screen as shown in Figure 14(b).
- If the request is for exit, the server will calculate the fare that the user needs to pay using the current time and the time of entry of the user. With this obtained value, the server will return the event *canExit* with a True value along with the *fare* that the user needs to pay by displaying them on the screen as shown in Figure 14(c).

2.1.3 Parking Spot Component

This component will handle an individual parking spot's functionality and working. As mentioned earlier, we would introduce proximity sensors in every parking lot and these sensors will help us with the availability of the parking. These sensors will contribute to be very important parts of the physical part of this cyber physical system. Given below is the react part of the parking spot component which will help us understand the functionality of this component.

Input, I = {bool *switch*}.
Output, O = {bool *availability*}.
State variables, S = {}.
Local variables, L = {double *x*}.
Init = {*x* := 100}.

Inputs of the ParkingSpot			
Name	Range of values	Initial value	Unit
switch	{0, 1}	N/A	N/A

Outputs of the ParkingSpot			
Name	Range of values	Initial value	Unit
time	{0, 1}	N/A	N/A

Local variables of the ParkingSpot			
Name	Range of values	Initial value	Unit
x	{-∞, ∞}	100	N/A

React:

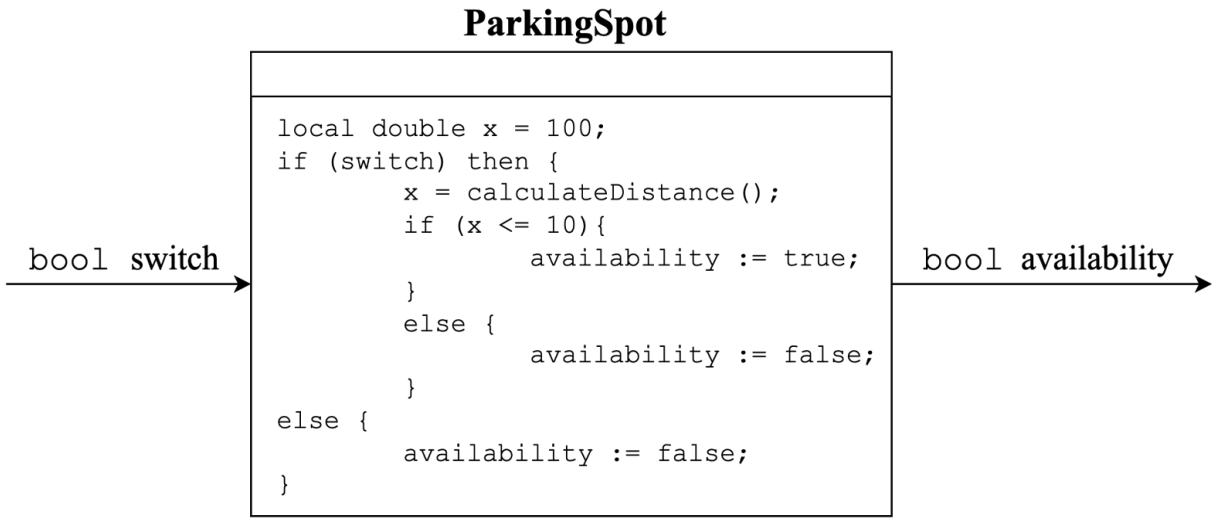


Figure 15. SRC for Parking Spot Component

This component will take a switch as an input variable. Whenever the operator turns on the parking lot's power, it is when this particular switch will turn True. When the switch is true, the sensors will get power to operate and get the proximity values. Whenever the switch is on, the sensor will calculate the distance of any object that is near to it, for our case, the object will be any car.

If there is any car parked in the parking lot, the sensor will get an obstruction in the rays it sends to find anything in its proximity. Based on the distance of the value provided as output by the sensor, if the distance is less than 10 meters, the component will sense that there is a car parked in the parking and the server will set the corresponding parking to be unavailable. Otherwise, if the sensor doesn't sense anything in its proximity or anything is at a distance greater than 10 meters, the sensor will sense nothing and the server will keep the parking available for other cars.

2.2 UML design specifications with descriptions

2.2.1 Class diagrams

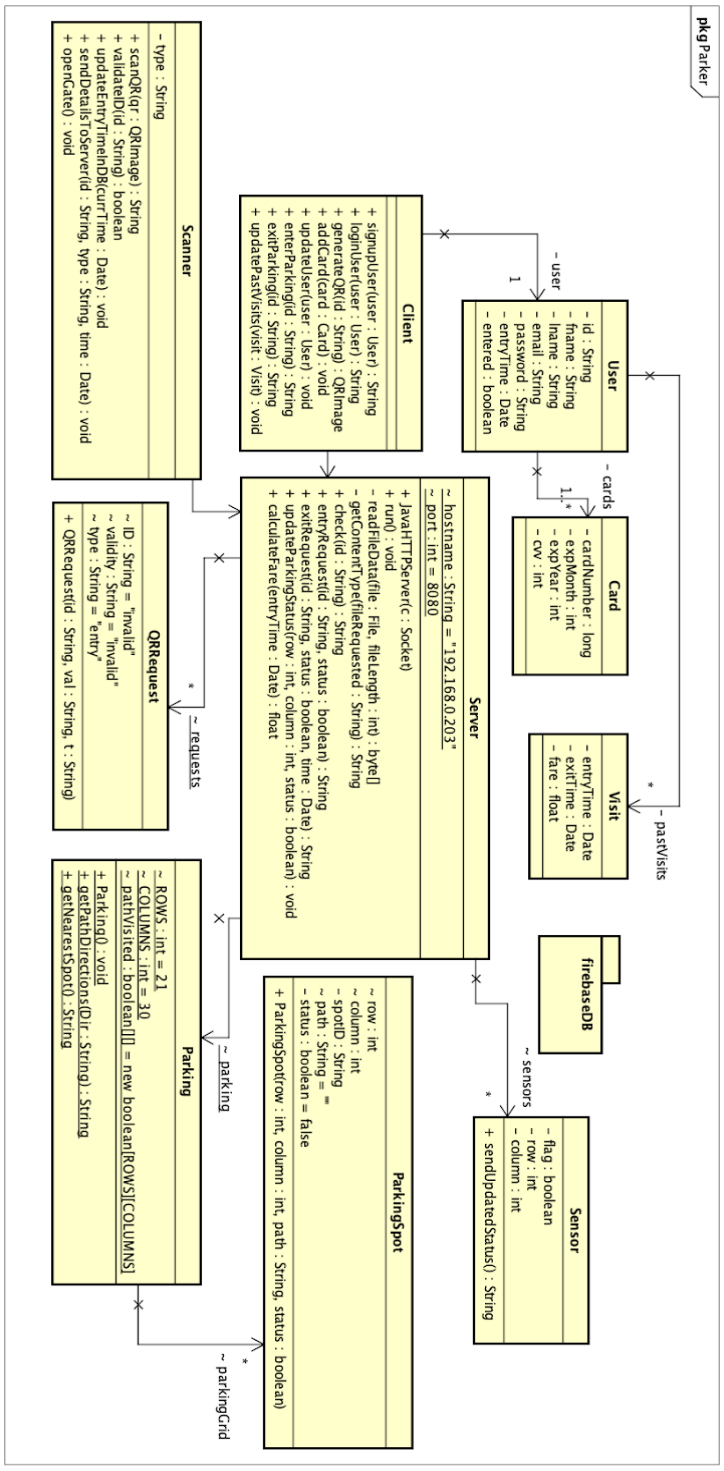


Figure 16. UML Class Diagram

Here are the main classes of our system:

2.2.1.1 Parking

This is the central part of the organization for which this software has been designed. It has attributes like 'ROWS', 'COLUMNS' that provide the total capacity of the parking lot.

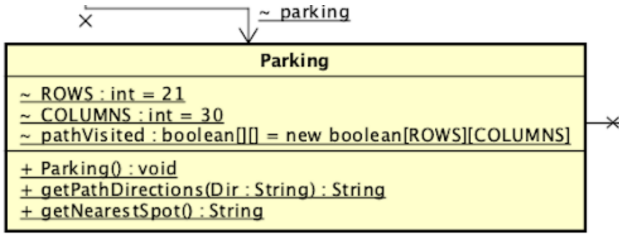


Figure 17. Class diagram for Parking

- 'pathVisited' is a 2-D array that maintains the list of paths that had already been visited to avoid going over those once again to save time.
- 'Parking()' is a constructor that initializes class attributes for each request.
- 'getPathDirection()' provides the path to the assigned spot as a string.
- 'getNearestSpot()' this method runs the BFS over the parking grid and provides the nearest spot available spotId to the client.

2.2.1.2 ParkingSpot

It extends parking class and has attributes like 'row' and 'column' to obtain the exact location of the parking spot in the 2 dimensional array of parking spots.

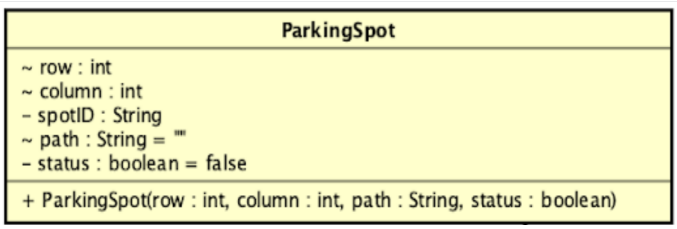


Figure 18. Class diagram for ParkingSpot

- 'status' attributes provide the information about whether the spots are vacant or not.
- 'spotID' to provide an identification number of the spot which also represents its location on the parking grid.
- 'path' provides you with the shortest path to reach the nearest spot.
- 'ParkingSpot()' this method generates the SpotID for the user.

2.2.1.3 User

We have two types of accounts in the system: one for a client, and the other for a scanner.

- 'id' represents the id of the user. It will be unique for each user.
- 'fname' attribute for the first name of the user.
- 'name' attribute for the last name of the user.

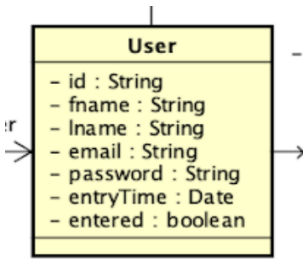


Figure 19. Class diagram for User

- 'email' attribute to store the email of the user which will also be the username.
- 'password' attribute for the user password.
- 'entryTime' attribute to store the timestamp during entry or exit.
- 'entered' a flag attribute which will be set true if the users QR code has been successfully validated at entry scanner.

2.2.1.4 Client

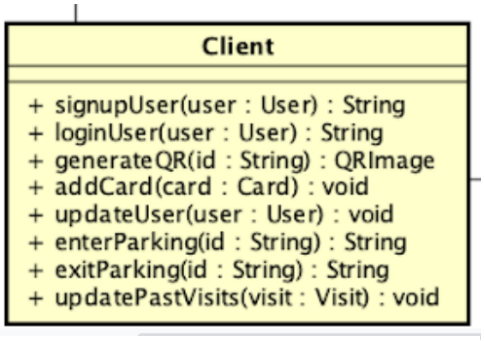


Figure 20. Class diagram for Client

It extends the User class and has method definitions for:

- 'signupUser()' it will be called when new user will register itself to the system.
- 'login()' , registered user will access the system after successfully entering the user credentials.
- 'generatingQR()', after making a signup or login user will be redirected to dashboard where he will press the generate QR button which will call this method and will provide user with the unique QR code at the time of entry or exit.
- 'addCard()' adds card to the database for a respective user.
- 'updateUser()' this method will be called when user wants to update his/her personal information.
- 'updatePastVisits()' this method will add record of user's visit history every time he/she visits the parking
- This all processes will occur in accordance with the server.

2.2.1.5 Server

This class works as a central connection link that not only provides user authentication but also is in a feedback loop with peripheral hardware i.e. scanner and sensor.

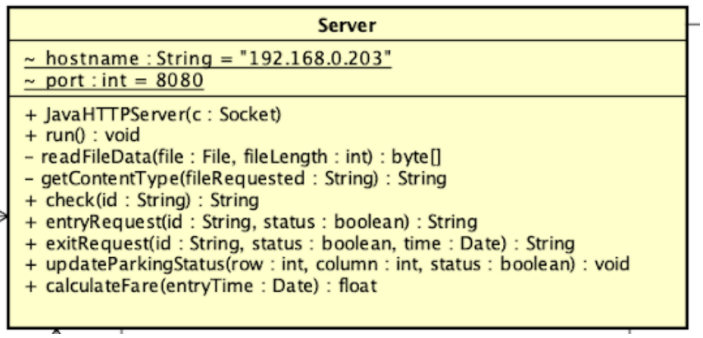


Figure 21. Class diagram for Server

- 'JavaHTTPServer' this method creates the HTTP Socket
- 'run()' makes the connection alive that makes the server alive.
- 'check()' validates the user.
- 'entryRequested()' this method will be called when the user is making entry to the parking lot.
- 'exitRequest()' this method will be called when the user is exiting the parking lot it will take date and time as parameters to calculate fare.

- `'CalculateFare()'` this method will be called when Qr code successfully scanned at exit scanner to calculate the fare.
- `'updataParkingStatus'` this method will take row and column as an argument and update the parking status accordingly.

2.2.1.6 QR request

This class will encapsulate a parking ticket. Customers will take a ticket when they enter the parking lot. it has following attributes:

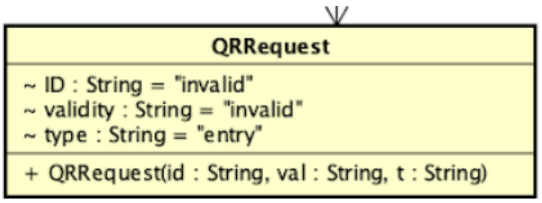


Figure 22. Class diagram for QRRequest

- `'ID'` attribute represents the unique id of the user; its initial value is set to invalid.
- `'validity'` attribute to evaluate whether QR code is valid or not it's initial value is set to invalid
- `'type'` a flag variable whose initial value is set to entry when user initially scans the QR code at entry scanner
- `'QRRequest()'` is a constructor that will handle the API request.

2.2.1.7 Sensor

This class represents the list of sensors that will be used to determine whether the spot is available or not.

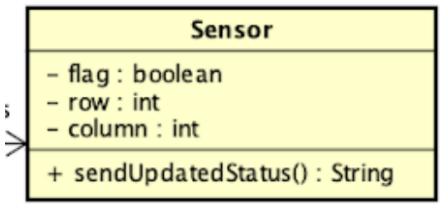


Figure 23. Class diagram for Sensor

- `'flag'` is a boolean attribute that will set true if spot is vacant and false if spot is occupied.
- `'rows'` and `'columns'` these attributes provide location of sensor.
- `'sendUpdatedStatus()'` it will update the status of the flag variable.

2.2.1.8 Scanner

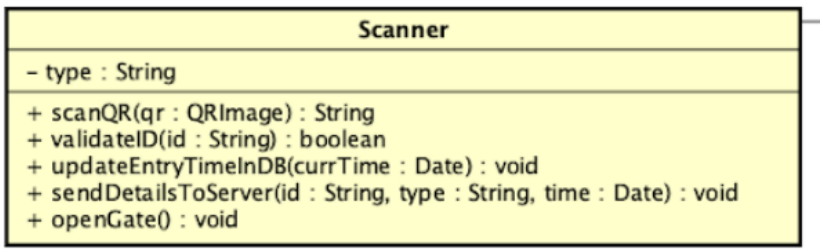


Figure 24. Class diagram for Scanner

Entrance Scanner will print tickets and Exit Scanner will facilitate payment of the ticket fee also it will decode the QR code to validate the user.

- `'scanQR()'` it scans the QR code at the time of entry and exit.
- `'validateID()'` checks whether the QR code id is valid or not.
- `'updateEntrytimeInDB()'` takes the time stamp and updates it's value in firebase.
- `'sendDetailsToServer()'` it sends client id, entry time or exit time depending on scanner and date to server to generate ticket.
- `'openGate()'` triggers acuaters at entry or exit once user is validated

2.2.1.9 Card

This class will be responsible for making payments. The system will support credit card transactions. it takes

- `'cardNumber'` take credit card number as value
- `'expMonth'` takes expire month of card as value
- `'expYear'` takes expire year of card as value
- `'cvv'` takes the user's credit card's CVV as value.

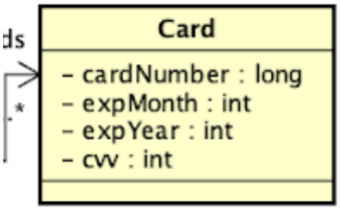


Figure 25. Class diagram for Card

2.2.1.10 Visit

It provides the history of client visits.

- `'entryTime'` this attribute take the time value during entry
- `'exitTime'` this attribute take the time value during exit
- `'fare'` calculates the fare based on entry and exit and stores it in user's history

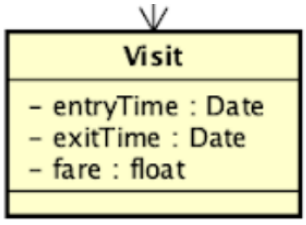


Figure 26. Class diagram for Visit

2.2.2 Use-case diagram

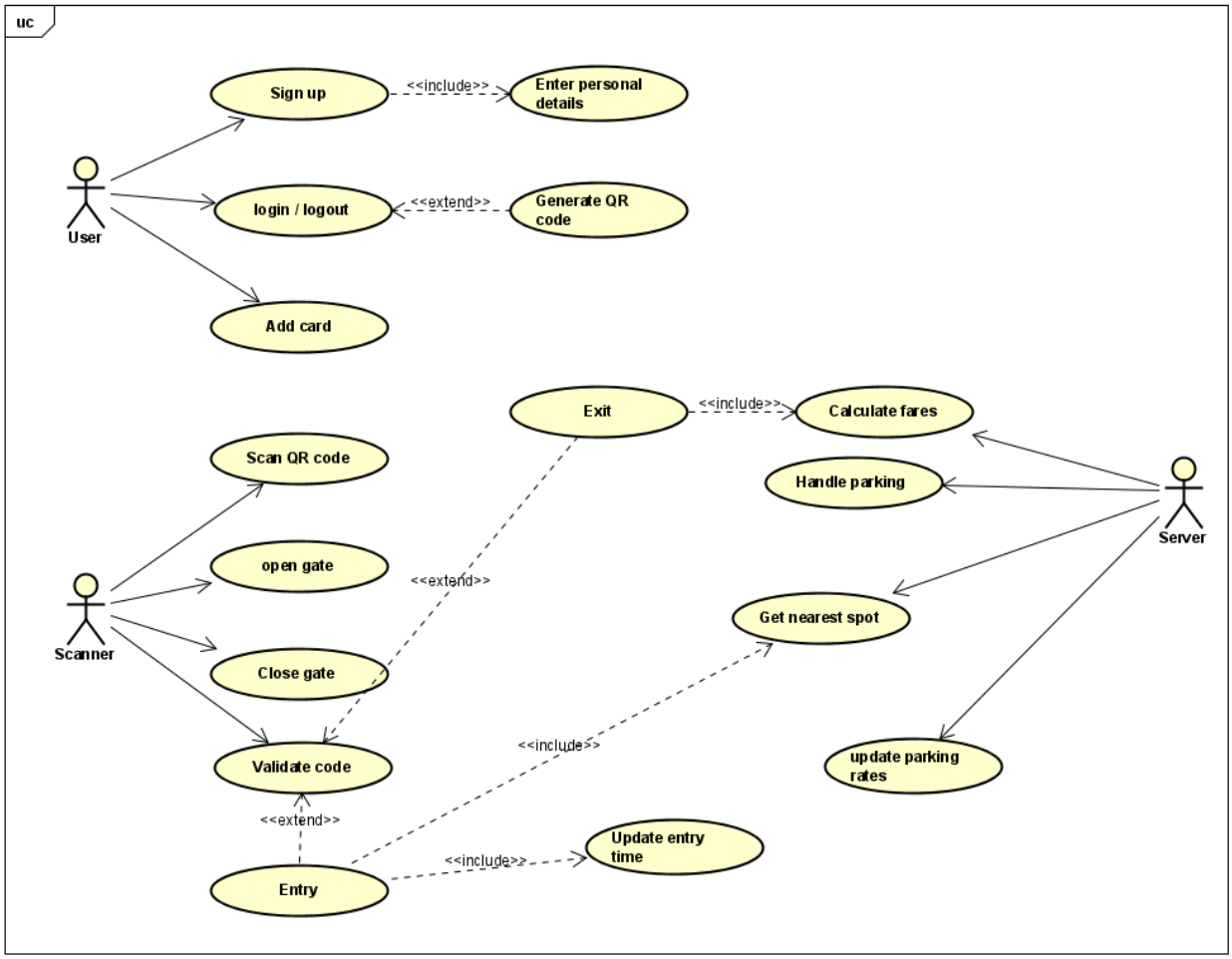


Figure 27. Use case diagram

The use-case diagram of the project shown above will help us understand the entities that exist in our scenarios and will make us aware of the use-cases that exists for the smart parking assistant. There will be 3 main entities in our project:

a) User / Client i.e. a person who is in search for a parking or who needs to park a vehicle)
b) Scanner i.e. a device that will validate the user and inform the server regarding entries and exits
c) Server / System i.e. the backend of the project that will be handling each and every request of client and scanner and be a bridge between them along with calculating the nearest parking spots and fares.

2.2.2.1 The User:

The user is the most crucial and important entity of our project without whom, there is no problem to be solved. The problem faced by a user while searching a parking lot for his vehicle is the most important problem statement of this project and that is what is being solved at every use-case. The user entity would primarily be performing 3 tasks each of which would contribute to a separate use-case.

- **Sign up :** If there is a new user who wants to use our application and start getting an efficient solution for their parking problem, the client needs to get registered in the database. For that, we have provided a signup page to the user. The client will register him/herself with the system by entering personal information like first name, last name, email and password. Once the uniqueness and authenticity of the details of the user are validated, he/she will be navigated to the dashboard.

- **Login** : If the user is an already existing customer of our service, then they must have already registered for an account earlier. In that case, they don't need to sign up again. If the user is already registered then he/she will login using their email and password which will be verified by the server via cross-referencing the firebase database. Once successfully validated the client will be redirected to the dashboard there after which he could generate the QR code by pressing the Generate QR code button.

- **Add Card:** If the client wants to enter the parking lot and park his vehicle, there is a basic prerequisite that we have kept in our application. The client always needs to have at least one card added to their account in order to use our service. If no card is added, the user will be prompted during the QR code generation process and the client won't be able to go ahead. For adding a card, we have provided a button to the client. The client will set up the payment method by entering his/her valid card details like card number, expiry month, expiry year and CVV. Once added, the user can successfully go ahead with generating a QR code for getting a parking spot.

2.2.2.2 The Scanner:

This entity proves to be a very important part for our project as all the validation of the user will be done by the scanner itself and the details of the user and the type of gate will be sent to the server from the scanner itself. The scanner will be performing 3 main task directly in all the use-cases:

- **Scan QR:** After generating the QR code, the user will scan the code at the scanners. If it's an entry scanner then it will validate the QR Code and send the user's ID along with the type of the scanner i.e. "entry" to the server which will generate the spot id for the nearest available parking spot and communicate it to the client and he will be allowed to enter. If it is an exit scanner then on successful validation, the scanner will send the user's ID, time of entry and the type of scanner i.e. "exit" to the server which will generate the fare and pass the details to the client.

- **Validate user ID:** When the scanner decodes the QR code shown by the user, before going ahead, the scanner properly validates the details of the user fetched from the QR code with the existing details in the firebase database. Upon successful verification, the scanner can go ahead with sending the API requests.

- **Open / Close gate:** Every scanner will be assigned a gate with it. If it is an entry scanner, an entry gate will be assigned to it. If it is an exit scanner, an exit gate will be assigned to it. Coupons successful

validation of the user, it is sure that the user can go ahead with either entering or exiting. So, after all the transactions are completed, getting the nearest spot while entering and payment of the parking while exit are completed, the scanner will open the gate and let the user go ahead.

2.2.2.3 The Server:

It serves as the central communication link between different entities of the system. Server will calculate and provide a spot id and generate the path to get to that spot. It will be in a closed feedback loop with proximity sensors to determine whether the spot is vacant or occupied. It will also be responsible for calculating the fare at the time of exit. So, the server will mainly be handling 2 important tasks:

- **Get nearest spot:** As discussed earlier, if the user is validated and the user needs to get the nearest parking spot, the server will run a breadth-first-search algorithm in the graph of parkings and get the nearest available parking spot for the user.

- **Calculate Fare:** If the user wants to exit, the server will take the entry time and calculate the fare that the user needs to pay at the time of exit.

2.2.3 Activity Diagrams

2.2.3.1 For the user entity

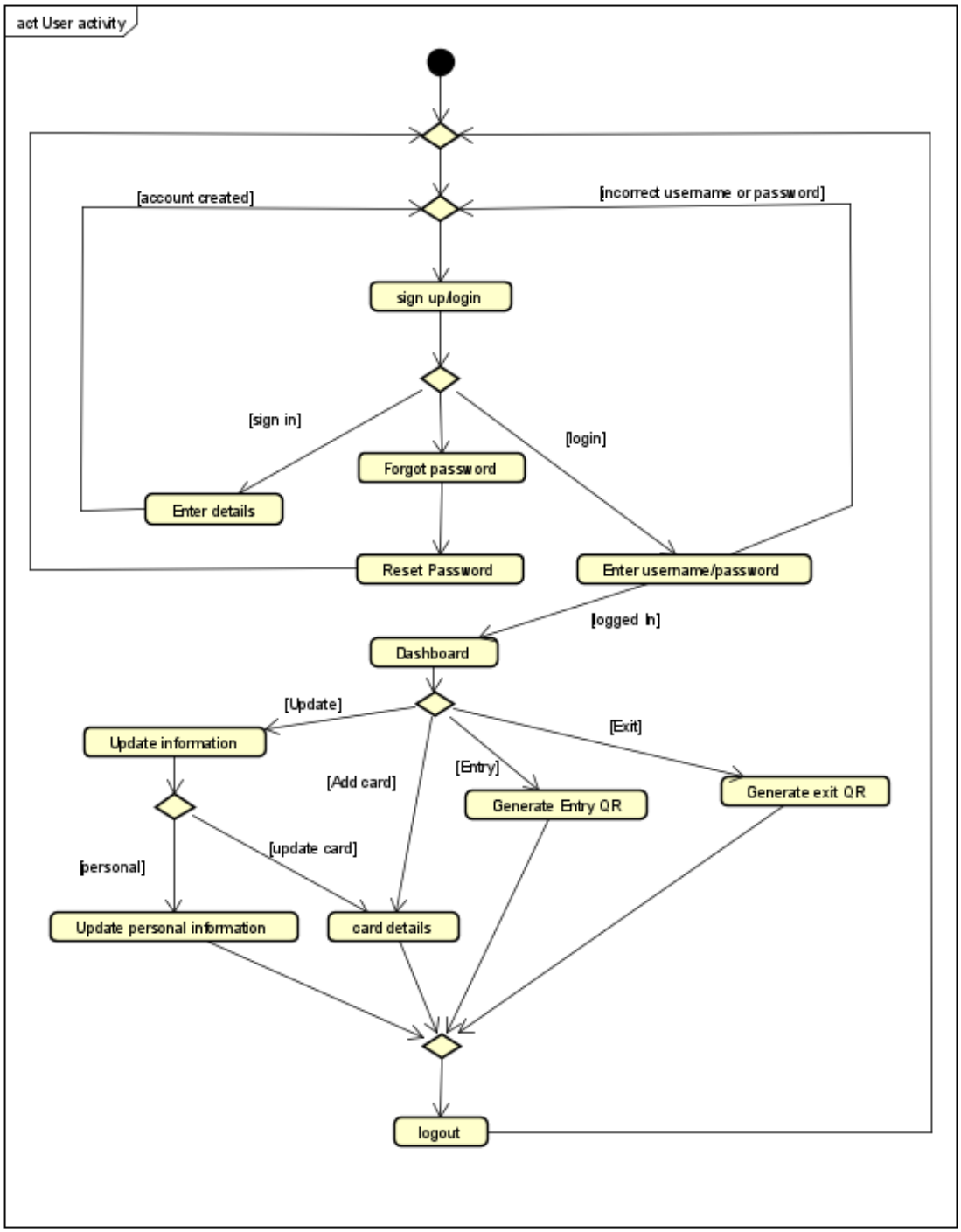


Figure 28. Activity diagram for User entity

2.2.3.2 For the Scanner entity

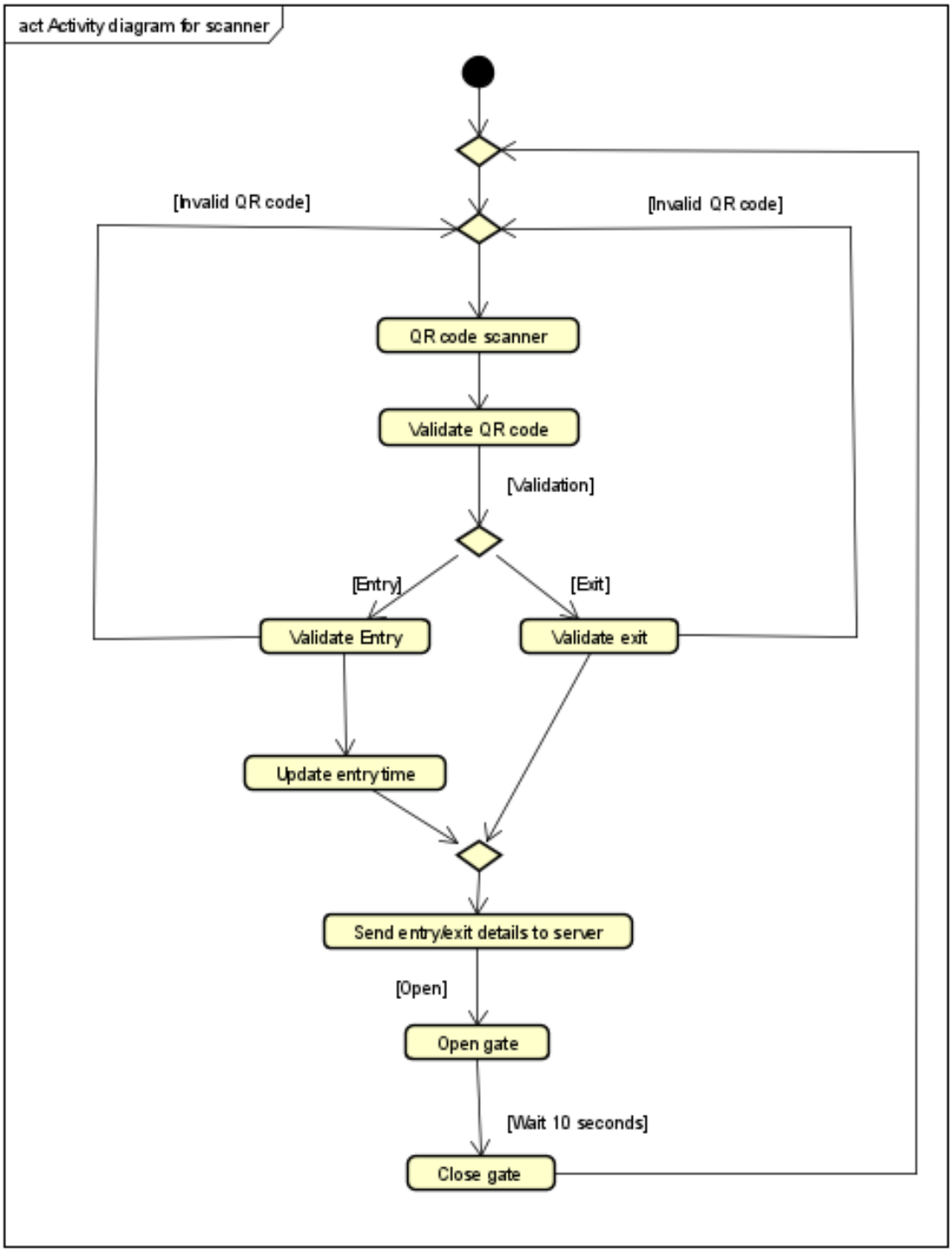


Figure 29. Activity Diagram for Scanner

2.2.3.3 For the Server entity

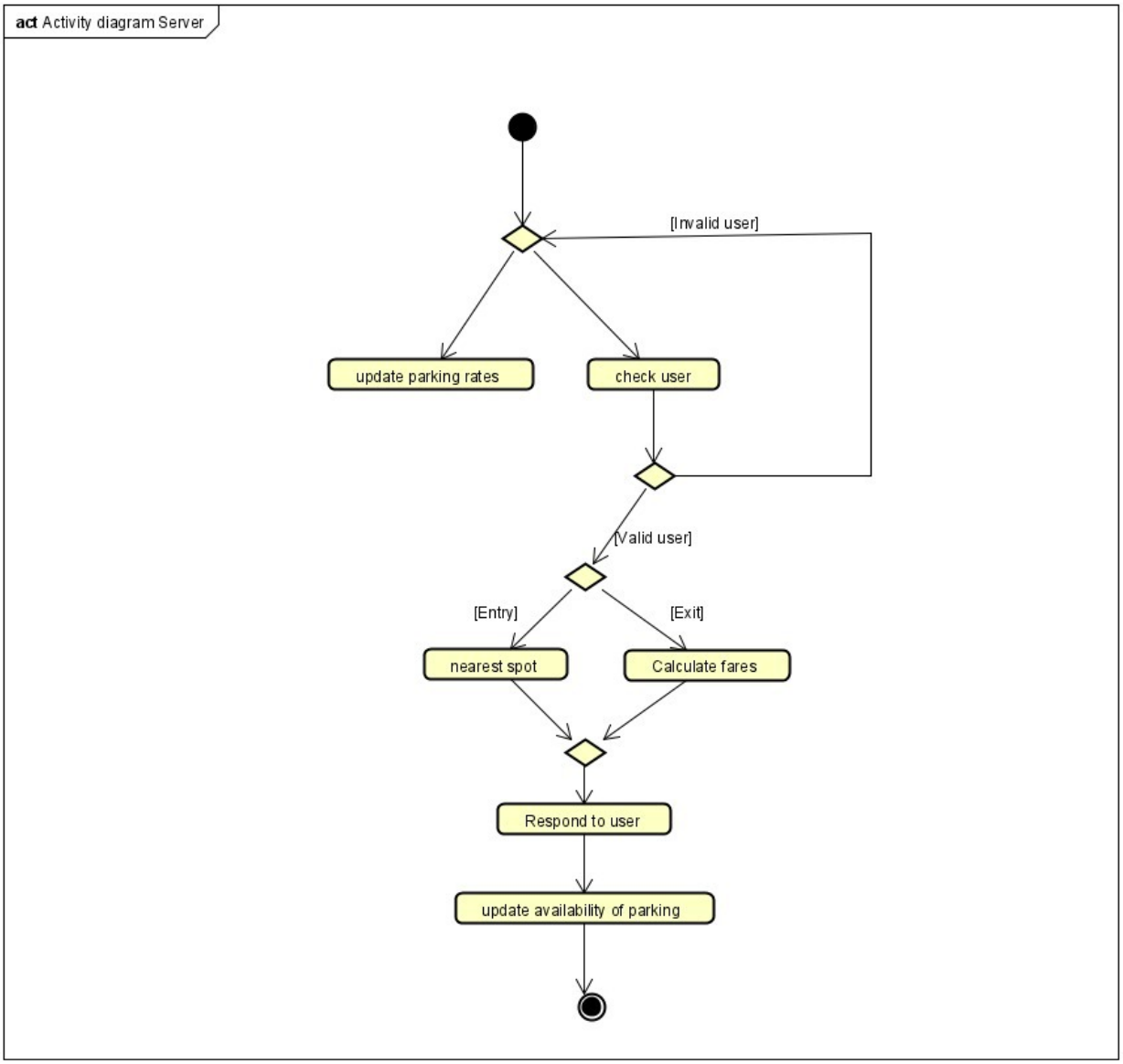


Figure 30. Activity diagram for Server

3 IMPLEMENTATION

3.1 Parker Prototype

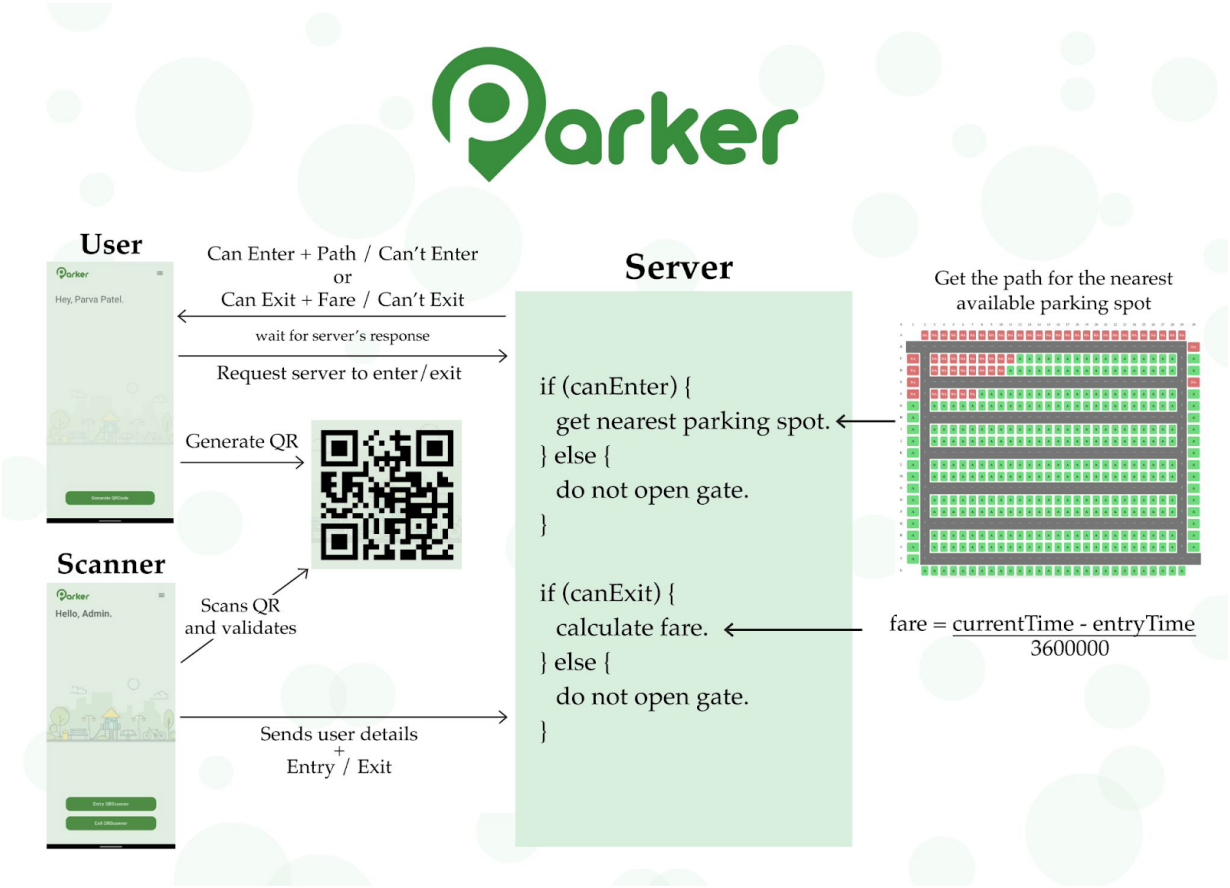


Figure 31. Design architecture of Parker

The goal of the design is to generate a system module that is used to build the system. The figure depicts the proposed mechanism.

The proposed smart parking system monitors and maintains parking lots in order to make the vehicle parking procedure easier and more comfortable for drivers by directing them to their selected parking location with less traffic and a shorter trip time. The system employs cloud computing, fog computing server, a network of sensors, and parking management software to collect and analyse data about parking booking, availability, and pricing. The components may connect wirelessly, allowing them to send data to the fog server, which processes it.

A parking system with a multi-layer IoT architecture efficiently manages and monitors parking lots while maintaining a low congestion rate. The design solves the constraints of standard sensor–cloud architectures by using an efficient fog server to analyse data locally and selectively convey necessary data to the database, rather than sending all the data. This ensures that only the data required for the general view or

long-term services (e.g., parking lots in various locations across a city or historical parking data) is processed and transmitted quickly, resulting in a real-time application experience in which only the data required for the general view or long-term services is transmitted to the cloud. Three layers make up the architecture as shown in the Figure 32:

i. Application layer (the Parker application that will be used by the clients and the scanners)
ii. Network layer (Java Server hosted on remote IP for efficient communication)
iii. IOT/ physical layer (Sensors and Actuators that will coordinate and handle the physical interactions)

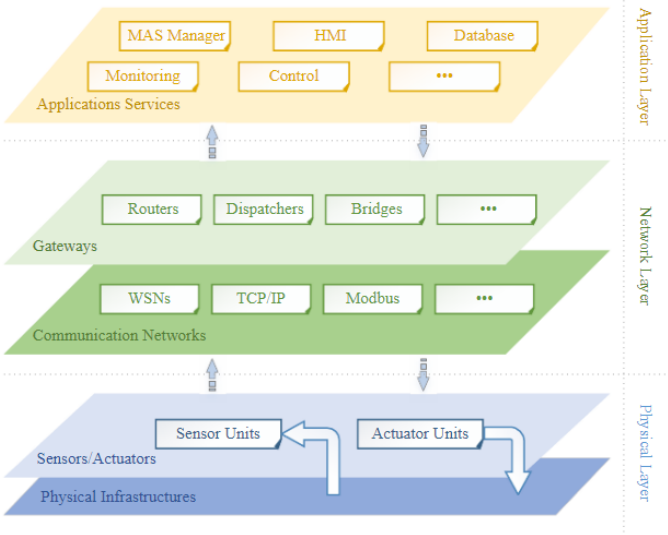


Figure 32. Multi Agent Framework

The bottom physical layer is made up of sensor nodes that monitor parking lots by detecting automobiles and collecting data about them. Different types of sensors (e.g., electromagnetic sensors) might be employed and mounted in the ground, ceiling, or street pillars (e.g., ultrasonic and visual sensors). The best sensor to employ is decided by the parking lot and its surroundings. Ultrasonic sensors and thermal cameras are utilized in a parking lot exposed to varied weather conditions to detect fluctuations in the magnetic field and temperature variations. Visual sensors are preferred for more advanced parking facilities or if they are already in place. For our project, we have used proximity sensors for detecting the availability of any parking. If the sensor detects any object within a range of 10 meters in its proximity, the server will be sent an update suggesting the unavailability of that particular parking.

The purpose of the middle network layer is to provide a robust solution that fulfils real-time control and analytics demands by transferring some processing and analysis to the lowest hierarchical layer (such as data pre-processing, filtering, and compression). This includes the mechanisms for parking-spot selection, parking availability computation, and pricing in our proposed parking system. This server will be handling all the important time consuming and heavy duty tasks that are needed to be handled in the project. We

have deliberately kept all such heavy tasks on the server because the server can be provided with necessary resources to handle such complex functionalities with ease.

The application layer is in charge of task-based decision-making as well as sending actuating orders to specific nodes. It encompasses a diverse set of applications, each with its own set of operating systems and security requirements. Malicious malware and privacy needs are the two biggest security risks. The security of data during transmission, storage, and the display is a must. To maintain the security of the whole system, strict access control policies and authentication mechanisms are practiced in the application so that users can ensure a secured experience.

3.2 Algorithm and working of the project

The application built using flutter and dart contains 2 major entities: the user and the scanner. As shown in the Figure 31, the working of the application starts with the user generating a QR code for the scanner to scan. When the user generates a QR code, a GET request is sent to the server asking for entering or exiting. The client application will wait till it gets a response from the server for this request. Meanwhile, the scanner will scan the QR code of the user, it validates the user and sends the necessary information to the server in a POST request along with the validity of the user. The server checks for the validity of the user and goes ahead with the following rules:

- If the user is invalid and he wants to enter, he can't enter. The server will respond to the GET request of the client with a response of "can't enter".
- If the user is valid and he wants to enter, the server will run the algorithm to fetch the nearest available parking spot and return the parking spot's name along with the path to that spot as a response to the client's GET request.
- If the user is invalid and he wants to exit, the user can't exit. The server will respond to the GET request of the client with a response of "can't exit".
- If the user is valid and wants to exit, but he never entered the parking, the server responds to the GET request to the client saying that "No entry detected for this account".
- If the user is valid and wants to exit, the server will fetch the entry time of the user, and calculate the fare that the user needs to pay and revert back to the client's GET request with the amount to be paid.

As an efficient method of computing a function, an algorithm may be described in a finite amount of space and time, as well as in a well-defined formal language. While allocating a parking place, Algorithm 1 defines the relationship between the user and the Sensors

3.2.1 Algorithm 1: Algorithm of System Operations.

```
Step 1: Start
Step 2: If user not registered
           User register into the system
        Else
           Login into the system
Step 3: User sends the request
Step 4: Server will receive the request
Step 5: If parking space is not available
           Server will send the message that parking is full
           Go to step 3
        Else
           Server will send the nearest parking slot number to the
user
Step 6: User enters the car parking
Step 7: End
```

When a user attempts to locate a parking space, he must first register with the system in order to locate a free parking space and then submit a request via the application. To get the message and to check the park utilizing the table, the system will receive the request and check the table of available parking. When a car arrives at a parking lot, the drivers should be validated by the server. This verification is accomplished by visiting the parking website. If the information is valid, the motorist will be given a receipt and allowed to enter the park. Later, the driver checks to see if the parking lot is empty. If this is the case, he will park and change the availability from unoccupied to occupied. If the current car parking lot is full, the system will send a message "Parking Full".

3.2.2 Algorithm 2: Update spot table

```
Step 1: Start
Step 2: Detects the vehicle using the proximity sensor
Step 3: Update the parking lot array
Step 4: If the vehicle is leaving
           Update the parking lot array
           Go to step 2
        Else
           Go to step 2
Step 5: End
```

After parking the car, the proximity sensors detect the change in the signal. The system updates the state of each lot every 2-3 minutes to update the table case, which is achieved by the setting of the system as shown in algorithm 2; Update urgent data on a new vehicle park containing the new address. The new message will be selected based on the reserved parking lot of the current vehicle.

3.3 Application Component

We have created the Android application in Flutter using the Dart language and have kept the code properly organized and build ready. Given below are some important methods for the application component that we created to overcome functional requirements.

3.3.1 qr_scanner [Lines of code : 256]

This function provides the functionality of the scanner for both entry and exit. The execution of this file opens the camera corresponding to type of scanner in the application and it will scan the user's QR-code which will be then decoded to fetch the user's ID and further will be cross-referenced from the database. If it's the valid match then and only then the user will either be allowed to enter or exit the parking garage based on the value of `Qr.Scanner.userStatus`. If it is set to *`entry`* then the user is at the entry gate else the user is exiting the garage and is at exit gate. Also whether the user is entering or exiting the parking space timestamp will be fetched which will be later used to calculate the fare by the server.

```
if (QrScanner.userStatus == 'entry') {
    await addEntryTime(code!);
    await sendApiResponse('/entry?id=$code');
} else if (QrScanner.userStatus == 'exit') {
    await getEntryTime(code!);
    await sendApiResponse('/exit?id=$code&entrytime=$entryTime');
}
```

3.3.2 api_call [Lines of code : 30]

Class `Apicall` make API request call to the serve which will be running at `IP: 192.168.0.203` and `port : 8080`. This class will make either a GET request call or POST request call to the server using async methods `getApiData(String urlBody)` and `sendApiData(String urlBody)`. Both the methods take `urlBody` as an argument which will hold the user information such as ID, email, password, etc. If the request is a GET request then `networkHelper.getData()` sends the GET request to the server and upon successful acknowledgement of the request, the server will send the requested data to the client which will be fetched and assigned to `apidata` which would be a String. If the request is a POST request then the data will be

```
const javaServerUrl = 'http://192.168.0.203:8080';
class ApiCall {
    Future<dynamic> getApiData(String urlBody) async {
        NetworkHelper networkHelper = NetworkHelper(
                javaServerUrl + urlBody,
        );
        var apiData = await networkHelper.getData();
        return apiData;
    }
    Future<dynamic> sendApiData(String urlBody) async {
        NetworkHelper networkHelper = NetworkHelper(
                javaServerUrl + urlBody,
        );
        var apiData = await networkHelper.sendData();
        return apiData;
    }
}
```

posted to the server and the request will be sent by instruction `networkHelper.sendData()`. On successful acknowledgment the server will send the response code of `200` as a response.

3.3.3 networking [Lines of code : 31]

```
class NetworkHelper {
    NetworkHelper(this.url);
    final String url;
    Future sendData() async {
        http.Response response = await http.post(
                Uri.parse(url),
        );
        if (response.statusCode == 201) {
                String data = response.body;
                return data;
        } else {
                return response.statusCode;
        }
    }
    Future getData() async {
        http.Response response = await http.get(
                Uri.parse(url)
        );
        if (response.statusCode == 200) {
                String data = response.body;
                return data;
        } else {
                return response.statusCode;
        }
    }
}
```

It contains the class *NetworkHelper* which will be called when call is made by `getApiData(String urlBody)` or `sendApiData(String urlBody)` in **api_call** based on the type of request i.e. GET or POST respectively. Asynchronous method `sendData()` will make the POST request call to the server which will send the request containing the `ID` of the user and receive a response with '`201`' status code. The method `getData()` will make a get request for user ID to which server will respond by sending the requested user ID and status code '`200`'.

3.3.4 authorization [Lines of code : 90]

```
try {
	if (_formKey.currentState!.validate()) {
		await _auth
		.createUserWithEmailAndPassword(
			email: emailController.text,
			password: passwordController.text)
		.whenComplete(() => addUser(_users));
		Navigator.popAndPushNamed(context, UserHome.id);
	}
} on FirebaseAuthException catch (e) {
	Fluttertoast.showToast(msg: e.message.toString());
}
try {
	if (_formKey.currentState!.validate()) {
		await _auth.signInWithEmailAndPassword(
			email: emailController.text.trim(),
			password: passwordController.text
		);
		if (emailController.text.trim() == 'admin@nexus.co'
		    && passwordController.text == 'admin123') {
			Navigator.pushNamedAndRemoveUntil(
				context,
				AdminHome.id,
				(route) => false
			);
		} else {
			Navigator.pushNamedAndRemoveUntil(
				context,
				UserHome.id,
				(route) => false
			);
		}
	}
} on FirebaseAuthException catch (e) {
	Fluttertoast.showToast(
		msg: e.message.toString(),
	);
}
```

It will perform the authentication of the user and determine whether the user is a client entering or exiting the parking space or an admin which simulates the scanning activity. The user email id and corresponding password entered by the user will be cross-referenced against data in firebase to check the validity of the user with `signInWithEmailAndPassword()` method that takes email and password as argument. Also, the sign up task will also be performed within this same class using `createUserWithEmailAndPassword()` that will register the user with provided email and password. The user details will be added to the firebase by executing the following instruction `whenComplete(() => addUser(_users);` which will consist of user's first name, last name, email, password.

```
if (emailController.text.trim() == 'admin@nexus.co' && passwordController.text == 'admin123')
```

If the user email and password are `'admin@nexus.co'` and `'admin123'` as shown above then it will open the scanner else for a valid email and password the user will be navigated to dashboard where he will generate the QR code.

3.3.5 user_home [Lines of code : 200]

```
QrImage(
    data: _auth.currentUser!.uid,
    size: w * 0.6,
    errorStateBuilder: (cxt, err) {
           return const Center(
                  child: Text(
                         "Uh oh! Something went wrong...",
                         textAlign: TextAlign.center,
                  ),
           );
    }
)
```

It has a package name `QrImage` that takes user Id to generate an image of the QR code else it will give an error message `"Uh oh! Something went wrong..."` in case of a problem. The generated QR code will be scanned by the scanner which will then be decoded by qr_scanner class as discussed above.

3.4 Server Component [Lines of code : 505]

This is the component of the application that contains the most important algorithms and decision making codes. The code of the server is completely written in java and with the help of socket programming, we have attained the functionality to let the client and scanner communicate with the server.

Socket programming enables us to handle multiple clients at any given time instance as any number of sockets can be opened at a particular IP address. The server of our project uses sockets for the handling of the API calls from the client and scanner. This method of communication has improved the scope of this project to a different extent. Because of a heavy traffic of vehicles on peak hours of a busy day, multiple client requests are highly likely for such a parking system. If there is a blockage in any request of a client, there can be many edge cases that get created and in some cases, the system may result in a deadlock situation or the system getting down. In such cases, the system would definitely require human intervention and needs to be reset. This can lead to a havoc in the parking till some technician arrives and may result in traffic getting accumulated. These kinds of failures are very heavy to take for our system and so, we have designed and developed our server system to be highly efficient and able to handle any number of client and scanner requests parallelly.

Whenever there is a request sent to the IP on which the server is hosted, the server will create a new thread and open a socket connection for the communication amongst the sender and the server. Since a new thread is opened, it will be independent of all the other processes that are running on the server and hence decreases the risk of failures and function overloads. With the help of the opened socket, the server can gather the parameters that are sent in the API request and act accordingly.

The server will handle a static instance of custom created class *parking*. This class will contain a 2-dimensional array for parking spots that will let the server know about the status of all the parkings in the parking lot. This 2-dimensional array will be used to fetch the nearest available parking spot for any new user to enter the parking area. In real life, we would have to handle an array of sensors that will handle the availability of the parking spots on the basis of the reading of the proximity sensors. For this project, we have virtually created a simulation of parking spots in an interactive user-interface. Given below is a simulation of the whole parking that will show us the availability and unavailability of all the parking spots.

We have simulated the working of a real life array of sensors in the parking area with this interactive UI that we properly depict the status of all the parkings. In the above parking lot, the green spots depict available parking spots and the red spots depict unavailable parking spots.

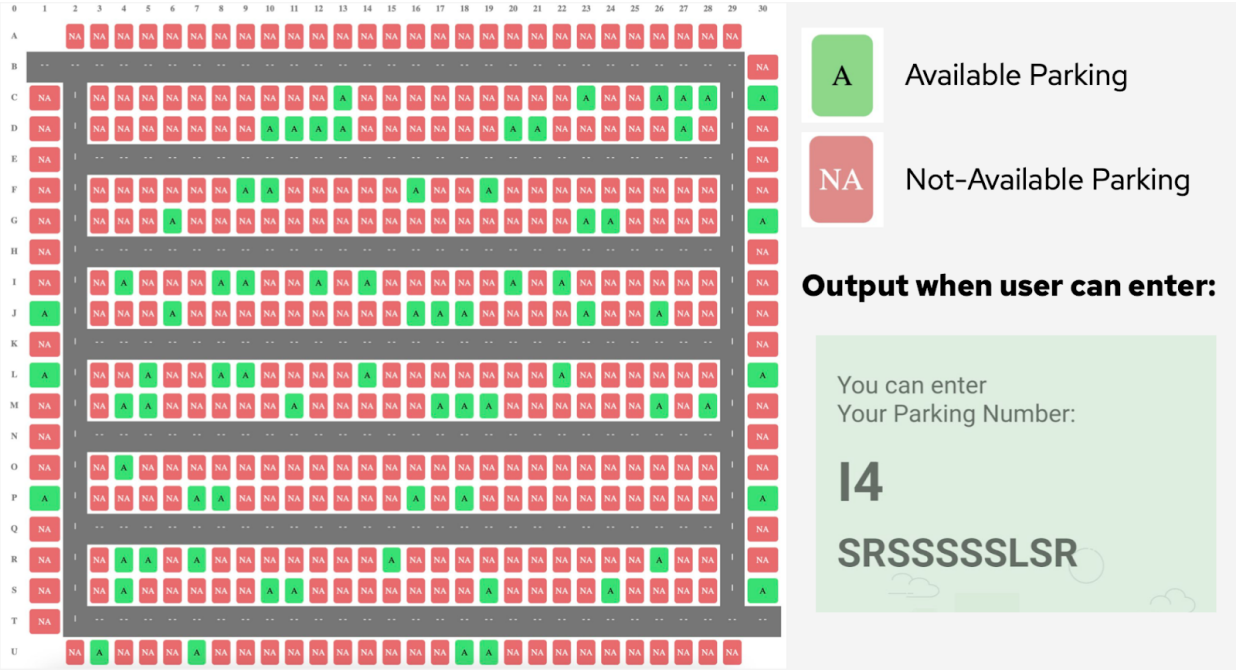


Figure 33. GUI of the sensor simulation of the parking lot

In real life implementation with sensors, the availability of a parking space will depend on the output of the sensor. Similarly, in this simulation, the availability of parking will be determined purely by human interaction. If the admin clicks on an available parking spot, the spot will go unavailable till it is clicked

again. If the admin clicks on an unavailable parking spot, the spot will become available till it is clicked again. We have deliberately kept this system to be admin interactive and not being set by any kind of algorithm. This is because, in real life, there can be a situation where a user doesn't park his vehicle in the parking spot that he is assigned. But instead, fill up some other parking spots. In this case, if we make the assigned parking spot unavailable from the algorithm itself, then there will be a definite mis-communication. The assigned parking spot doesn't have any vehicle parked in it but yet, it is showing to be occupied. To avoid such circumstances, it is always good to rely on sensors, and admin interactions for the availability of any parking spot.

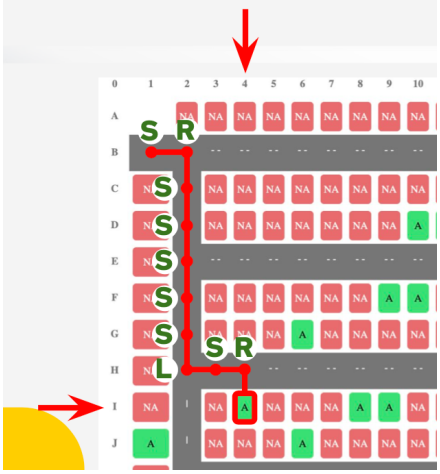


Figure 34. Path to nearest spot

Now, once the user is allowed to enter the parking, the server will call for an algorithm to fetch the path of the nearest parking spot. For fetching the nearest parking spot, we developed an algorithm which traverses every road block from the starting point and checks for the availability of adjacent parking spots. The earliest available parking that is received will be the nearest available parking spot because we traverse the road blocks in a breadth first search manner. So, for the situation of parking spots in the above diagram, we can see that the nearest available parking spot is ***I4***. The path to the destination parking spot will be provided in the form of a string that guides the user on every further block whether to take a left, go straight or take a right. So, the path for reaching ***I4*** will be given as ***SRSSSSSLSR***. This path can be interpreted by the user as shown in the Figure 34 where *S means go straight*, *R means go right* and *L means go left*.

4 EXPERIMENTS AND RESULTS

4.1 User is not signed up.

- If the user tries to sign in without signing up then the app will catch the error and in response, it will show a toast to the user saying there is no user corresponding to that email in the database. This means the email that the user is trying to use to sign in to the application doesn't exist in our database and for logging in to the application, the user's email needs to be existing in the database.

- The toast will say "*There is no user record corresponding to this identifier*".

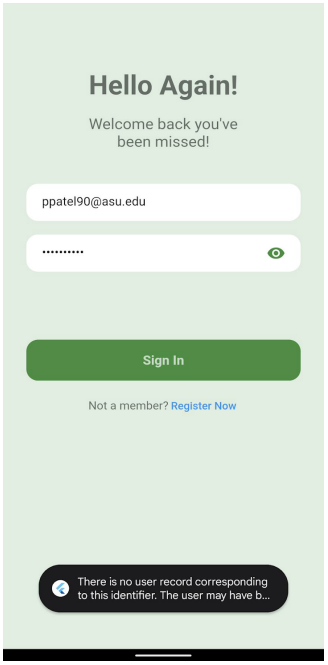


Figure 35. Screenshot of no user found use case

4.2 User enters the wrong password.

- If the user tries to log in with the wrong password then the app will throw an error message as a toast saying that the password is invalid.

- The toast will say "*The password is invalid or the user does not have a password*".

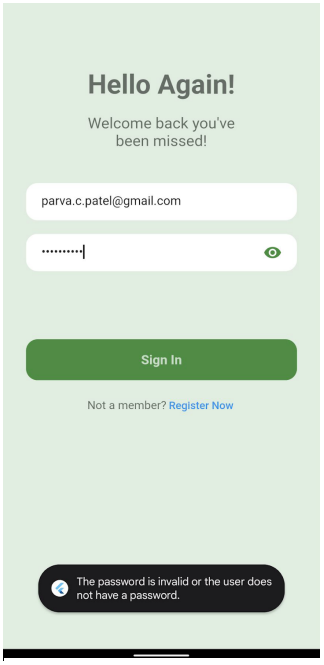


Figure 36. Screenshot of incorrect password

4.3 User successfully signs up or successfully logs in.

- If the user provides a valid email and password, the app will be redirected to the dashboard of the application allowing the user to perform any of the following tasks:
  - Generate a QR code to enter/exit
  - See the past visits and transactions
  - Add a card
  - Log out of the account

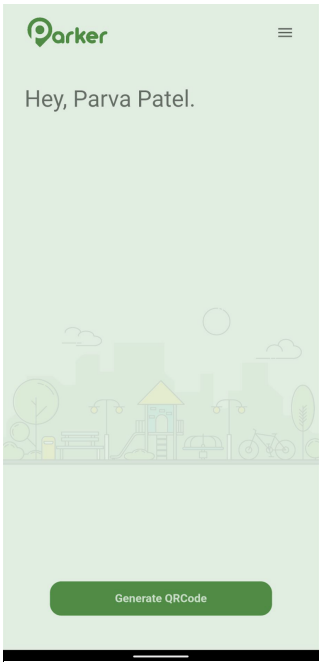


Figure 37. Screenshot of dashboard

4.4 User generates QR code.

- When the user clicks on the "Generate QRCode" button, a QR code will pop up on the screen for the scanner to scan. The scanner will scan this QR code and validate the user. Here, an API call is also sent to the server to check if the user wants to enter into parking or he wants to exit and also to check for the validity of the QR code.

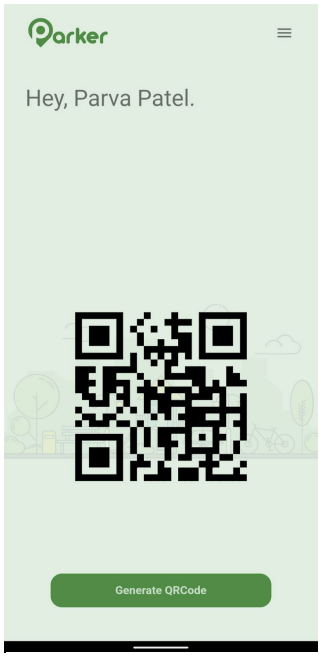


Figure 38. Screenshot of generated QR code

4.5 Admin/scanner logs in.

- For admin access, we have assigned a particular email that corresponds to the admin page. If we log in using those credentials, the application will open the scanner page, where the admin can open the scanner. In the drop-down menu, the admin can go to the account, settings or can sign out.
- There are two types of scanners buttons on the home page.

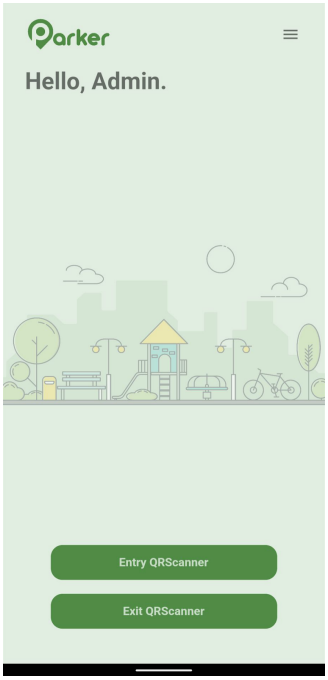


Figure 39. Screenshot of Admin page (Scanner)

4.5.1 Entry scanner

- It will scan the QR code generated by the user, decode the code and validate the user's credentials in the database. It will check whether the user is valid and if the user is already in the parking lot or not. If it goes successfully then an update request will be sent to the database to change the entry time and an API response will be sent to the server along with necessary parameters.

4.5.2 Exit scanner

- It will scan the QR code generated by the user, decode the code and validate the user's credentials in the database. It will check whether the user is valid and if the user is already in the parking lot or not. If it goes successfully then a read request will be sent to the database to get the entry time to calculate the fare and an API response will be sent to the server along with the necessary parameters.

4.6 User can enter.

- Once the validation is complete, and the server receives an API call from the scanner, the server will fetch the nearest parking spot and the path to that spot. Once the processing of the algorithm is done and the server is ready with the response, it responds to the clients by sending the spot name and path to the nearest available parking spot.

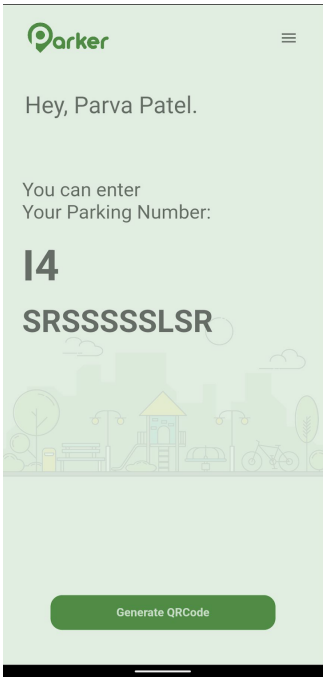


Figure 40. Screenshot of enter output

4.7 User can exit.

- Once the validation is complete, and the server receives an API call from the scanner, the server will calculate the fare from the time difference between entry time and exit time, for the vehicle staying in the parking. (Base fare = $4, Charge per hour = $4).

```
fare = maximum of ($4, $4 * {(currentTime - entryTime) in hours})
```

- After paying the parking fee, users can exit the parking.

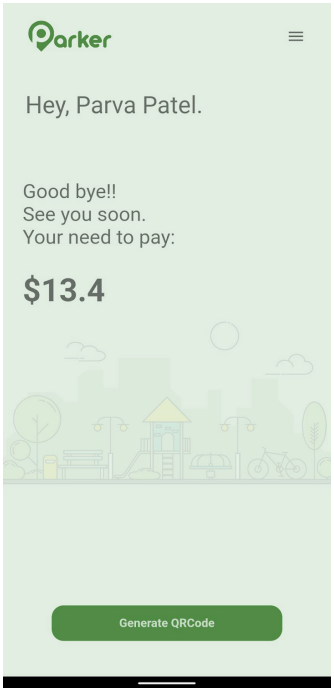


Figure 41. Screenshot of exit output

4.8 User cannot enter/exit.

There can be two reasons for users not being able to enter/exit the parking:

- If the user has an invalid QR code or user tries to scan other types of QR code that may not be interpreted by the scanner.
- The scanner did not scan the QR code. In that case, the QR code will be available for only 10 seconds and the server will send a response asking to try again.

4.9 Parking is full.

- If there is not parking that is available, the server will respond with a message “Parking full”.

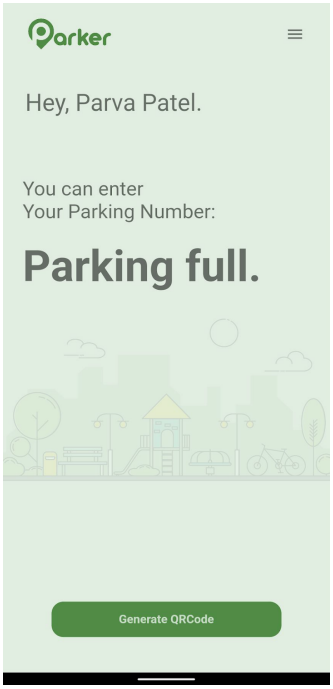


Figure 42. Screenshot of no available parking

5 FRAMEWORKS AND SOFTWARE TOOLS

5.1 Software Tools

Draw.io: to create a formal and semi-formal design that includes SRCs
Astah: To create UML class diagrams, use-case diagrams and UML activity diagrams
Android studio: the front end-user interface was created using flutter in android studio.
IntelliJ IDE: used to create JAVA server
FireBase: for data Source
Postman: for API testing

5.2 Frameworks

As discussed above a distributed middleware based on a hierarchical multi-agent framework is used in SPA to enhance the resilience of CPSs over heterogeneous networks. Here we will take a deeper dive into this framework's network layer that facilitates the interaction between the application and the physical layer.

The network layer is split into the FOG layer and Cloud Layer-The fog layer is often composed of several fog nodes, with each service's process being allocated to a specific node based on its requirements. Fog nodes are widely spread computing units with processing power, each of which may be created by one or more physical Electronics devices (e.g., network device, dedicated server, or computational server). If there is a high execution load or a connectivity issue, each node can process many sensor nodes at once, with the possibility of sensor-node handover. A collaborative task allocation may also be employed to share the computational strain between fog nodes to reduce overall operational latency. The purpose of this layer is to provide a robust solution that achieves real-time control and analytics requirements by relocating some of the processing and analysis (e.g., data pre-processing, filtering, and compression) close to the lowest hierarchical layer. This includes the processes of parking-spot selection, parking availability estimation, and pricing in our proposed parking system.

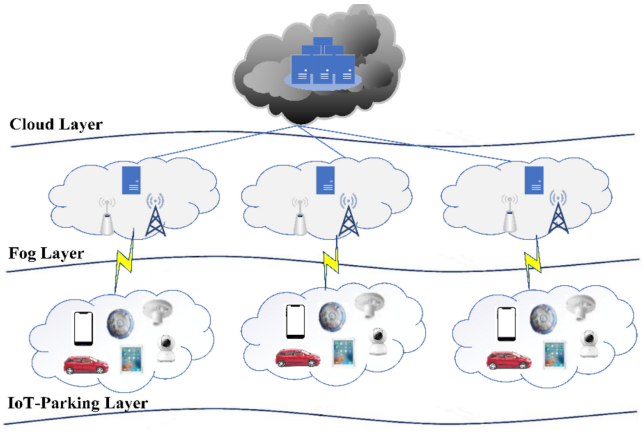


Figure 43. IoT to cloud interaction

The upper cloud network layer provides near-infinite storage and processing capabilities. The cloud layer enables a number of services for the proposed parking system, including usage analytics, statistical and historical data, revenue monitoring, and automatic payment and management. Through a parking platform that runs on the web or is loaded on a mobile device, the cloud layer also provides secure, real-time online parking reservation information and navigation to reserved spaces. To offer security and privacy, the payment plan and driver information might be hidden using an encryption technique on the cloud layer. Historical and long-term data might also be used for big data analytics to enhance traffic management and smart city solutions, as well as real-time feedback notification and emergency alarms.

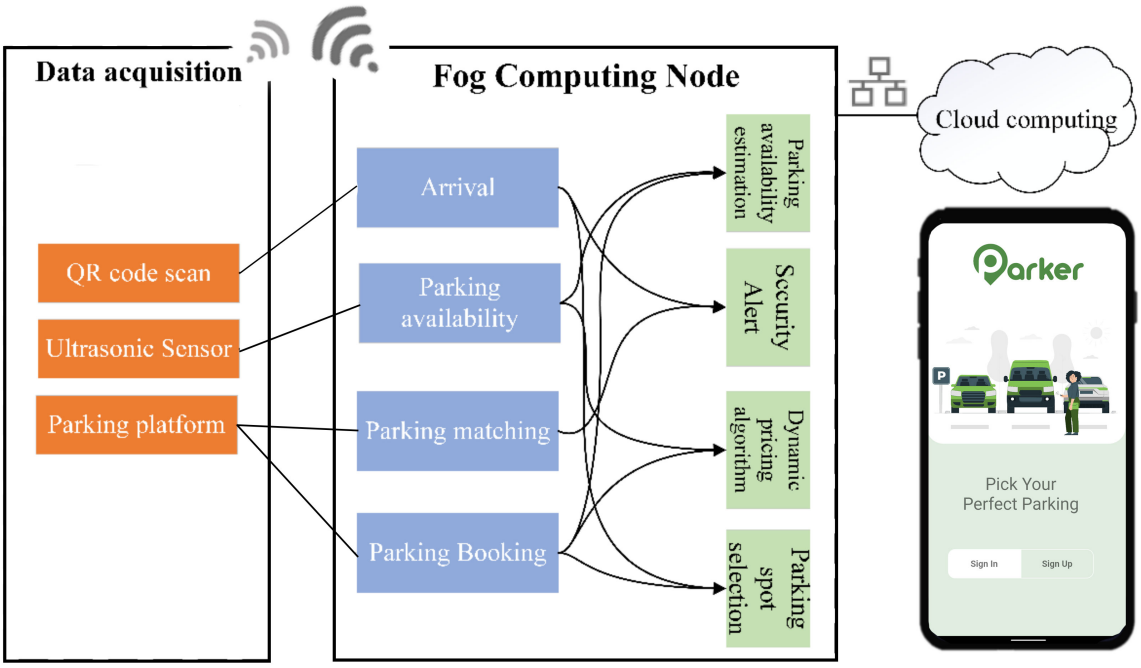


Figure 44.

Figure 44 depicts the proposed parking system's process. The process begins with sensor nodes and the parking management platform gathering parking-lot data. The physical characteristics of the car, such as parking reservation, arrival, and vehicle detection, are relayed to the fog nodes after data capture. Every 10 milliseconds, the fog nodes process the acquired data to ensure precise judgments and to save resources and computing power. The information on arrival, parking availability, parking matching, and parking bookings is derived from the collected data. Parking availability prediction, security warnings, dynamic pricing, and parking-spot selection are all algorithms that are run based on the data. The sub-sections below provide a full discussion of various algorithms.

6 CONCLUSION

Smart cities aim to improve the quality of life of their residents in different aspects of life including transportation and travel services. In this context, smart parking systems help drivers to obtain and reserve a parking spot in advance. In this project, a smart parking system is proposed based on a multi-layer IoT architecture to overcome the limitations of the current parking systems. A novel algorithm to obtain the optimal parking spot with the minimum arrival time was introduced in this project. The results showed that the parking system enhanced the parking arrival time and revenue.

The developed software is classified as a Cyber Physical System. The sensors, actuators, proximity sensor, barcode scanner forms the **reactive** part that will interact with the environment. All this sensor will be running in a closed loop with the server providing the **feedback** from the environment. The system is made **fail safe and safety critical** using MAS framework. The server works as middleware between the cyber part i.e. user interface and physical part. All the sensors will work **concurrently** to one another. All the process execution will be done in **real-time**. Thus, the project suffices all the properties of a cyber-physical system.

A distributed middleware based on a hierarchical multi-agent framework is proposed above in this work to enhance the resilience of CPSs over heterogeneous networks. The CPS is analysed by taking into account its layers, namely the physical, network and application layers, where the main vulnerabilities are the potential physical and cyber-attacks. These vulnerabilities could be addressed based on five groups of agents, namely master agents, MAS managers, robust control agents, cyber security agents, and network reliability agents, which provide the necessary flexibility and counter measures. They allow deploying specific functions, to address cyber security and physical security issues. The developed hierarchical methodology embeds and prioritizes incoming information to ensure state and context awareness, which is used in accommodating resilience-compromising events. One aspect that we believe can be improved upon taking this project into a commercial setting is improving the ease of user interaction with our system. This can be done by providing smart cards to the users that sign up for that program. These smart cards can then be fixed onto the car and can be detected by the sensors at the barriers. The QR code functionality will still remain in the mobile application to not affect user access to the system.

Throughout the development and documentation of this project, we learned the principles of developing model-based design abstractions. Semi-formal and formal designs are significantly used in today's and tomorrow's complex and scalable software-based systems. Good systems are investigated with a focus on design and implementation, using carefully selected frameworks that comply with standards and are supported by software engineering tools.

6.1 Code Quality

A cyclomatic complexity of 1-10 per module is recommended by the software industry. None of the functions exceed this limit, and all classes have a cyclomatic complexity of less than 10. When modularization techniques are used, however, the cyclomatic complexity of any function is never surpassed, making it intelligible and error prone. Programs with lesser cyclomatic complexity are easier to test because the tester does not have to navigate through sophisticated control paths.

The code's classes/object analysis reveals a good balance of public and private methods. This means that the programmer went over and above to make functions that aren't for classes or the user's secret.

The deeper insight reveals inheritance ties with API classes that are either GUI classes or patterns such as observer - observable. This indicates that the design was created first, and the code does not use the JAVA API, but rather inherits and utilizes the relationships provided by an object - oriented programming language.

The average LOC per file is 120 indicating software is brief and easy to comprehend. The overall cyclomatic complexity is 5.2, with an average of 8.7 methods per class, showing that functionality is dispersed evenly among classes. The recommended number of arguments for a function is 7, and no function inside any of the classes exceeds this restriction. There are comments added to every method definitions and code implementations of the java and dart classes – which means the program is properly documented and shall provide an improvised readability.

6.2 Report Quality

The report's overall quality is good. The material was contributed by the developers, who additionally labelled the figures with appropriate subtitles. Making reports simple to navigate and comprehend. The text's creators chose a suitable font (14,12,11) and style (Times New Roman) to distinguish the title subtitle from the content, making the document understandable, as well as specific fonts for code snippets. Each portion is well-explained, and the writers have stayed on topic.

The authors have supplied pictures and diagrams such as the UML Class Diagram, the UML Use Case Diagram, the UML Activity Diagram, and the SRCs that provide formal and semi-formal requirements for the CPS project. They've also released screenshots depicting the experiments done by them in the created application.

They've also provided a Code Quality section, implying that they've gone through their code extensively and followed clean coding methods. The authors of these papers also included an Experiments and Results section, which details the results of their software unit testing. In conclusion they have provided a befitting summary of write up as well as learnings and scope for future works. They've also provided references to which they've referred during the development process towards the conclusion.

6.3 Evaluations

In this project, we focused on delivering high-quality outcomes in every aspect of our work. We also tried to have good relationships with each other as a good bonding amongst the teammates is very important . We believe it is important to be there for each other and provide necessary support when anyone of us get stuck on SRC and UML specifications and work on developing the algorithm that would provide users with directions to reach the spot. During the project, we worked on keeping our knowledge fresh and collaborating with other members to bring new skills on the table and have a wonderful learning experience.

7 REFERENCES

[1] R. Alur, (2015), Principles of Cyber-Physical Systems, MIT Press.

[2] G. Booch, et al., (2007), Object Oriented Analysis and Design (OOAD), 3rd Ed., Addison Wesley.

[3] Java Platform, Standard Edition (Java SE), (2019), https://www.oracle.com/java/technologies/java-se.html.

[4] OMG 2012. "Unified Modeling Language version 2.5.1". https://www.omg.org/spec/UML/2.5.1/.

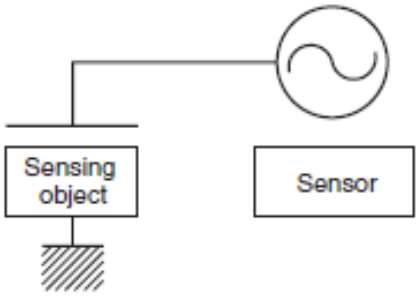
[5] H.S. Sarjoughian, (2020), Software Design Course Project, CSE 564: Software Design (2019 Spring) 2019Spring-T-CSE564-32043 (Blackboard).

[6] A Distributed Multi-Agent Framework for Resilience Enhancement in Cyber-Physical Systems March 2019 - F. JANUÁRIO1,3, A. CARDOSO2, and P. GIL

[7] Smart Parking System using Android and QR Code March 2020 - Yuqbal Faza Aula Dipa, Dani Hamdani

[8] https://www.ia.omron.com/support/guide/41/introduction.html

A APPENDICES



"Proximity Sensor" includes all sensors that perform non-contact detection in comparison to sensors, such as limit switches, that detect objects by physically contacting them. Proximity Sensors convert information on the movement or presence of an object into an electrical signal. There are three types of detection systems that do this conversion: systems that use the eddy currents that are generated in metallic sensing objects by electromagnetic induction, systems that detect changes in electrical capacity when approaching the sensing object, and systems that use magnets and reed switches. The Japanese Industrial Standards (JIS) define Proximity Sensors in JIS C 8201-5-2 (Low-voltage switchgear and control gear, Part 5: Control circuit devices and switching elements, Section 2: Proximity switches), which conforms to the IEC 60947-5-2 definition of non-contact position detection switches.
JIS gives the generic name "proximity switch" to all sensors that provide non-contact detection of target objects that are close by or within the general vicinity of the sensor, and classifies them as inductive, capacitive, ultrasonic, photoelectric, magnetic, etc.