SMART PARK

"RU Ready to Park Smart?"

Report 2

Github: https://github.com/swetha-5689/parking-manager

Group 3

Disha Bailoor, Neha Nelson, Param Patel, Swetha Angara, Nicholas Meegan, Thomas Murphy, Charles Owen, Jeffrey Samson, Aniqa Rahim, Brian Ogbebor

Table of Contents

Individual Contributions Breakdown

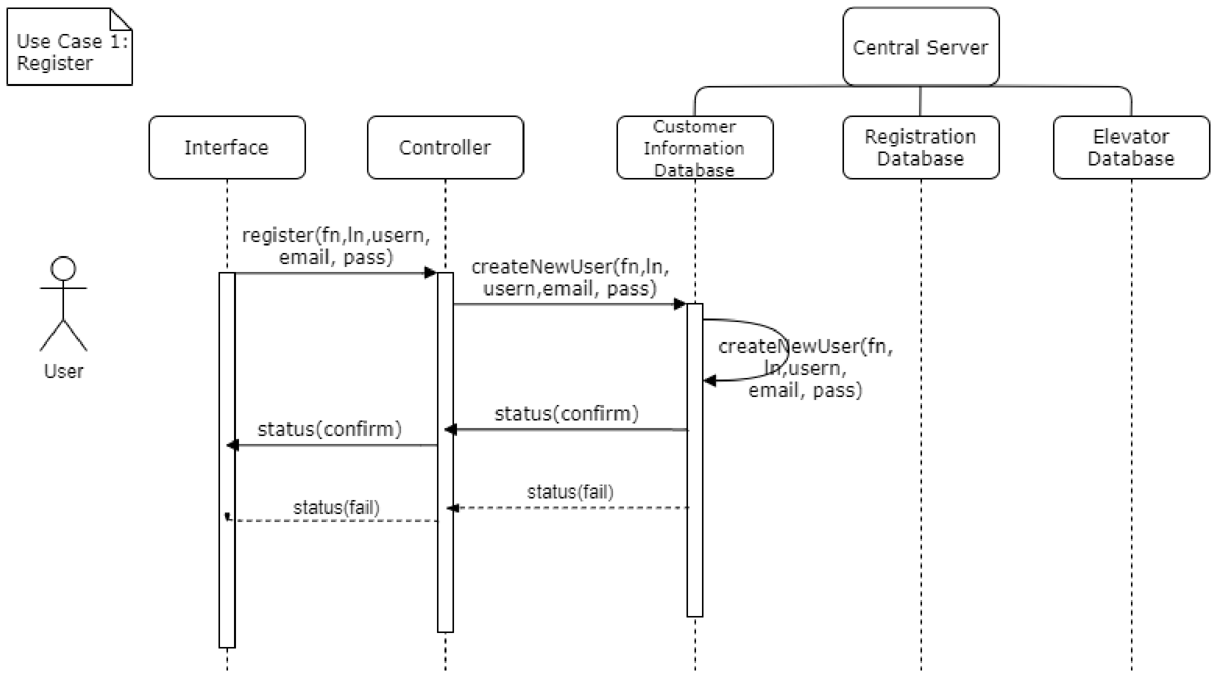
After consulting the entire group, we have decided that all of the team members have equally contributed to this report.

Interaction Diagrams

Customer Registration Subgroup

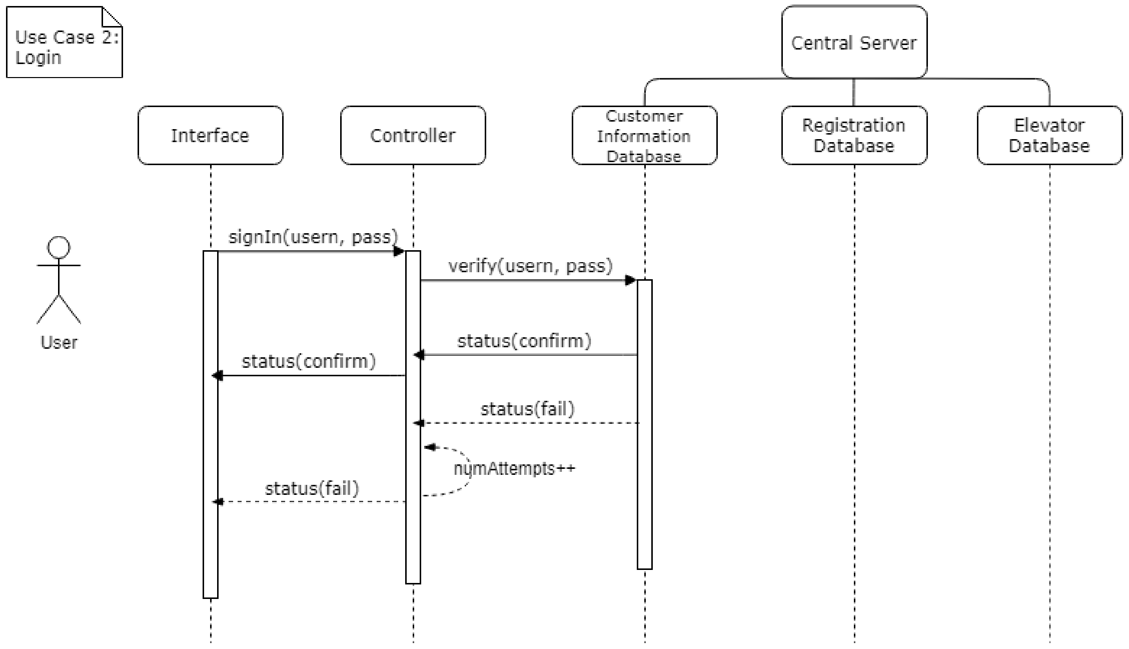
Complete Use Cases are in solid black arrows; Dashed arrows show alternate scenarios

a. Use Case 1: Register



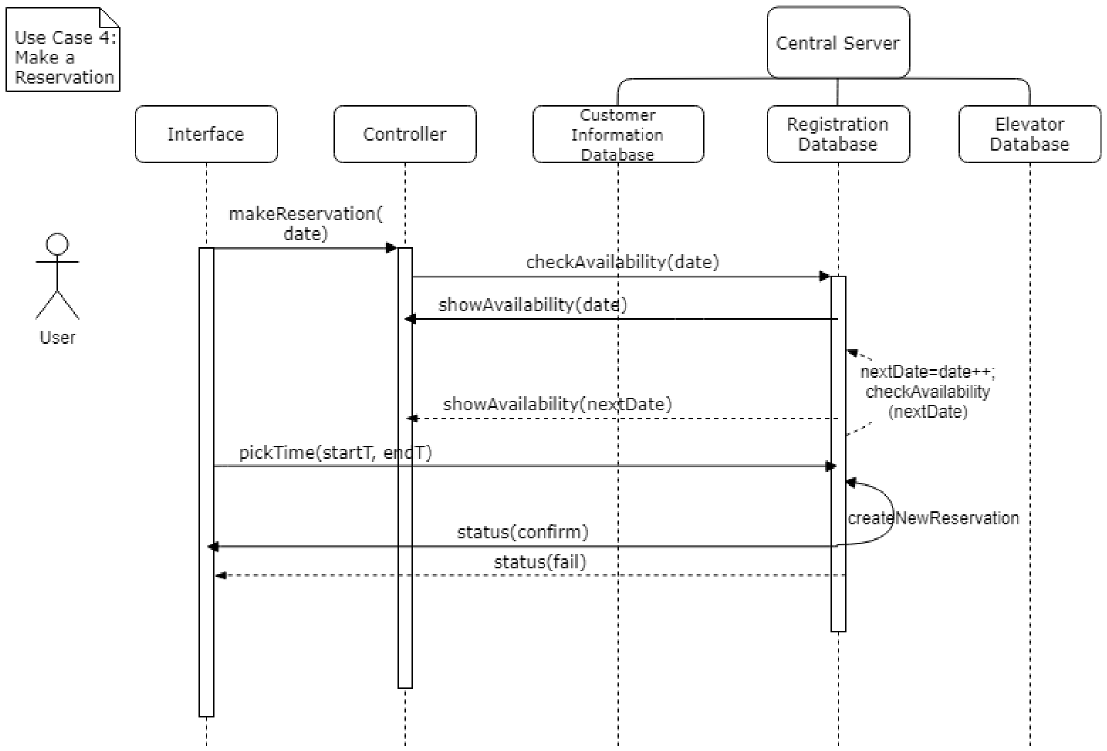
The main flow of this use case is the actions with the solid black arrows. If the user wants to create a new account, they will be brought to a page which will require them to input their first and last name, followed by a username, email, and password. This information is then sent to the Customer Information Database, under the Central Server that checks if there is no existing customer with any commonalities in email and username. Then, it creates a new entry for the user with their corresponding information that will be accessed and checked when he/she logs in. After the entry is created, the user is given a confirmation to go ahead and login with the information they provided. The alternate scenario is that there is already a customer existing with the credentials that were entered. In this case, the system prompts the user to re-enter the information and/or check if they already have an account with the SmartPark system.

b. Use Case 2: Login



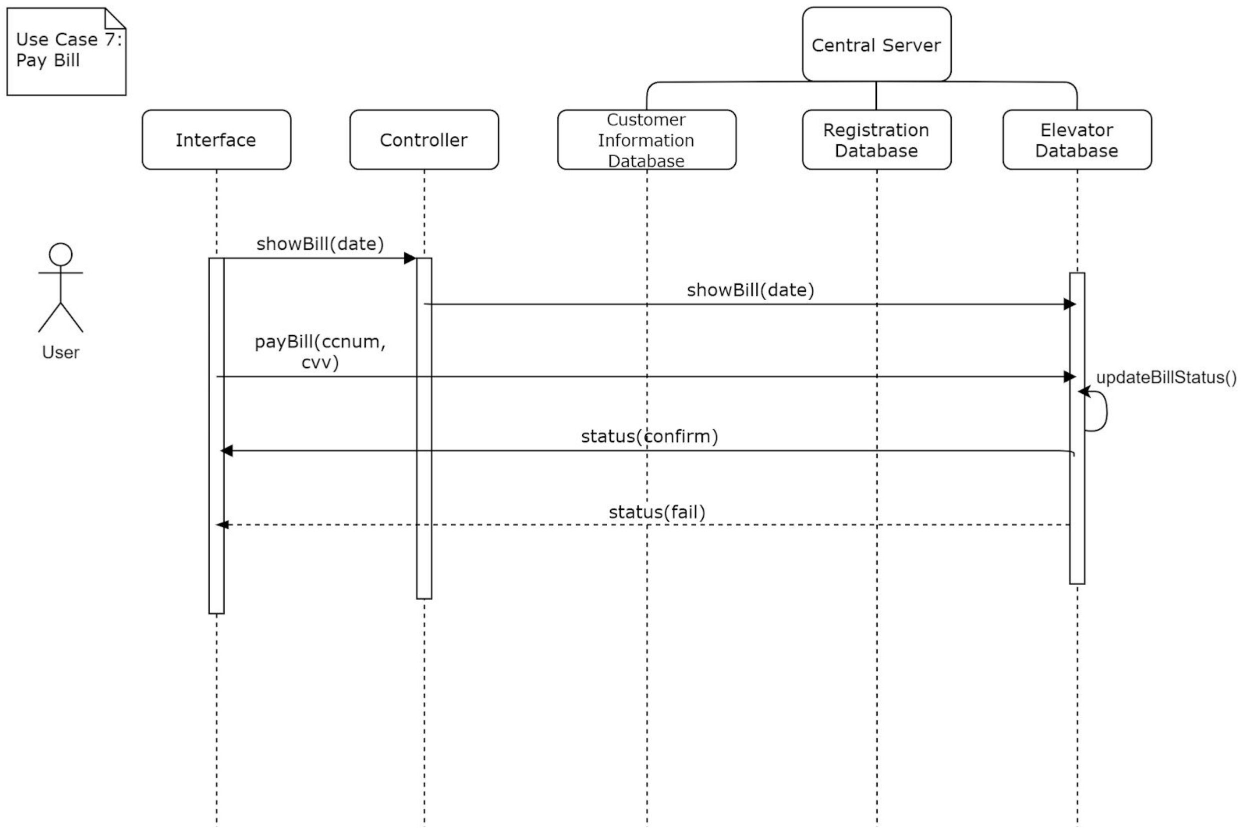
This is the use case for when the user wants to log in to the SmartPark system, it is assumed that the user already has an account. Once they enter their information then the system verifies the matching set of username to the password and either finds it to be true or false. If the username and password match an entry already on file then the system logs in the user. If the information is found to be false, then the system increments the number of attempts to be 1 and then prompts the user to re-enter their username and password. This goes on until there are 5 incorrect attempts and the account becomes suspended for 24 hours, the user is informed of this in an email and can choose to reset their password for security purposes.

c. Use Case 4: Make Reservation



This is the use case that makes a reservation for the user, it assumes that the user has already logged into the SmartPark system. After the user decides to make a reservation they are asked what day they want to park and then the system proceeds to check the availability for that date. If there is availability then the system asks the user to pick the start and end time between which they want to park. If there is no availability then the system finds and displays the next date of available parking.

d. Use Case 7: Pay Bill



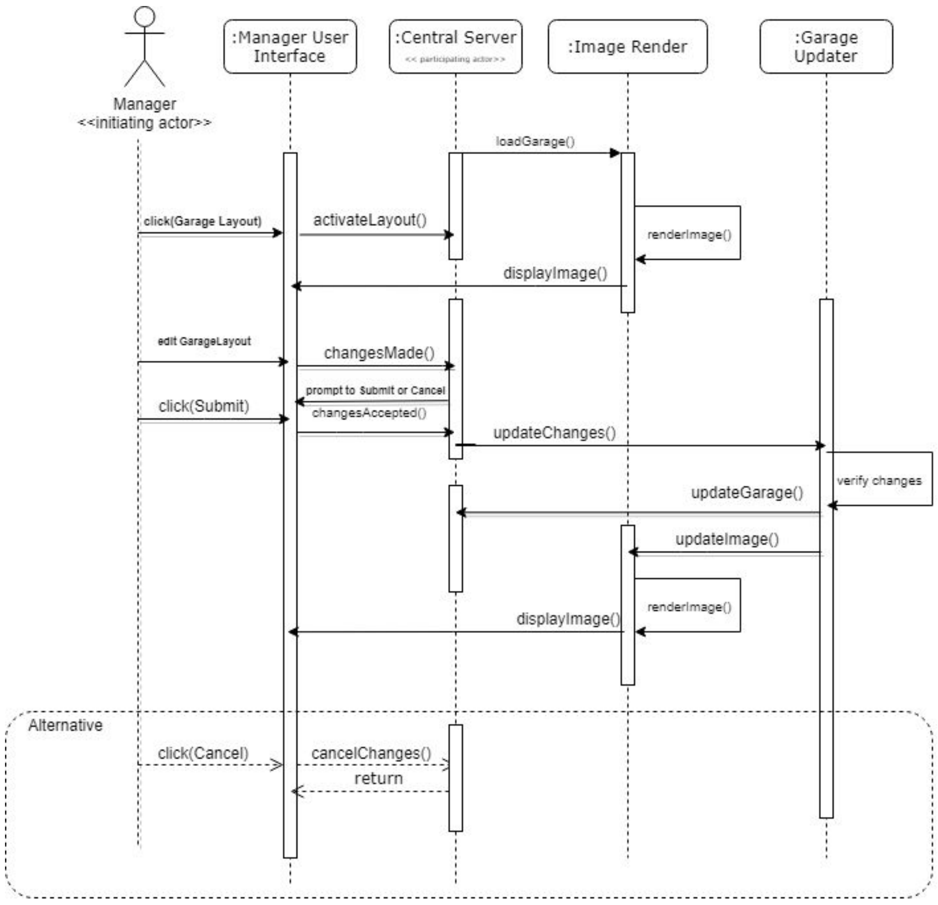
This use case deals with the situation in which the customer wishes to pay their bill online. This use case assumes that the user has already logged in and completed making a reservation. The controller requests the elevator database to display the bill from the beginning of the month to the date entered by the user. Then, the customer puts in their credit card number and CVV pin. The system checks this information and pays the bill by updating the status of the bill statement. If this information is false then the payment fails and the user is prompted to enter another credit card number and CVV.

Managerial / Administration Subgroup

Design Principles Overview

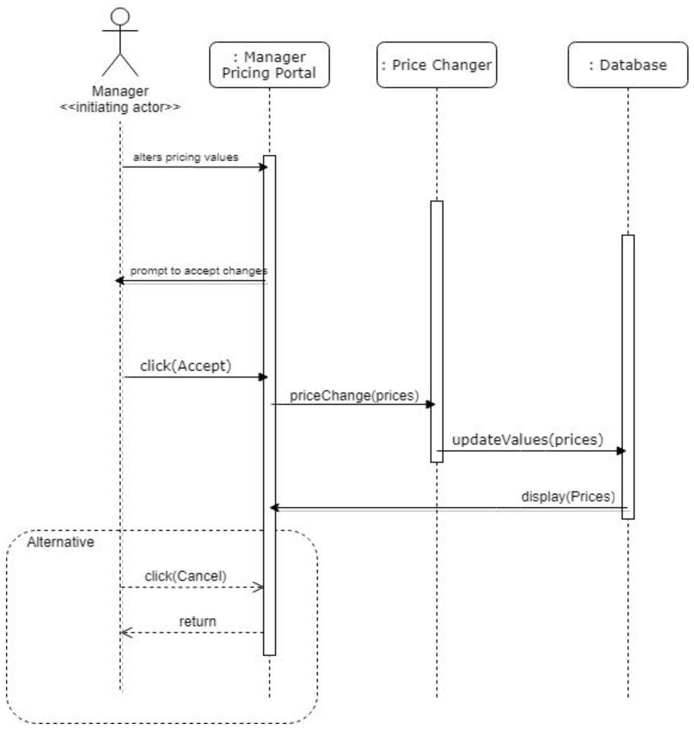
MERN (MongoDB, Express.js, React, & Node.js), the software stack that we are employing, enables us to apply SOLID design principles. React organizes the different parts of the user interface into components. Each of these components performs its own function and only that function; for example, there will be a component with the sole responsibility of displaying the statistics retrieved from the database, thus achieving the single responsibility principle. We plan for the designs of all of these use cases to be minimum viable products that attempt to follow the Principle of Least Astonishment.

a. **Use Case 9: Create Garage Layout**



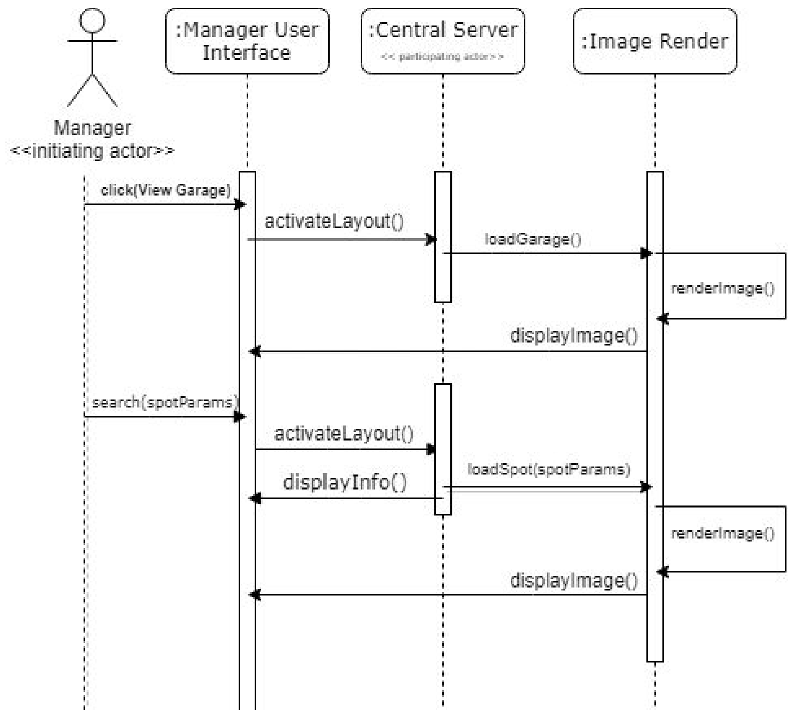
We used the Inversion of Control design principle when the Central Server relinquishes control to the Garage Updater to make and verify the changes to the garage layout data. The Manager User Interface has the responsibility of Delegation where whenever the user interacts with the interface, (in this case clicks the GarageLayout button to prompt the program to render an interactive model of the parking garage), the interface sends a message to the corresponding dependency that can perform this task. This can be described as delegation. The Central Server has the Knowing responsibility of collecting data, such as the garage layout data and its changes. It can also reference other objects such as the Imager Render and the Garage Updater. The Garage Updater updates and verifies the garage data in the Central Server, and the Image Render renders the model of the parking garage using the current data stored in the Server. They are both tasked with a Doing responsibility. It also proves that we used the High Cohesion Principle as we gave each object dependency a different computational task. We are using the LSP as the managers can make any number of changes using the garage layout interface, and the proper garage layout information should be stored and rendered. We use the Low Coupling Principle as the delegates split complex responsibilities such as rendering and displaying the virtual model of the parking garage to the user, between three delegates.

b. Use Case 10: Set Pricing



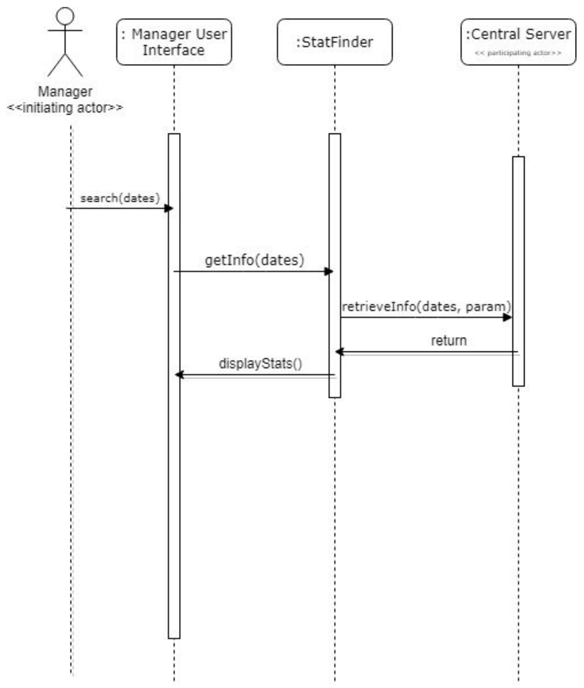
The Database has the Knowing responsibility of memorizing the pricing values of the parking garage which are sent to it by the Price Changer who performs the computations needed to change the prices. The Manager Pricing Portal delegates subtasks such as changing the prices, priceChange(prices), to the Price Changer. This makes the Price Changer the object with the responsibility of Doing. We plan to use the DRY principle when developing this software.

c. Use Case 11: View Garage



We are using SOLID design principles to achieve this use case as well. As with the other use cases, we are implementing each function in a modular way; each component will only perform one task. This is an example of the single responsibility principle. Furthermore, our design also employs the inversion of control principles. The View Garage page of the user interface will only implement the function of displaying the image rendered by the image renderer. The user interface will receive an instance of the rendered image to display rather than calling the Image Render. We use the Low Coupling Principle as the delegates split complex responsibilities such as rendering and displaying the virtual model of the parking garage to the user, between two delegates.

d. Use Case 14: View Garage Usage Statistics

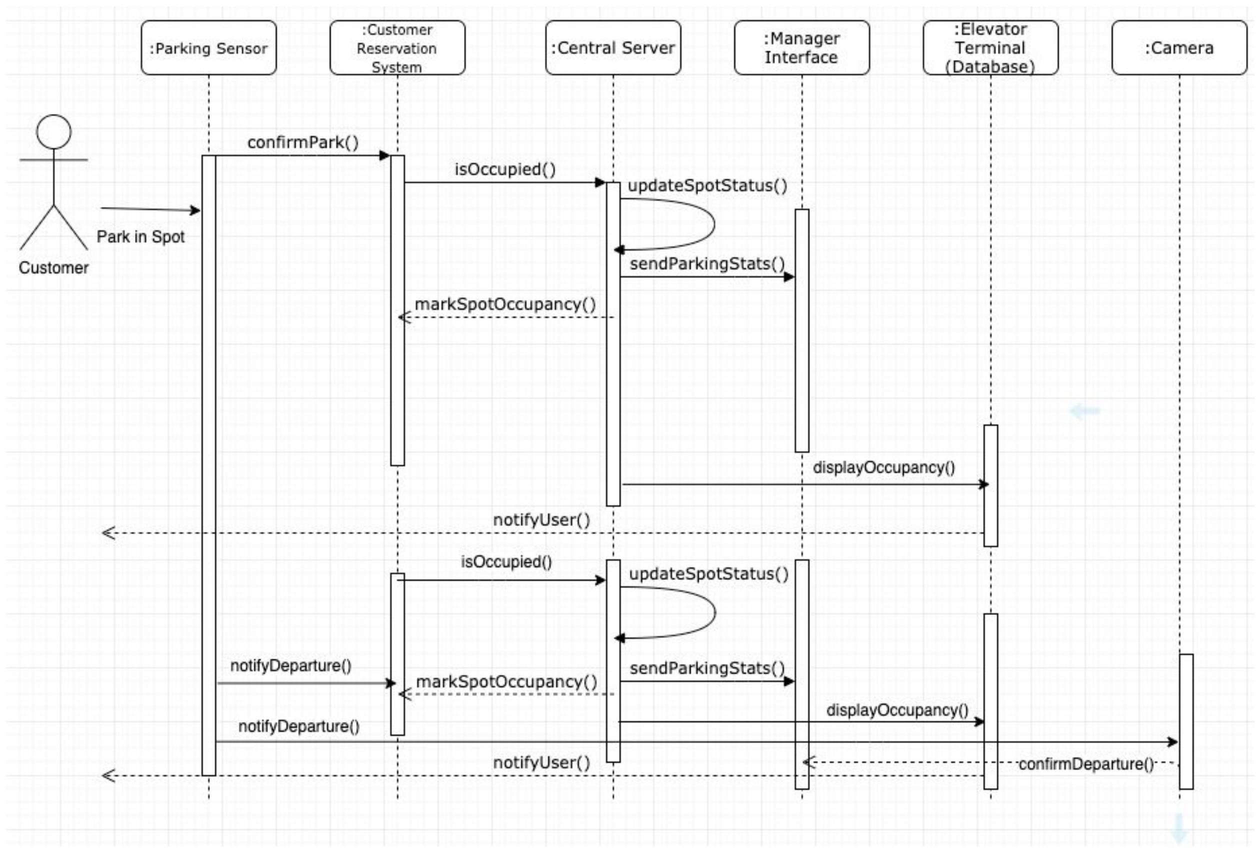


We used the Inversion of Control design principle when the StatFinder relinquishes control to the Central Server to retrieve statistical information. The StatFinder has the responsibility of Doing as it processes data and performing computations needed for creating graphical representations of statistical data. The Central Server has the Knowing responsibility of memorizing the statistical data and is able to reference other parts of the system (actors) when necessary, as it does when it returns to the StatFinder with the retrieved information. The Manager User Interface has the responsibility of Delegation where whenever the user interacts with the interface, (in this case enters dates which the program needs to find and display statistical data for), the interface sends a message to the corresponding dependency that can perform this task. This can be described as delegation. The Interface separation principle could be applied here as well since the Manager User Interface only has two dependencies that have very few tasks, and since the interface itself only has two tasks to carry out. It sends information and displays an image from the information it receives.

Interaction Overview Diagram: manager_interaction_diagram.drawio

Elevator Operation Subgroup

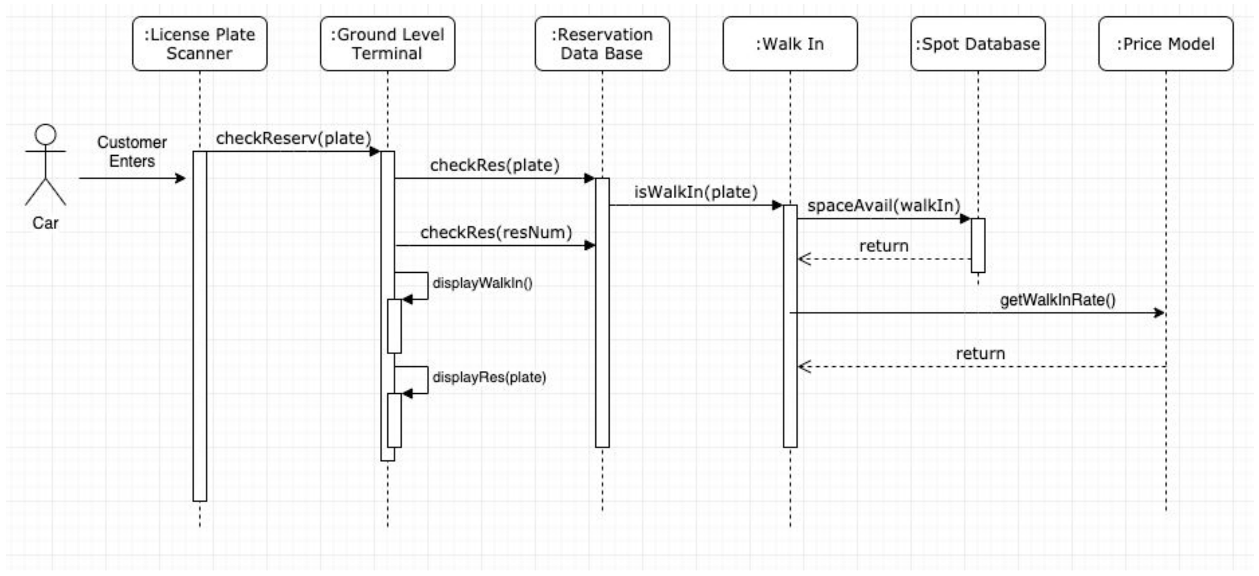
a. Use Case 18: Update Parking Spot Status



Use Case 18 Description:

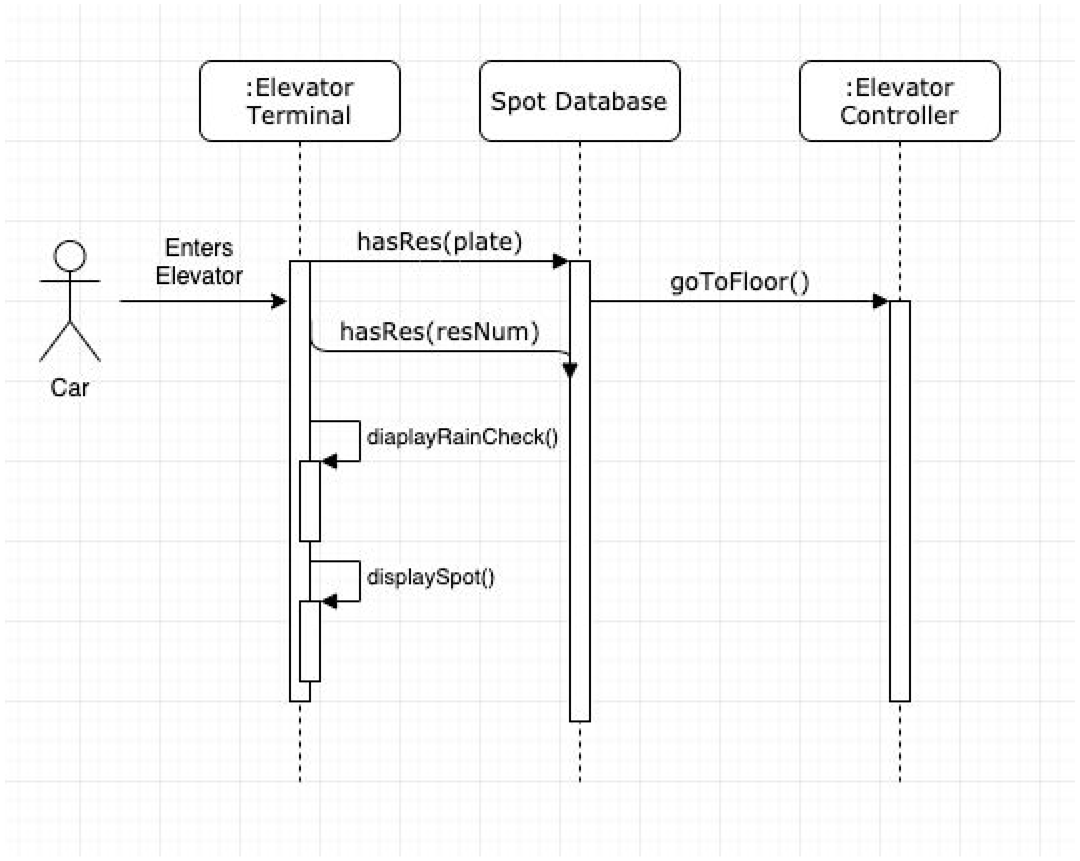
In use case 18 the customer's car interacts with the parking spot sensor. This sets off a sequence in which the parking spot sensor communicates status to the reservation system. The reservations system's job is to inform other systems about the newly occupied or unoccupied spot. Because of this it is a knower in our system. The information about the spot status is made available to the manager interface and elevator terminal via communication of this knowledge.

b. Use Case 19: Park, Entryway Terminal



Use Case 19 Description:
Use case 19 is initiated when the customer enters the garage. When entering the garage the license plate scanner takes a snapshot of the plate and passes this to the ground level terminal. The ground level terminal has the responsibility of *doing* in this sequence of interactions. The ground level terminal is designed using High Cohesion Principle in that it doesn't take on too many computations, it outsources requests for the information it needs and then displays this to the customer at the terminal.
In this sequence we omit the elevator portion of the use case, leaving it for below. Here we see the sequence for a walk-in customer. The modules after the Ground Terminal are all considered knowers. The spot database reports back to the walk-in module about space availability and the price module provides a current rate for the customer.

c. Use Case 19 (Continued): Elevator Terminal

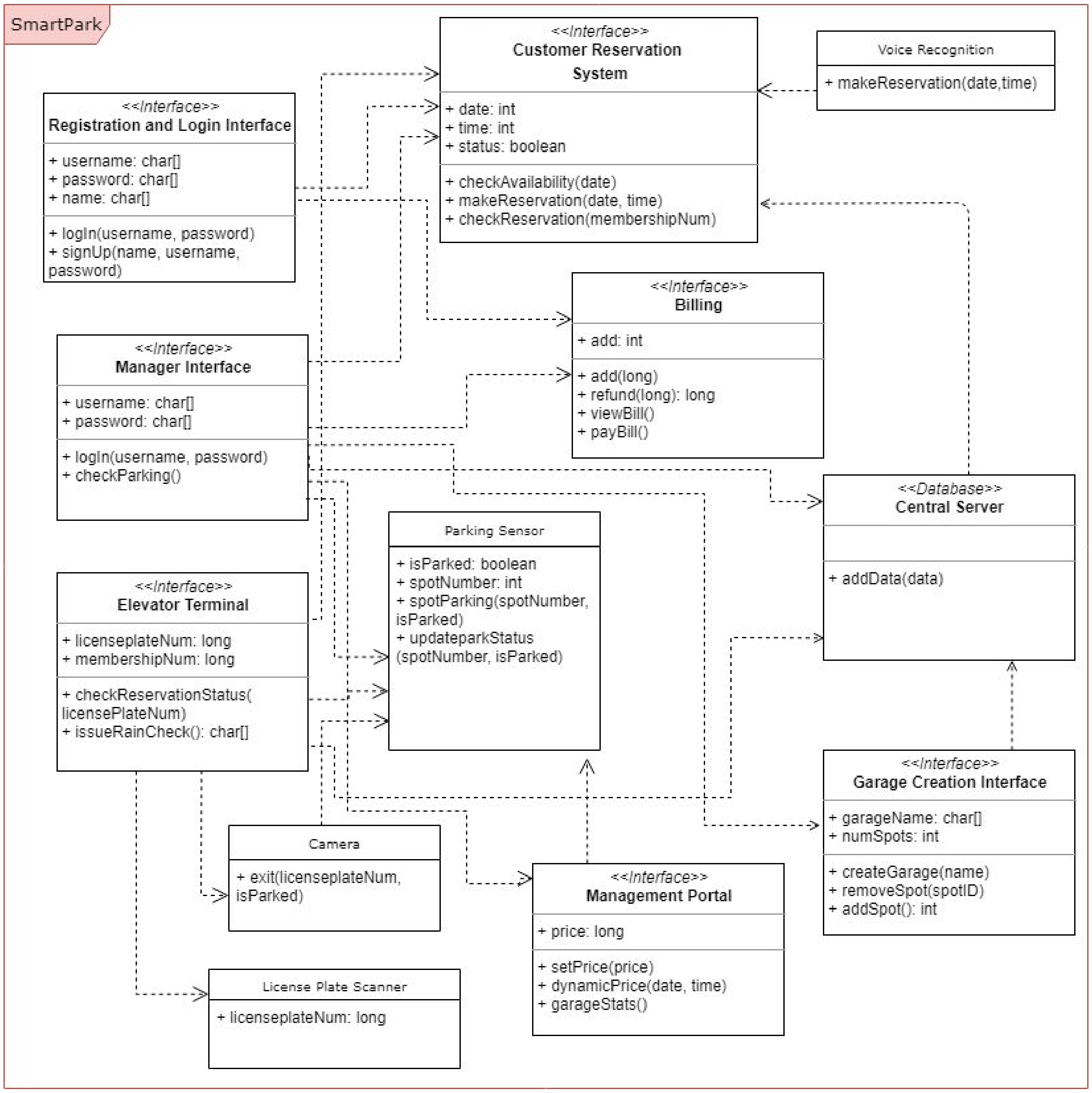


Use Case 19 Description:

When the car advances to the elevator, the terminal within acts as an doer; requesting information from the spot database and the elevator controller. This is another use of the high cohesion principle in which the elevator terminal offloads a great deal of computation and simply references other modules for what it needs. We call on the spot database to act as the knower and provide relevant information back to the elevator terminal. Finally, the elevator controller acts as doer and takes the car to the floor corresponding to the customer's reservation.

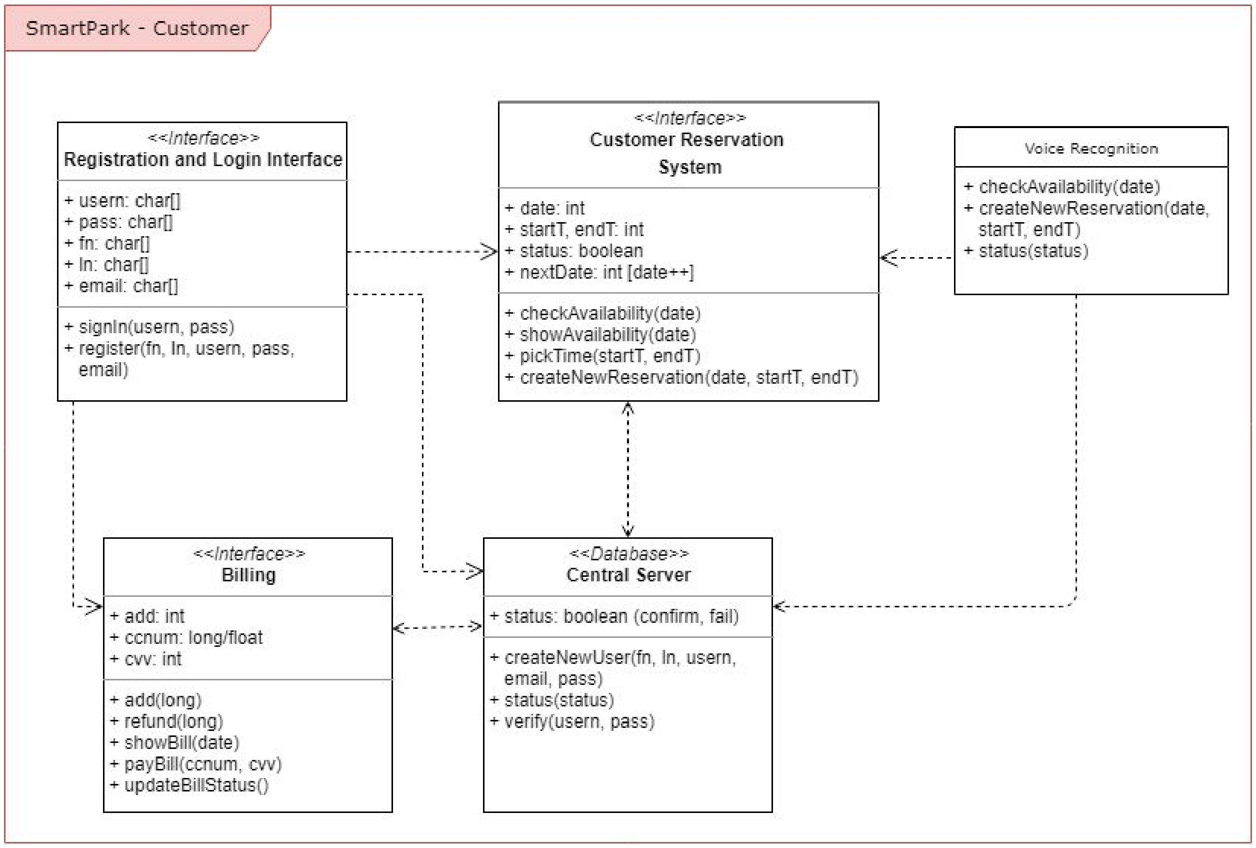
Class Diagram and Interface Specification

Class Diagram



This is the complete class diagram of the SmartPark Parking System. It is a large-scale overview that will be followed up by the specification of its subparts by subgroup. They will also include the complete list of the variables and classes that are seen here along with a list of what their purpose is and how they fit into the functioning of SmartPark. The three main components on the left-hand side are the Registration and User Interface which corresponds to the Customer Registration Subgroup, the Manager Interface which is associated with the Managerial / Administrative Subgroup and the Elevator Terminal managed by the Elevator Operation Subgroup. This class diagram works well with the flow of the team structure already created and allows each subgroup to add/subtract responsibilities through the 'Central Server' that will have tables for each of the data-heavy storage for the subgroups.

1. Customer Registration Subgroup

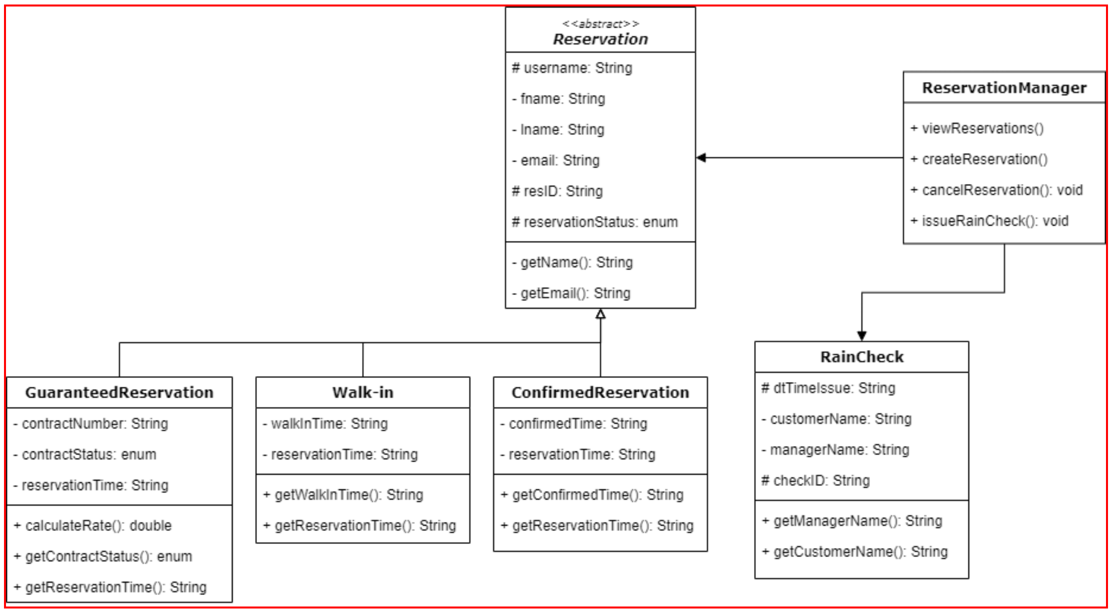


This is the class diagram for the Customer Registration Subgroup. It shows 3 main interfaces: Registration/Login, Reservation, and Billing. A reason why we decided to separate these three interfaces is because of the 'requirements to fulfill to be able to access the pages. For example, to make a reservation the User must have an account and be signed in and to view a bill the User must have already made and completed a reservation to be charged for it. The Registration and Login Interface is the first page encountered by the user as soon as they reach the SmartPark website. Here they can either select to make an account or sign in to an already existing account, the information entered is then cross-checked by the Central Server to see if the Customer Information table has an entry of the same credentials. The next interface is of the Reservation System that allows the user to make a reservation based on their preferred time and date, this is then checked against the Parking Reservation table to see if there will be any open spots during that time. If there are no spots then the user is shown availability for the next closest day. Lastly, the Billing interface allows the user to see their current billing cycle up to a certain day and pay a prior month's bill if desired.

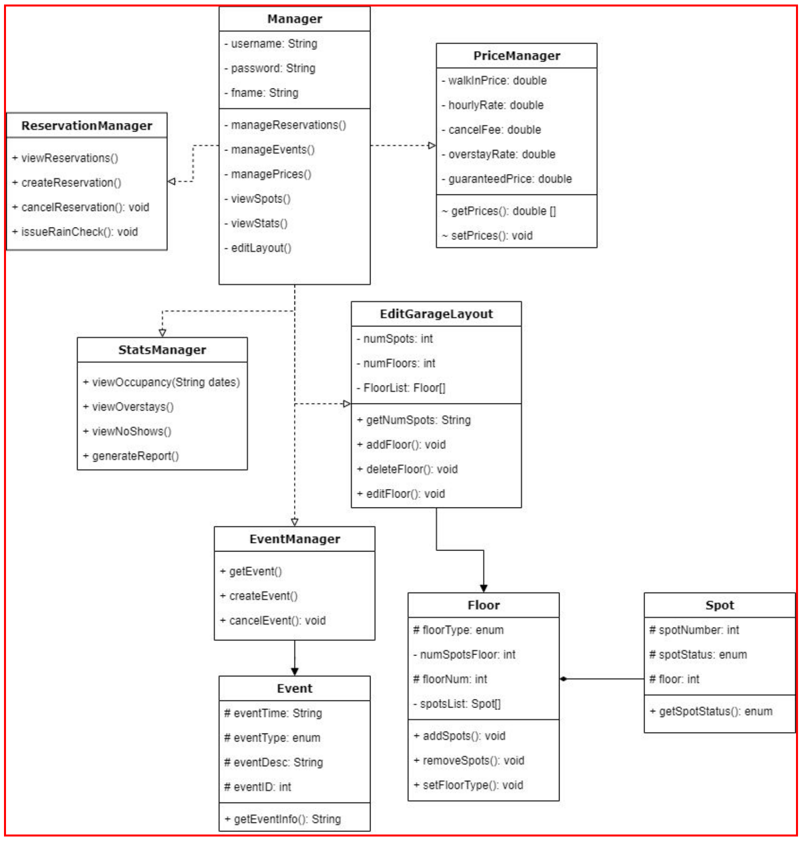
2. Managerial / Administrative Subgroup

Manager Class Diagram

Part 1:



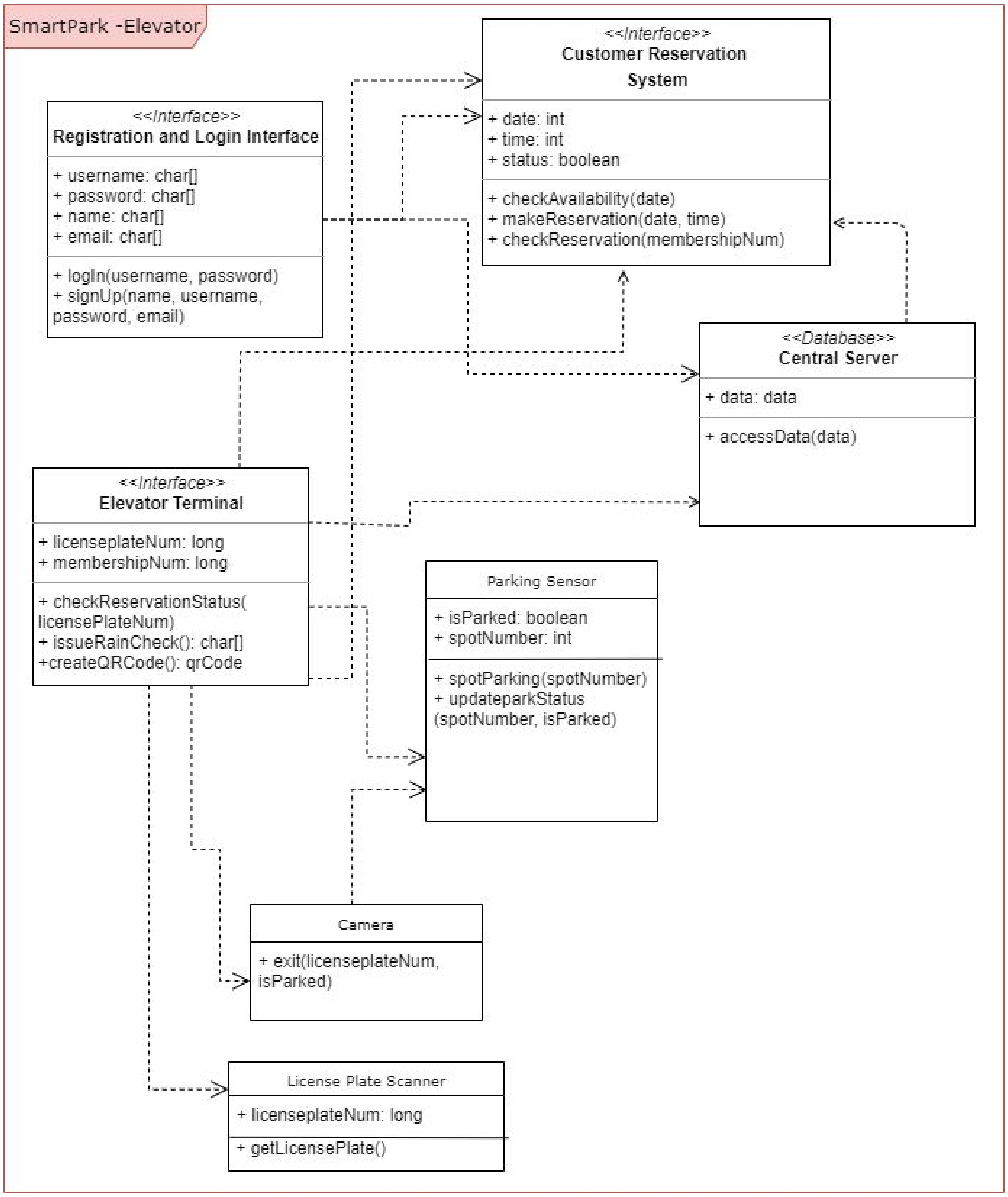
Part 2:



Part 2 of the Manager portion of the class diagram shows exactly what the Management Portal has the ability to access. The Manager class handles and authorizes all administrative credentials through the Manager Interface. It then gives the Portal and therefore the user, access to the Customer Reservation System (ReservationManager) and the Central Server (PriceManager, StatsManager, and EditGarageLayout). The Dynamic Pricing System (PriceManager) affects Billing as well as Managers set the parking prices and fees for customer reservations. The Garage Overview (EditGarageLayout) is the system with the classes that run throughout the Garage Creation Interface. The Manager class also implements the calendar system (EventManager) showing current and future “real-world” events.

Part 1 of the Manager portion of the class diagram shows that the ReservationManager has access to all parts of the Customer Reservation System, including Walk-ins and Guaranteed (Contracted) reservations. It also allows for Managers to create and issue rain checks to customers through the RainCheck class. The License Plate Scanner scans the license plates of all cars that enter the Elevator Terminal. The rain checks can be shown on the Elevator Terminal screen once the Camera and the Parking Sensor send the data to the Terminal that there are no available parking spots when the customer arrives.

3. Elevator Operation Subgroup



The elevator class diagram consists of three interfaces and one connection to the central server database. The interfaces consist of the Elevator Terminal, the main source of user interaction in the elevator subgroup, the Registration and Login Interface, and the Customer

Reservation System. These interfaces are each connected to the Central Server database, which stores all relevant information related to the elevator operation. Information from the database is then crossed checked between the various interfaces to make sure that the elevator operation runs successfully.While the elevator operation is broken up into three different interfaces, each one goes hand in hand with one another. For instance, if a customer's license plate is unable to be scanned, then the elevator terminal interface reports this error to the Customer Reservation System which allows the customer to enter in his or her membership number. When this interaction fails, the user is redirected to the Registration and Login Interface. With this scheme implemented, each group member is able to work on a separate interface and the modules will be connected with one another in the final build of the project.

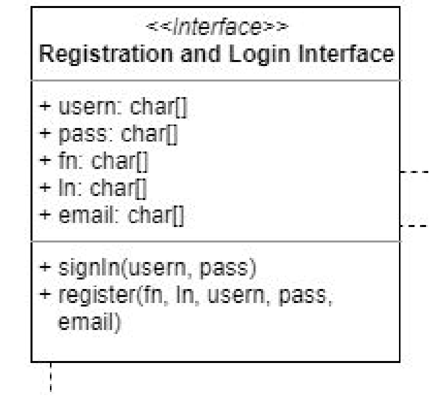
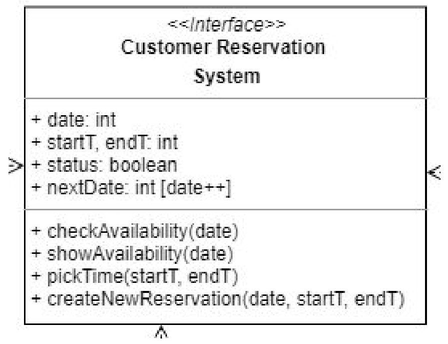
The Elevator Terminal is the interface responsible for scanning the customer's license plate and displaying the user's parking location, provided the garage is not full. Little user interaction is necessary with the elevator terminal, as the license plate scanner is the primary interactor with the elevator terminal. If the license plate could not be read in successfully, then the control is swapped in for the Customer Reservation Interface. This interface prompts the user for his or her membership number to determine whether the customer has a reservation or not. This information is cross checked in the Central Server database. If the user does have a reservation, then control is shifted back to the Elevator Terminal interface, where the usual interaction continues to occur with the user. If, however, the user does not have a reservation, then the control is once again shifted over, this time to the Registration and Login Interface. The customer is then prompted to login to his or her account or to register for the first time. Once the login or registration is complete, control is sent back to the Customer Reservation Interface, where the user can make a reservation. After the reservation is made, then the control is finally shifted back to the Elevator Terminal.

If the garage is full, then the Elevator Terminal will issue a rain check to customers with a reservation. Otherwise, they will be prompted to exit the elevator terminal. Additionally, when the customer receives his or her spot number, the elevator will generate a QR code which can be scanned. The QR code contains an image of the location of the parking spot.

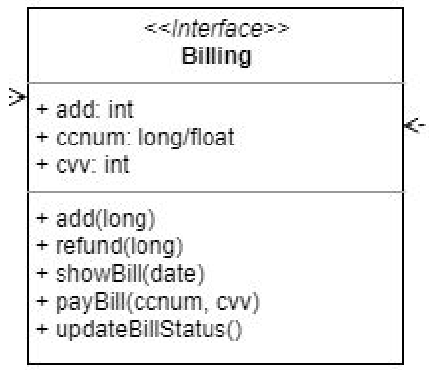
Data Types and Operation Signatures

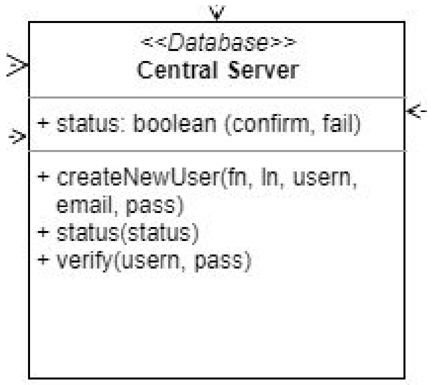
Below are the date types, operation signatures and class explanations by subgroup:

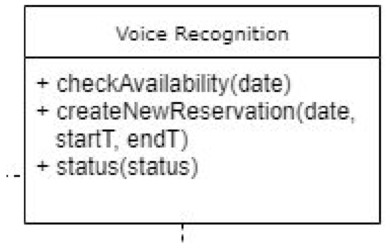
1. Customer Registration

Class	Explanation
 <<Interface>> Registration and Login Interface + usern: char[] + pass: char[] + fn: char[] + ln: char[] + email: char[] + signIn(usern, pass) + register(fn, ln, usern, pass, email)	Data Types: - Usern (string) - A username that is unique to every customer - Pass (string) - A password that is unique to every customer, associated with the username and can have capital-case, lower-case letters, symbols, and numbers - Fn (string) - The user's first name - Ln (string) - The user's last name - Email (string) - The user's email that will be used to send confirmation emails and timely updates Operation Signatures: - signIn(usern, pass) - A function that takes in the user's username and password and searches the Central Server table of customer information for an exactly corresponding entry - register(fn, ln, usern, pass, email) - A function that takes in the user's first and last name, desired username, password and email address
 <<Interface>> Customer Reservation System + date: int + startT, endT: int + status: boolean + nextDate: int [date++] + checkAvailability(date) + showAvailability(date) + pickTime(startT, endT) + createNewReservation(date, startT, endT)	Data Types: - Date (int) - Desired date of reservation - StartT (int) - Start time of parking - EndT (int) - Ending time of parking - Status (boolean) - True for successfully reserved spot - False for unsuccessful reservation - nextDate (int) - Increments date to find the next day with available spots

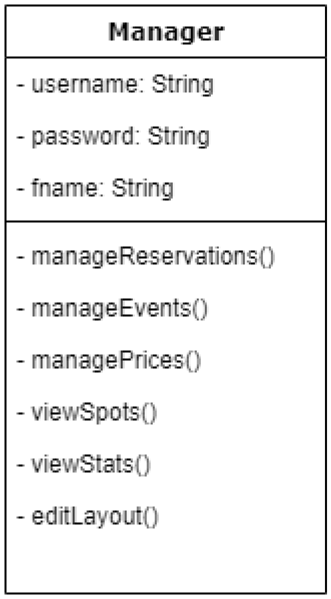
	Operation Signatures: - checkAvaliability(date) - A function that checks the number of available parking spots for a specific indicated date - showAvaliability(date) - Displays the times that correspond to open parking spots on the given date - pickTime(startT, endT) - The user picks a start and end time between they want to park - createNewReservation(date, startT, endT) - Updates the Central server with a new entry in the Reservations table that will hold a parking spot for that user on a given date and specified times

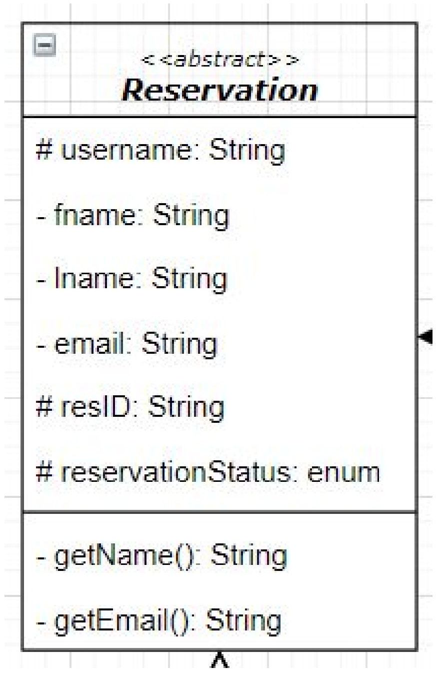
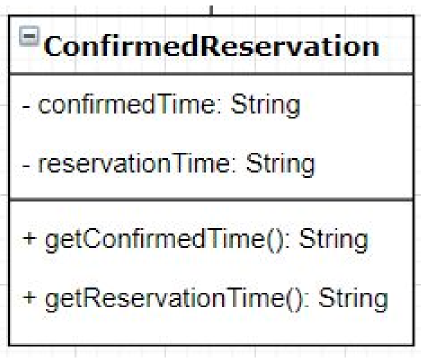
 <<Interface>>
Billing
+ add: int
+ ccnum: long/float
+ cvv: int
+ add(long)
+ refund(long)
+ showBill(date)
+ payBill(ccnum, cvv)
+ updateBillStatus()
	Data Types:
	- Add (int)
	    - Adds a new expense to the user's account
	- Ccnum (long/float)
	    - Holds the customer's credit card number
	- Cvv (int)
	    - Holds the customer's cvv pin
	Operation Signatures:
	- add(long)
	    - Adds and retotals the complete bill for the user from the beginning of the month until a specified date
	- refund(long)
	    - Issues a money return transaction in case of a rain check
	- showBill(date)
	    - Displays the user's bill from the the beginning of the month until the given date
	- payBill(ccnum, cvv)
	    - Allows the user to pay online with their credit card information
	- updateBillStatus()
	    - Changes the outlying balance for the user to zero in case of a successful payment

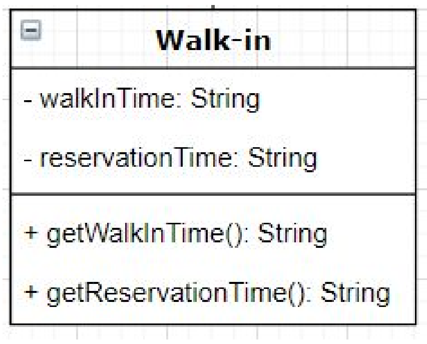
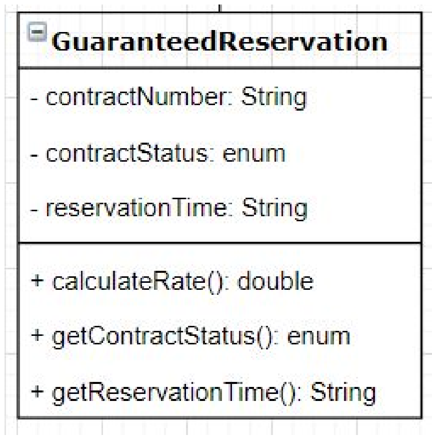
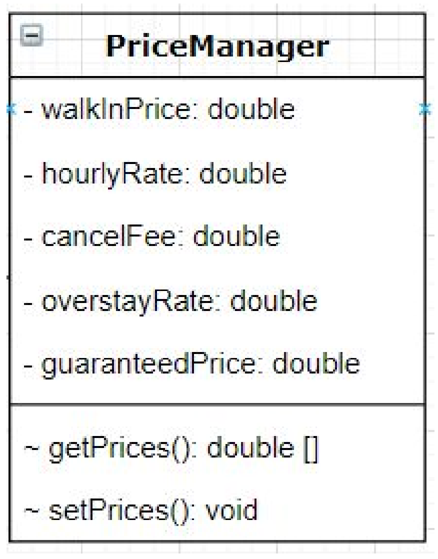
 <<Database>> Central Server + status: boolean (confirm, fail) + createNewUser(fn, ln, usern, email, pass) + status(status) + verify(usern, pass)	Data Types: - Status (boolean) - True for match to an entry - False for not match to an entry Operation Signatures: - createNewUser(fn, ln, usern, email, pass) - Creates a new entry in the customer information table - status() - Verifies successful or unsuccessful account creation - verify(usern, pass) - Searches for a match for username and password to allow a user to sign in

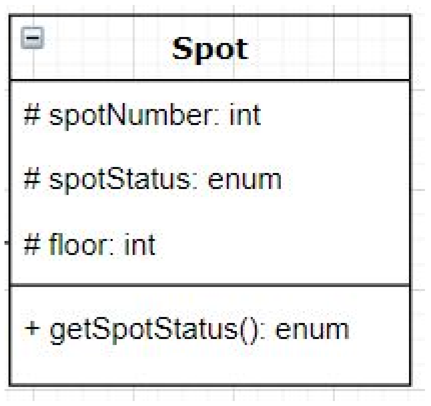
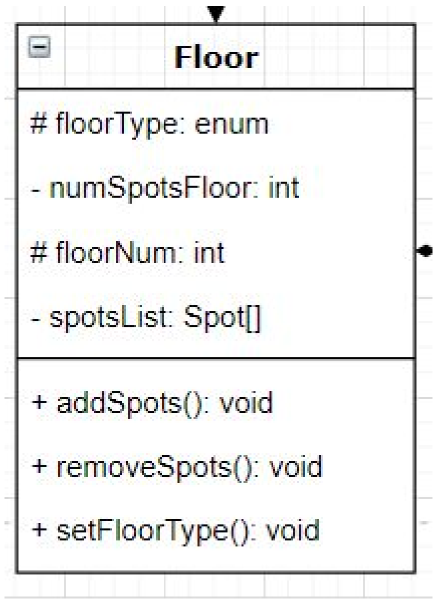
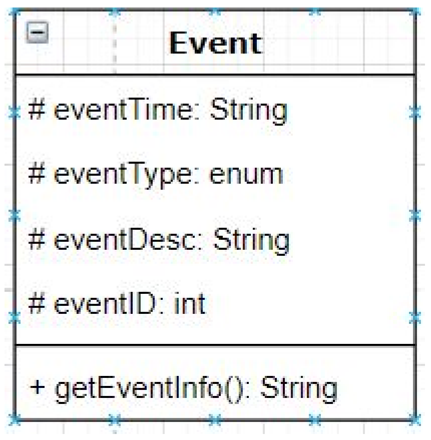
 Voice Recognition
+ checkAvailability(date)
+ createNewReservation(date, startT, endT)
+ status(status)
	Data Types:
	- N/A
	Operation Signatures:
	- N/A

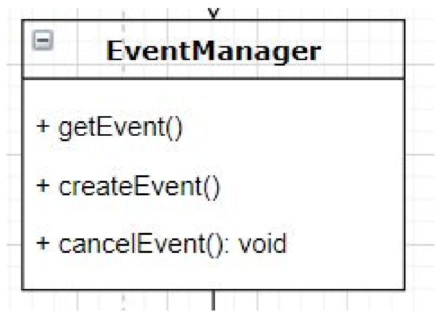
2. Managerial / Administrative

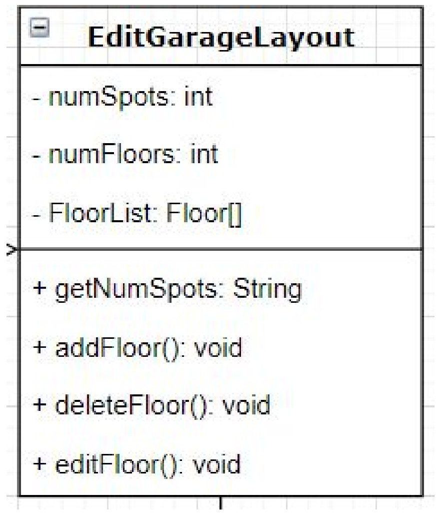
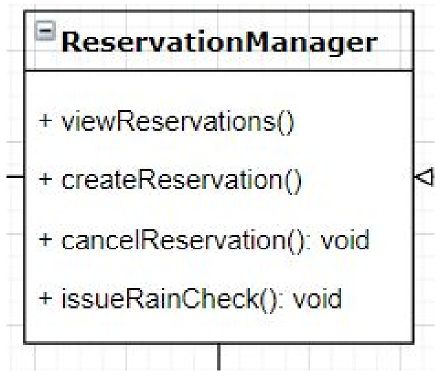
Class	Explanation
 Manager - username: String - password: String - fname: String - manageReservations() - manageEvents() - managePrices() - viewSpots() - viewStats() - editLayout()	Data Types: - Private username: String A string for the user's username - Private password: String A string for the user's password - Private fname: String A string containing the user's name Operation Signatures: - Private manageReservations(callback): void Invokes the ReservationManager class to perform functions on reservations. Will call the appropriate method in ReservationManager based on user input. - Private manageEvents(callback): void Calls the EventManager class to perform actions on events. Will call the appropriate function in EventManager based on user input. - Private manageEvents(callback): void

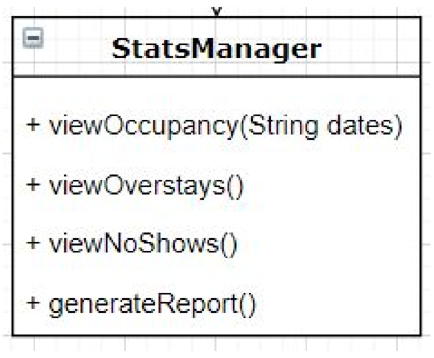
	Calls the EventManager class to perform actions on events. Will call the appropriate function in EventManager based on user input. - Private viewSpots(): void Displays status of all spots in numerical order - Private editLayout(callback): void Calls the editLayout class to perform actions on floors and spots. Will call the appropriate function in the EditGarageLayout class based on user input.
 <<abstract>> Reservation # username: String - fname: String - lname: String - email: String # resID: String # reservationStatus: enum - getName(): String - getEmail(): String	Data Types: - Protected String username: username of the customer making the reservation - Private String fname: First name of the customer - Private String lname: Last name of the customer - Private String email: Email address of the customer - Protected String resID: reservation ID of the reservation - Protected enum reservationStatus: Status of the reservation: confirmed, paid, cancelled Operation Signatures: - Private String getName(): Returns the full name of the customer - Private String getEmail(): Returns the customer email address associated with the account
 ConfirmedReservation - confirmedTime: String - reservationTime: String + getConfirmedTime(): String + getReservationTime(): String	Data Types: - Private String confirmedTime: Time at which the reservation was confirmed - Private reservationTime: Time at which the reservation was made for Operation Signatures: - Public String getConfirmedTime(): Returns the time the reservation was confirmed - Public String getReservationTime(): Returns the time the reservation was made for

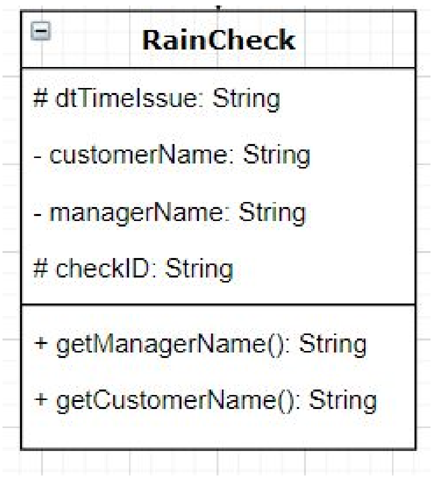
 Walk-in - walkInTime: String - reservationTime: String + getWalkInTime(): String + getReservationTime(): String	Data Types: - Private String walkInTime: Time at which the reservation was made for the walk-in - Private String reservationTime: Time for the duration of the reservation Operation Signatures: - Public String getWalkInTime(): Returns a String with the Walk in Time - Public String getReservationTime(): Returns a String with the Reservation Time
 GuaranteedReservation - contractNumber: String - contractStatus: enum - reservationTime: String + calculateRate(): double + getContractStatus(): enum + getReservationTime(): String	Data Types: - Private String contractNumber: Contains the customer's contract number - Private enum contractStatus: Contains the status of the contract: paid, unpaid, expired - Private String reservationTime: Time the reservation is held for based on the contract Operation Signatures: - Public double calculateRate(): Calculates and returns the rate for guaranteed reservations - Public enum getContractStatus(): Returns the status of the contract: paid, unpaid, expired - Public String getReservationTime(): Returns a String with the Reservation Time
 PriceManager - walkInPrice: double - hourlyRate: double - cancelFee: double - overstayRate: double - guaranteedPrice: double ~ getPrices(): double [] ~ setPrices(): void	Data Types: - Private double walkInPrice: Contains the parking garage price for walk-ins - Private double hourlyRate: Contains the hourly rate for reserved customers of the parking garage - Private double cancelFee: Contains the fee for reservation cancellations - Private double overstayRate: Contains the fee for overstays - Private double guaranteedPrice: Contains the price for guaranteed reservations Operation Signatures: - Package double[] getPrices(): Calculates and returns the prices for walk-ins and reservations, and the fees for cancellations and overstays - Package void setPrices(): Sets the parking garage prices to new values calculated using the Manager's input values

 Spot # spotNumber: int # spotStatus: enum # floor: int + getSpotStatus(): enum	Data Types: - Protected int spotNumber: The number of the parking spot - Protected enum spotStatus: Contains the status of the parking spot: occupied, available - Protected int floor: The number of the floor that the parking spot is located on Operation Signatures: - Public enum getSpotStatus(): Returns the status of the parking spot
 Floor # floorType: enum - numSpotsFloor: int # floorNum: int - spotsList: Spot[] + addSpots(): void + removeSpots(): void + setFloorType(): void	Data Types: - Protected enum floorType: The type of the floor in the parking garage: walk-in, reservation-only - Private int numSpotsFloor: The number of parking spots on the garage floor - Protected int floorNum: The number of the garage floor - Private Spot[] spotsList: An array that holds the information for each spot Operation Signatures: - Public void addSpots(): Creates and adds a Spot Object to the chosen floor when a user adds a spot to the virtual garage layout - Public void removeSpots(): Removes a Spot Object to the chosen floor when a user adds a spot to the virtual garage layout - Public void setFloorType(): Sets the type of the garage floor: walk-in, reservation-only
 Event # eventTime: String # eventType: enum # eventDesc: String # eventID: int + getEventInfo(): String	Data Types: - Protected String eventTime: The time the event is scheduled for - Protected enum eventType: The type of event: national holiday, religious holiday, concert, show, picnic, etc. - Protected String eventDesc: The description of the event - Protected int eventID: The identification number for the event Operation Signatures: - Public String getEventInfo(): Returns the information for the event that is saved in the managerial system as Strings

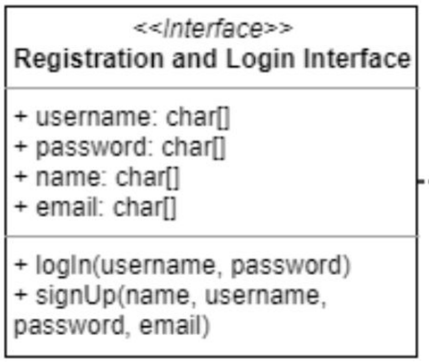
 EventManager
+ getEvent()
+ createEvent()
+ cancelEvent(): void
	Operation Signatures:
	- Public getEvent(): Returns the specified event
	- Public createEvent(): Creates a new event in the calendar
	- Public void cancelEvent(): Cancels an event by removing it from the calendar and deleting its information in the server

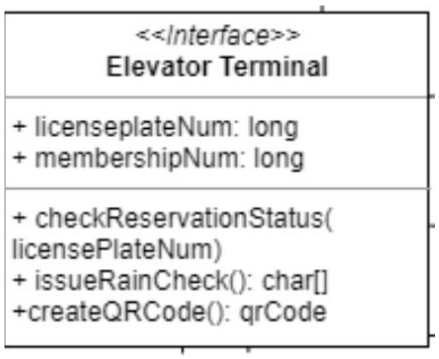
 EditGarageLayout - numSpots: int - numFloors: int - FloorList: Floor[] + getNumSpots: String + addFloor(): void + deleteFloor(): void + editFloor(): void	Data Types: - Private int numSpots: The number of parking spots on the current floor - Private int numFloors: The number of the garage floor - Private Floor[] FloorList: An array that holds the information for each floor Operation Signatures: - Public String getNumSpots(): Returns the number of parking spots on the floor in the form of a String - Public void addFloor(): Creates and adds a floor to the garage when a user adds a floor to the virtual garage layout - Public void deleteFloor(): Removes a floor to the garage when a user adds a floor to the virtual garage layout - Public void editFloor(): Edits a floor when a user edits it in the virtual garage layout
 ReservationManager + viewReservations() + createReservation() + cancelReservation(): void + issueRainCheck(): void	Operation Signatures: - Public viewReservations(): Opens the reservation list in the GUI - Public createReservation(): Creates a new customer parking reservation - Public void cancelReservation(): Cancel a customer parking reservation - Public void issueRainCheck(): Issues a rain check to a customer when they have made a reservation, but there are no parking spots currently available

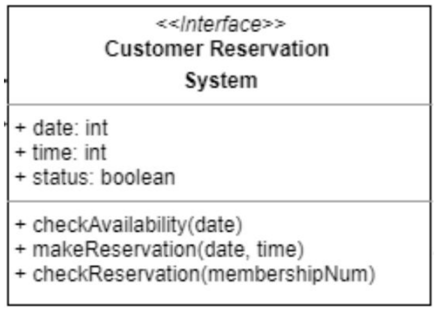
 StatsManager
+ viewOccupancy(String dates)
+ viewOverstays()
+ viewNoShows()
+ generateReport()
	Operation Signatures:
	- Public viewOccupancy(String dates): Opens the parking garage occupancy statistics for the specified dates in String dates, within the Manager Interface
	- Public viewOverstays(): Opens the overstay statistics for the parking garage in the Manager Interface
	- Public viewNoShows(): Opens the no-show statistics for the parking garage in the Manager Interface
	- Public generateReport(): Generates a statistics report for the parking garage for the Manager to view on the Interface

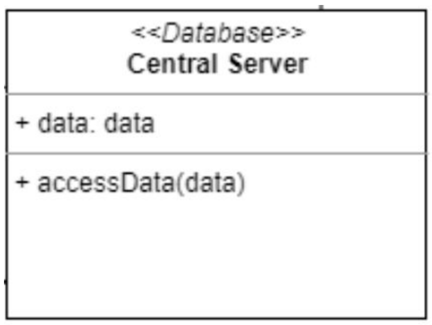
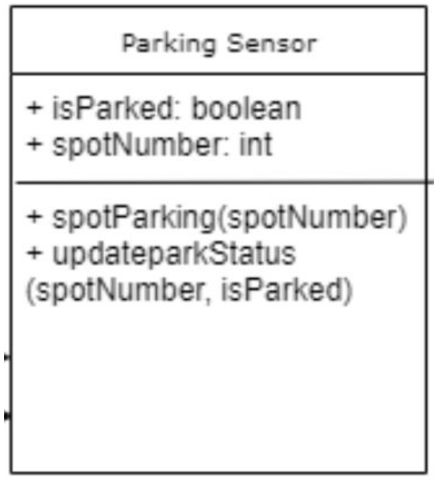
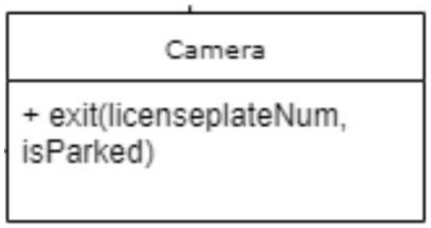
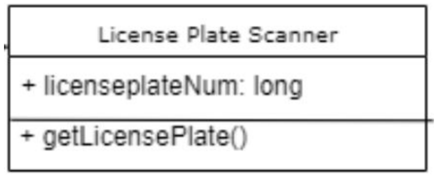
 RainCheck # dtTimeIssue: String - customerName: String - managerName: String # checkID: String + getManagerName(): String + getCustomerName(): String	Data Types: - Protected String dtTimeIssue: A String containing the date and time that the rain check is issued - Private String customerName: A String containing the customer's full name - Private String managerName: A String containing the manager's full name - Protected String checkID: A String containing the identification number of the issued rain check Operation Signatures: - Public String getManagerName(): Returns the managerName String - Public String getCustomerName(): Returns the customerName String

3. Elevator Operation

Class	Explanation
 <<Interface>> Registration and Login Interface + username: char[] + password: char[] + name: char[] + email: char[] + logIn(username, password) + signUp(name, username, password, email)	Data Types: - username (String) - Unique identifier assigned to a customer, the customer uses his or her username to access the SmartPark account. - password (String) - The user's key in order to gain access to his or her account. The password can be made up of letters, numbers, or special characters. - name (String)

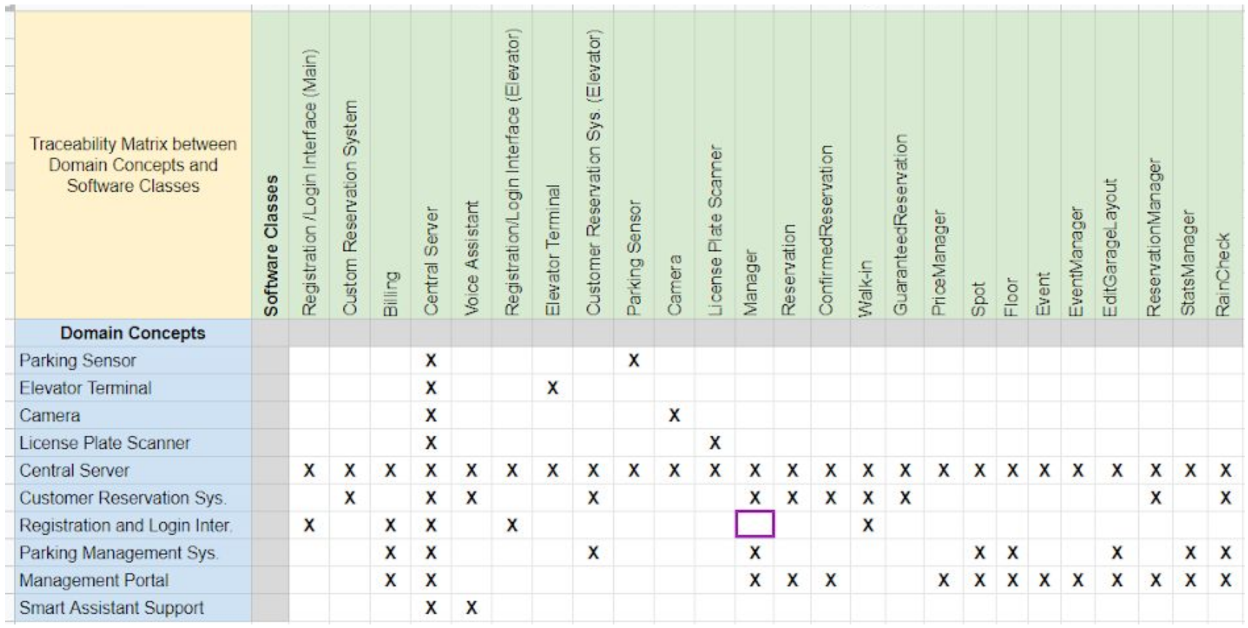
	- The customer’s name, consisting of first and last. One of the necessary fields for creating a SmartPark account. - email (String) - The customer’s email address, a necessary field for the creation of the SmartPark account. The user’s email address will be used to notify a customer about his or her reservation, when a bill arrives, or outstanding payments to his or her SmartPark account. Operation Signatures: - logIn(username, password) - The user’s login information is provided to the system, where the username can optionally be replaced with the customer’s email address. The system then does a check to see if the information provided by the user matches the information in the database, and a response is returned to the user. The response is either success (correct login combination, the user is granted access) or failure (incorrect login combination or username not found in the system). - signUp(name, username, password, email) - A function that takes in the user’s name, desired username, password and email address. If there is a matching email or username in the system, then the system notifies the user that the username and/or password have already been used.
 <<Interface>> Elevator Terminal + licenseplateNum: long + membershipNum: long + checkReservationStatus(licensePlateNum) + issueRainCheck(): char[] +createQRCode(): qrCode	Data Types: - licensePlateNum (long) - The customer’s license plate number scanned in by the license plate reader. - membershipNum (long) - Another unique identifier given to the customer, the customer’s membership number allows for quick entry in order to determine whether or not the customer has a reservation or not. Operation Signatures: - checkReservationStatus(licensePlateNum) - Using the license plate scanned in by the license plate reader as well as information

	provided by the central database and customer reservation in order to determine if the customer has a reservation or not. - issueRainCheck() - When the garage is full - all of the spots are currently in use- then the elevator terminal will issue a rain check to customers that had a spot reserved. - createQRCode() - When a customer's reservation is successfully found and the spot is available, a QR Code is sent to the user on the elevator terminal's screen. This QR code can be used for more information regarding the customer's parking spot as well as information regarding the reservation.
 <<Interface>> Customer Reservation System + date: int + time: int + status: boolean + checkAvailability(date) + makeReservation(date, time) + checkReservation(membershipNum)	Data Types: - date (int) - The date for which the customer's reservation is created. - time (int) - The length of time a customer's reservation is for. - status (boolean) - A true or false value indicating the reservation status of a spot. If the reservation was successful, true is returned. If a spot could not be reserved - for instance there are contiguous reservations - then the status is false. Operation Signatures: - checkAvailability(date) - A function that checks the number of available parking spots for a specific indicated date. - makeReservation(date, time) - Make a reservation for a particular date and time if available. - checkReservation(membershipNum) - Displays the reservation for the customer given the membership number. If the customer does not have a reservation or the membership number does not exist, an error is returned.

 <<Database>> Central Server + data: data + accessData(data)	Data Types: - data (data) - Information stored in the database. Could take on a variety of forms (int, float, char, varchar, etc.) Operation Signatures: - accessData(data) - Function to access the data in the database. Used to check values entered by the user to see if they exist in the database or not, or used to check the status of certain values.
 Parking Sensor + isParked: boolean + spotNumber: int + spotParking(spotNumber) + updateparkStatus (spotNumber, isParked)	Data Types: - isParked (boolean) - Determines if a parking spot is occupied or not. If a spot is occupied, then the value for isParked is true. Otherwise, the value is false. - spotNumber (int) - Gives the spot number for a specific parking spot. Operation Signatures: - spotParking(spotNumber) - Checks if a specific parking spot is occupied or not. Given the spot number, the database is checked to see if a spot is occupied or not. The status of the parking spot is then returned to the user. - updateParkStatus(spotNumber, isParked) - Given a spot number and occupancy status, the status of a parking spot is changed to the specified status.
	Data Types: - N/A Operation Signatures: - exit(licensePlateNum, isParked) - Given a license plate number and its parking status, determines whether a vehicle has entered or exited the garage.
 License Plate Scanner + licenseplateNum: long + getLicensePlate()	Data Types: - licensePlateNum (long) - The license plate number of the vehicle scanned in the elevator. Operation Signatures:

	- getLicensePlate() - Obtains the license plate number of the vehicle in the elevator.

Traceability Matrix



Traceability Matrix between Domain Concepts and Software Classes	Software Classes	Registration /Login Interface (Main)	Custom Reservation System	Billing	Central Server	Voice Assistant	Registration/Login Interface (Elevator)	Elevator Terminal	Customer Reservation Sys. (Elevator)	Parking Sensor	Camera	License Plate Scanner	Manager	Reservation	ConfirmedReservation	Walk-in	GuaranteedReservation	PriceManager	Spot	Floor	Event	EventManager	EditGarageLayout	ReservationManager	StatsManager	RainCheck
Domain Concepts																										
Parking Sensor					X					X																
Elevator Terminal					X			X																		
Camera					X						X															
License Plate Scanner					X							X														
Central Server		X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
Customer Reservation Sys.			X		X	X			X				X	X	X	X	X							X		X
Registration and Login Inter.		X		X	X		X									X										
Parking Management Sys.				X	X				X				X						X	X			X		X	X
Management Portal				X	X								X	X	X			X	X	X	X	X	X	X	X	X
Smart Assistant Support					X	X																				

Customer Traceability Matrix Description

Registration/Login Interface: The main terminal that will be used to collect the information that the user inputs to send to the central server to cross check.

Custom Reservation System: The subsystem that deals with the reservations that need to be made by the customer, it cross checks the dates and times of the proposed reservation until confirmed by both the user and the central server.

Billing: This is where all the money balance of the customer takes place. This is where we will generate and log all the costs for every reservation made, grace period extension charge, overstay fee and raincheck credit.

Central Server: The database that will have tables for every subgroup that will be referenced for account information

Managerial / Administrative Traceability Matrix Description

The Manager Group always has to use the Management Portal to access any of the data stored in the Central Server. Thus, almost all of our classes are related to those two Domain Concepts. The Walk-in and GuaranteedReservation classes are not connected to the Management Portal because contracts and walk-in reservations are mainly created using the Customer's User Interface. The domain concepts that we originally defined were split into the software classes based on implementation. As we implemented the concepts, we found that multiple classes would use and interact with the functionalities defined by the domain concepts.

Manager: The manager class will access the central server to access information to display for the administrator. The manager class also needs to interact with the customer reservation system to view customer profiles and reservations. The manager class will also view information from the parking management system to view the status of each of the spots. The manager class will serve as the controller component of the management portal.

Reservation: The Reservation class will access the customer reservation system to find the customer information associated with each reservation. The Reservation class also needs to access the Parking Management System to see which spots are open and the overstay status of a reservation.

ConfirmedReservation: The managers interacts with the Confirmed Reservation because the managers will be able to see how much of the garage is reserved in advance to see how well their business is doing. To see if a reservation's status the manager will need access to the Central Server and Customer Reservation Systems.

Walk-in: The walk-in class interacts with the central server, registration and login interface, and customer reservation system. The class first checks the customer reservation system to see if a spot is available. If it is, then a customer can use the customer reservation system to reserve that spot as a walk in. The central server is what the classes use to communicate to the other classes.

GuaranteedReservation: Draws up a contract once a customer makes a reservation. Accesses the Customer Reservation System to gather details about the reservation and customer.

PriceManager: The Price Manager class accesses the management portal to receive the amount that the manager would enter as the price for a reservation.

Spot: The spot class interacts with the parking management system and the manager portal. The managers can edit a spots characteristics like spot number and status and these changes are updated in the parking management system.

Floor: The floor class interacts with the parking management system and the manager portal. The managers can edit a floors characteristics like floor number and status and these changes are updated in the parking management system.

Event: If there is a local event the managers can change or discount the price based on the event.

EditGarageLayout: This class interacts with the management portal and the parking management system. Managers can edit the garage layout in their management portals and these changes will be updated in the parking management system.

ReservationManager: It interacts with the Customer reservation system and the management Portal. The manager through the Management Portal can see what spots have been reserved by customers and that data is provided by the Customer Reservation System.

StatsManager: Interacts with the Management Portal and the Parking Management System. The manager can see statistics provided by the Parking Management System in the Management Portal

RainCheck: The RainCheck is issued when there are no spots available for the customer with a reservation, which requires information from the Parking Management System.

The RainCheck class uses information from the Customer Reservation System to determine which customer the RainCheck needs to be given to.

Elevator Group Traceability Matrix Description

Central Server: The database will store information for a variety of different entity sets and relationships. The specific entity sets for the elevator would include data regarding the various sensors- the parking sensor, the camera, the license plate scanner- as well the customer reservation information and information regarding the parking spots. The central server connection with the elevator will be used to search and update information on reservations and parking spots, and to make decisions based on what is returned.

Elevator Terminal: The interface for the elevator itself. The elevator terminal uses information from the customer reservation system and the central server in order to determine the spots that the customer has a reservation and the spot is available.

Registration/Login Interface for Elevator: The registration/login interface for the elevator is used in tandem with the central server and the registration and login interface used for the SmartPark system. This interface allows the customer to enter in his or her membership number, which is then cross checked with the central server. If the membership number is in the system, then the interaction is complete with the user in the elevator in the event that the license plate could not be scanned.

Customer Reservation System for Elevator: The customer reservation system is used to determine if the customer whose license plate was scanned or membership number was entered has a reservation. If the customer has a reservation, then the elevator brings the user to the floor where his or her parking space is. If, however, the customer does not have a reservation, then the customer is asked to leave the elevator terminal and wait at the walk-in terminal.

Parking Sensor: The parking sensor determines whether or not a vehicle is occupying a parking spot or not. This information is then sent to the central server which retrieves it, and sends the information to various locations such as the elevator terminal when prompted for that specific spot or to the management portal where the managers can view the garage occupancy.

Camera: The camera's use in the elevator terminal is to determine when vehicles enter or exit the parking garage or the parking elevator.

License Plate Scanner: The license plate scanner is used to scan the customer's vehicle's license plate to determine if the customer has a reservation or not. The license plate is then used in the central server and the customer reservation system to cross check the reservation, and then appropriate decisions are made by the elevator terminal.

RainCheck: The rain check logic is used to distribute a rain check to customers when the parking garage is full. If the garage reaches occupancy and the customer has a reservation, then a rain check is given to the customer. This updates the customer reservation system accordingly.

System Architecture and System Design

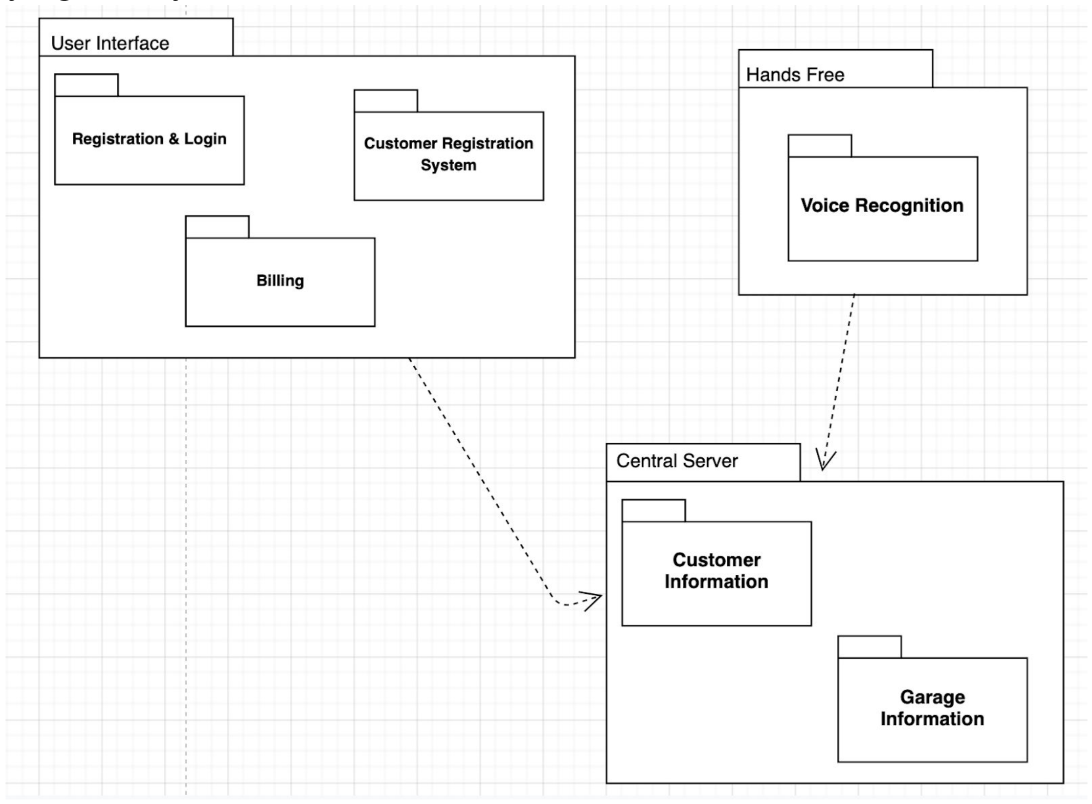
Customer Registration Subgroup

Architectural Styles

The customer subgroup will be utilizing the Component Based Development Style. This style allows individuals in the subgroup to develop independent components that can be linked together for a functional product. We believe that this approach will be best because the members on our team consist of those that are proficient in different building blocks that our software will be dependent on. Since our project will consist of a front end service that users interact with, those that excel in React and Javascript will be able to develop the website component of the architecture. On the other hand, the database will need to be created and set up in a way that allows for easy interaction with the website which entails a database component. Lastly, a connection to the managerial side must be made in order to update spot locations as well as access locations so this will be another component.

Another reason we wanted to choose this architectural style is its ability to help our team break down key design classes which we have already made in previous reports and incorporate them in the system when we are building subsystems. For instance, keeping parts of the system in specified components will help us understand what each component is required to do. Not only that but we start to see what other smaller components are needed to complete larger ones. Overall, we expect these components to be reused and refined as technology and technology stacks improve so keeping our architecture in a building block format will allow us to make changes that show us the effects that our changes can have on other components.

Identifying Subsystems



Shown above is our package diagram which shows the packages we know need to be made and the dependencies that each one shares with others.

The User Interface subsystem which consists of the 3 different packages. The first package, titled *Registration + Login*, will handle a new account being created as well as account verification when a user tries to log in. I paired this along with the *Billing* package which serves to view and pay an account balance. Since this package would need the user to log in in order to see his account, that is why it is included within the User Interface Subsystem. Lastly, the *Customer Reservation System* package allows a user to make a reservation for a spot in the parking garage. Given the fact that a user would need to have an account to reserve a spot as well as have their bill adjusted accordingly, this is why the package is included within the subsystem. Together, these 3 packages need to access other information that would be a part of another subsystem which is the Central Server

The Central Server subsystem contains 2 different packages. The first package which is *Customer Information* aims to utilize a MongoDB database in order to save customer information such as username and passwords, billing information, license plate numbers, and other info that the User Interface subsystem needs. The other package is *Garage Information* and it aims to contain or obtain information about the garage capacity and spot status from the elevator and manager subgroups.

The Hands Free subsystem only contains a package that allows the user to bypass using the user interface in order to make reservations for the parking garage. By using their voice, they can sign in to their account and make parking reservations which is the only functionality that the subsystem allows. It will depend on Customer Information from the Central Server so that it can verify a user's account and make their reservation based on the garage information.

Mapping Subsystems to Hardware

The hardware for the Central Server subsystem will mainly be MongoDB. MongoDB will allow us to store information and pull information whenever the user requests it through the User Interface. MongoDB is also very easy to get started and tinker with since it is lightweight and ideal for web applications.

Persistent Data Storage

The Customer Group will use MongoDB to store customer data including all the login information such as name, date of birth, license plate, email address and payment information. This information needs to be dynamically updated by the user at any time but also be protected so that only the user will be able to see their own information. Additionally, in order to check if reservation times are available the customer and manager group databases will need to communicate with the database/server so that we can show the information to the user so they can decide where they would like to park.

Network Protocol

For communication between client and server and for the structure of the web applications, our system uses Node.JS. Node allows users to load scripts inline with the web pages and redirect the user to different pages via routes based on requests they send to the server. Additionally, using node allows us to utilize various web applications such as code that can securely handle and encrypt passwords.

Global Control Flow

Execution Orderliness:

This system is procedure-driven and executes in a linear fashion. Customers will have to create an account, enter their name, payment information and license number. Then, they can create or adjust a reservation and pay for said reservation. Every customer will have to follow this procedure in this order. All walk-in customers would also have to create an account first and follow the procedure above.

Time Dependency:

There is time dependency in this system. When a customer chooses to make or adjust a reservation they will be doing so in real time. The system will record the customers reservation and block off a spot for the reserved time. Additionally, after the customer checks in to their parking spot, the number of minutes the customer has left of their reservation will also be tracked by the system.

Hardware Requirements

The Customer group only requires user interfaces and there are no hardware requirements.

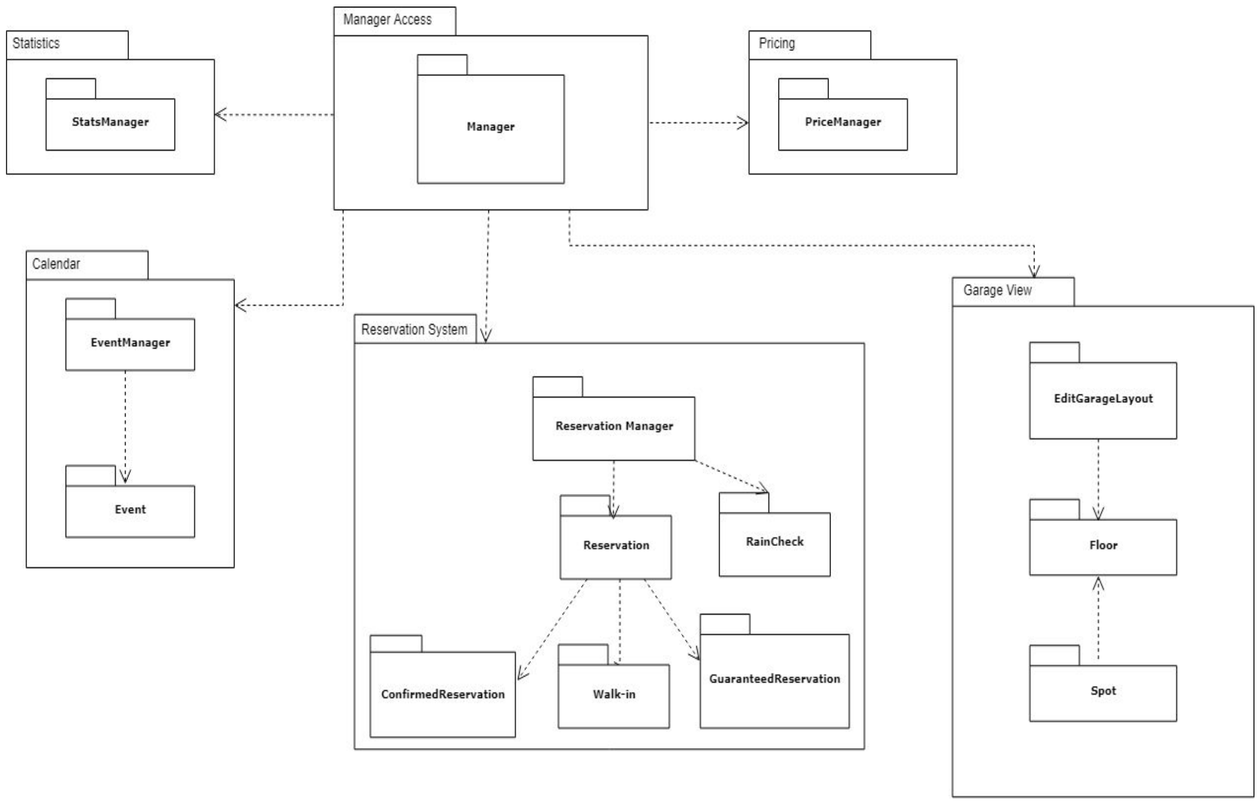
Managerial / Administrative Subgroup

Architectural Styles

The manager group will be creating a component-based web application as this is the architectural style best supported by the MERN stack. Each of the functions our application supports will be the responsibility of a dedicated component. For example, there is a component dedicated to displaying the current reservations in the garage.

The manager group will also be implementing a Model-View-Controller solution pattern. The user will mainly interact with the controller, which will interpret the user's actions such as button clicks. Then, the controller will edit or display the data as the user desires by editing the view or the data in the model. Crucially, the model represents only the data, and thus can be reused without modification. This will streamline our development process and allow us to easily support multiple functionalities for the same underlying data.

Identifying Subsystems



The Manager package diagram has five main subsystems with subpackages within each one. The Manager Access subsystem contains the *Manager* package which allows a parking garage manager access to data within the Central Server regarding all aspects of logistics. The viewStats() method within *Manager* relies on the *StatsManager* package located within the Statistics subsystem. This package manages and creates charts and graphs of all the data the parking garage has collected over a specified period of time. It is how Managers can track no-shows, overstays, reservations, and payment in general.

Manager also depends on the Calendar subsystem to keep track of upcoming and current events. The *EventManager* package accesses the *Event* package to create events in the calendar. The *EventManager* can also delete events and show them on the calendar.

Manager uses the Pricing subsystem to change the prices and fees for parking in the garage. The *PriceManager* package calculates and controls the prices of walk-ins and reservations. It does the same for actions that would cause fees such as overstays and no-shows.

The Garage View subsystem contains the controls and elements for the virtual garage layout. The *EditGarageLayout* package manages the number of floors and parking spots in the parking garage when the manager is editing the garage layout using the computer. It directly manages the *Floor* package in order to do this. The *Floor* package creates and removes parking spots on a specified floor when prompted by manager interaction with the render. The *Spot* package contains parking spot information and statuses. Since the spots need to be on a floor and will not exist otherwise, the two packages are paired together within Garage View. Also in this way, *EditGarageLayout* can access information from both *Floor* and *Spots* at once.

The Reservation System subsystem contains the tools for a manager to create,view, and manage reservations in the Central Server. The main package, *ReservationManager*, has the power to view, create, and manage reservations in the system, as well as issue rain checks for customers that had made a previous reservation, but could not park due to lack of spot availability in the garage. To create a rain check, this package accesses *RainCheck* which records all of the necessary information required to create a rain check such as a checkID, the time of issue, and the names of both the manager and customer involved. To create a reservation, *ReservationManager* calls the *Reservation* package to organize and send the relevant information to it. *Reservation* then imports the correct package based on the type of reservation that is being dealt with: *Walk-in, GuaranteedReservation,* or *ConfirmedReservation*.

The *GuaranteedReservation* package has access to monthly contracts, the reservation time, and the rate of charge for the customer. The *ConfirmedReservation* package would contain the times for the reservation and the confirmation for the reservation. The *Walk-in* package contains the reservation times for walk-in customer reservations.

Mapping Subsystems to Hardware

Currently, the system is hosted locally, so we can run our client application and our server on the same computer. In the future, we hope to obtain a website hosted on Rutgers servers at the url smartpark.rutgers.edu. At that point, our website will be hosted by Rutgers servers and can be accessed via any computer.

Persistent Data Storage

The MongoDB database will hold all data relevant to the Manager group. It is where all of the subsystems that the Manager package has to access are located. The Reservation System, Garage View, Statistics, Pricing, and Event subsystems are all implemented in the database. MongoDB uses collections to organize data; unlike SQL databases, MongoDB does not enforce a strict schema. Below is a list of collections the manager application will be able to access:

Customer-info: Contains customer data including name, phone number, and email address
Reservations: Includes current and past reservations

Pricing: Current prices for hourly and overtime parking
Events: Information about upcoming events, includes fields like title, description, etc.
Spots: Information about the number of spots available, status of the spot, etc.

Network Protocol

We are using the HTTP network protocol. We need a protocol that will send requests individually between client, proxies, and server. This is because our SmartPark website will be responding to numerous individual HTTP requests when managers type the URL into a web browser to login to their account and look at specific web pages within the Manager Portal. It is also useful for fetching images, media, and HTML files to the server and updating the website quickly with new information. HTTP can also protect important web pages with an extra layer of authentication, giving the website the ability to handle sensitive user information while also remembering the state of the server. We plan to use HTTPS for extra security on the manager subsystem after we deploy on the web server.

Global Control Flow

Execution Order:

The Manager application will operate in an event-driven manner. The user can choose to perform (or not perform) any of the actions that are available to them such as change prices, edit garage layout, or view existing reservations.

Time Dependency:

The manager system will not have any timers. However, in a real-world situation such as a sudden influx of customers, the view reservation page would have to be dynamically updated to ensure data validity.

Concurrency:

The manager system does not use multithreading. However, the simulation of the parking garage may use multiple threads.

Hardware Requirements

1. **<u>Screen Display</u>** - React.js for a useable Manager Interface
2. **<u>Central Server</u>** - At least 100 MB of space on MongoDB for reservation and rain check data storage, on top of the virtual garage layout

Elevator Operation Subgroup

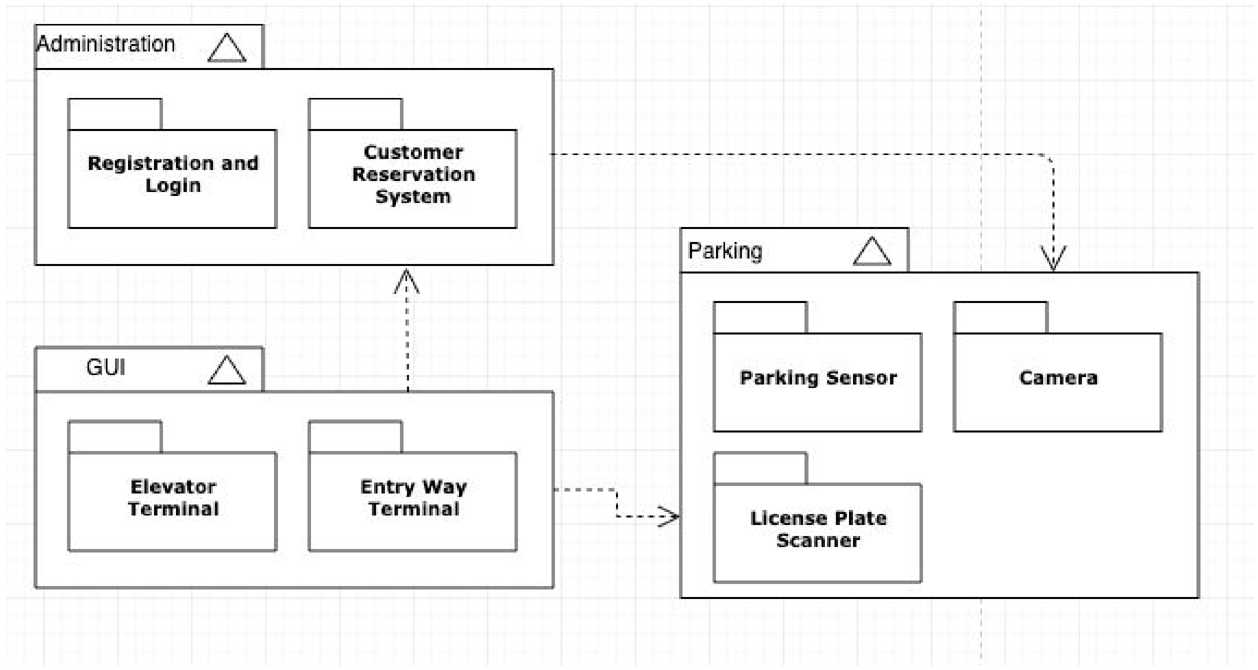
Architectural Styles

The elevator subgroup employs various architectural styles depending on the subsystem of the elevator. For the elevator terminal that the user interacts with, an *event-driven architecture* is utilized. Each user input into the terminal can be viewed as an event, as it leads to a change of state that the elevator terminal follows. For instance, if the license plate scanner was able to successfully scan a license plate, then this state is passed on to the customer reservation system to identify if the customer has a reservation. If the license plate could not be scanned, then control is passed on to a different state- prompting the user for a membership number to determine if a reservation can be found. The elevator terminal consists primarily of branching paths depending on what the user does or enters, and the state of this event is passed on to another state to control the system.

Apart from the event driven model, the elevator subgroup also employs the *client server model* because there needs to be communication between the database with all the customer information and the elevator interface. We are allowing the customers that reserve a spot in advance to be able to select the general area in which they want to be parked and to do that we need to ensure that they customer is not a walk in, that is that their information exists in the database, check for an empty spot in the region that they want to park (also from the database) and display the closest empty spot in the desired region on the elevator interface. In this case, the client would be able to interact through the elevator system and the server would be the database that would respond to the client's requirements.

The architecture will also employ the *data centered model*, since the client will have access to data from a central server. The exchange of data between the client and the server is mutual; the server can provide data to the client, and the client can modify data in the server. One such example of a client modifying data is in the event that they decide to register as a user or change information about their account. All clients, whether they are registered or not, have the same amount of access to the data. This will make the server integrable and the data will always remain centralized. The data inside of the server will never branch off, even though there are many different scenarios for what could happen. Each scenario, while different, will always call back to the data that is centralized.

Identifying Subsystems



The package diagram depicts subsystems of the elevator functionality grouped into logical subcategories and connected via dependencies.

The GUI subsystem interfaces with the customer and contains the logical grouping of the two distinct GUI elements; Elevator Terminal and Entry Way Terminal. Its functionality further depends on the Administration package and Parking package to be able to convey useful information to the customer.

The Administration subsystem encapsulates the Registration and Login class as well as the Customer Reservation System. This logical packaging of classes reflects the logical commonalities between the two classes. Both deal with the business logic of the customer interaction and dictate the states displayed by the GUI subsystem based on their interactions with database elements and the real-time dependency on the Parking package.

Finally, the parking package encapsulates the Parking Sensor, Camera and License Plate Scanner class. These three sensors comprise the “eyes” of the system and report to the database, and thereby the administrative classes the real time activity in the parking garage.

Mapping Subsystems to Hardware

The elevator terminal can be extended to include all of the user input devices contained in the garage facility. This would not only encompass the elevator terminal’s interface but also the input devices for walk-in customers as well. Each of these separate walk-in terminals will require its own hardware, meaning that for every walk-in terminal introduced a new computer will be necessary to run the interface for user interaction. These devices would each run the walk-in customer subsystem, whereas the main elevator terminal will control the flow of events for customers in the elevator. Each of these terminals, both walk-in and the main elevator, will be able to assign rain-checks to the customer in the event that the garage is filled.

In a real world deployment of this system there would need to be a minimum of two cameras; one placed at the entrance and one at the exit. For reliability the interface would be wired. Camera data would feed directly to the associated class modules for processing in real time.

Finally there would need to be a physical parking spot sensor to determine the occupied/unoccupied state of each parking spot. This would also be hardwired into the administrative system.

Persistent Data Storage

The elevator terminal needs to access a database to check for already reserved spots, valid customer reservations and spot vacancies that can be filled. This information will need to be kept for a long period of time. It can not simply delete this information once the user is done using the service. There will be a relational database between all of the groups, that can be accessed by the manager, the customer, and the elevator. All of the essential information for the project will be available in this relational database; it is the foundation of this project.

The elevator terminal will need access to the database which stores information such as garage occupancy status, reservations, and access to customer login information. Each customer interaction must be able to obtain information from the database to determine the flow of events the customer will experience in the elevator terminal. The customer interaction may also go in the other direction, where they modify data such as registering as a new customer or changing information about their account. We will use a remote hosted instance of MongoDB to handle database needs for the system. We will integrate MongoDB with React.js in order to appropriately link the front end and back end applications. The React.js code will be set up to update itself in accordance with any changes that happen in the MongoDB database. React.js can update in real time and will not require the user to have to reload the interface. The updates to the front end will be fast and in real time. It is important that we integrate our MongoDB with React.js as precisely as we can.

Network Protocol

To accommodate the flow of data between the elevator system (on site) and the hosted backend our system would rely on a variety of HTTP requests. The systems from the parking package will update the database to reflect real time in/out flow of cars. This is to create a network communication chain of events in which the entryway terminal prompts the customer in a manner appropriate to their reservation status.

Since we would be using a combination of a data centered model and the client server model as architectural styles in this subgroup, we should be employing an extension of the TCP/IP model to implement the socket communication structure between the information that the client enters into the system, to the elevator system communicating with the database that has past information.

When the client enters the elevator with the relevant number plate and proper reservation, the elevator system has to make a connection with the server that is connected to the database with the repository of relevant information for reservation. To implement this, we need to form a socket connection between the elevator terminal and the server system which is connected to the database for proper flow of information. Since a simple client server model is something that our entire group is familiar with, in terms of understanding and implementing, we are considering using this for the socket connections that need to be made as a part of the project.

Global Control Flow

Being that there are three different groups, the control flow may differ from group to group. The elevator group will have an event based control flow.

Hardware Requirements

1. **Server** - To process the information in the database and convey it to the server terminal
2. **Database** - At least 1TB to store all the relevant data for the customer reservations
3. **Cameras** - 1080p Wireless Security Cameras to ensure security of the garage
4. **License Plate Reader** - 25 GB minimum to store the information in the reader as a backup
5. **Touchscreen-capable display terminal** - Visual Interface System as User Interface
6. **Parking Sensors** - Wireless Garmin 40 seems to be a good parking sensor which would serve our purposes
7. **Display Screen** - to show the vacant spots of the garage outside of the garage

Algorithms and Data Structures

Algorithms

Customer Registration Subgroup

No algorithms were needed for the customer subgroup. No mathematical models were used since our group only needs to make user accounts, show billing and other relevant data, and create a secure way for the customer to make a reservation.

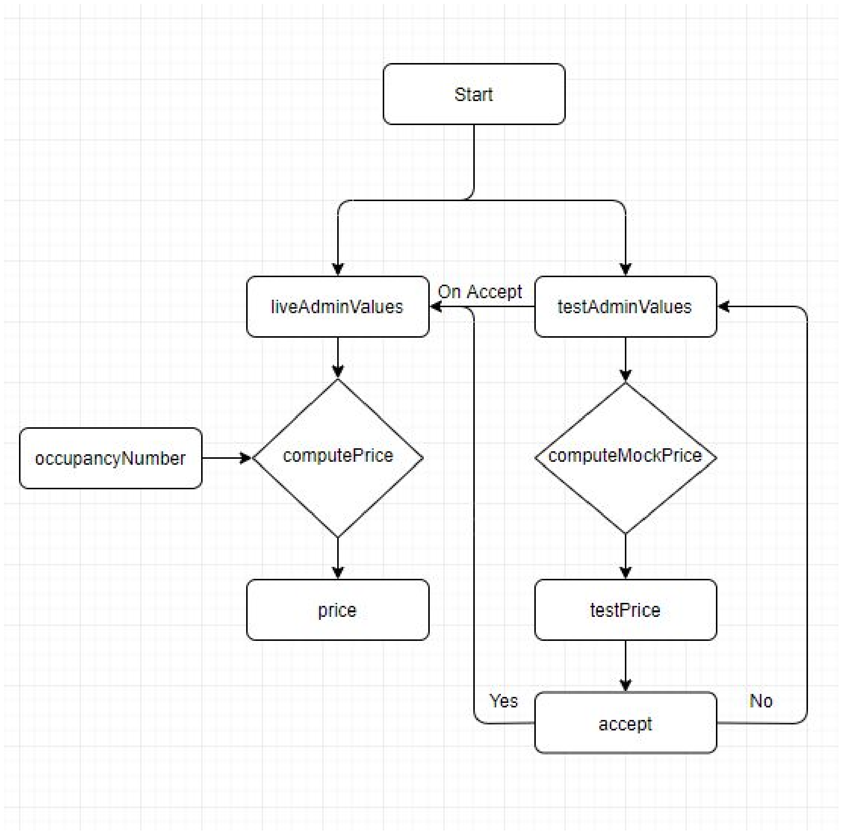
Managerial / Administrator Subgroup

Dynamic Pricing Algorithm

Dynamic pricing will require the use of a custom algorithm. This algorithm performs its computation phase with a formula derived from research referenced earlier in the report under the heading "Mathematical Models." The expression is represented again for reference:

$$base * hrsParked * [1 + \left[(currOccPercent - minThresh) * \frac{1 - baseMult}{maxThresh - minThresh}\right]]$$

In practice, we will need to develop a logical flow by which the administrator of the system can enter relevant data and receive a revenue projection from the model. The administrator can then choose to implement the price model. When a model is live it takes in actual parking occupancy data and uses the administrator-controlled values to compute a price for the customer.



One can see that the algorithm will have two state options from the beginning. The admin can opt to see the price output data from the currently set model, or can enter new administrator set values and compute recompute the model. If the admin decides to implement the new price then the algorithm recomputes a live price with the test values.

Elevator Operation Subgroup

Poisson Random Processes

The poisson random process is used in the simulation of the parking garage arrivals, departures, and overbooking. To restate from report 1, the poisson random process is modeled by the probability of seeing n arrivals in the time interval Δt:

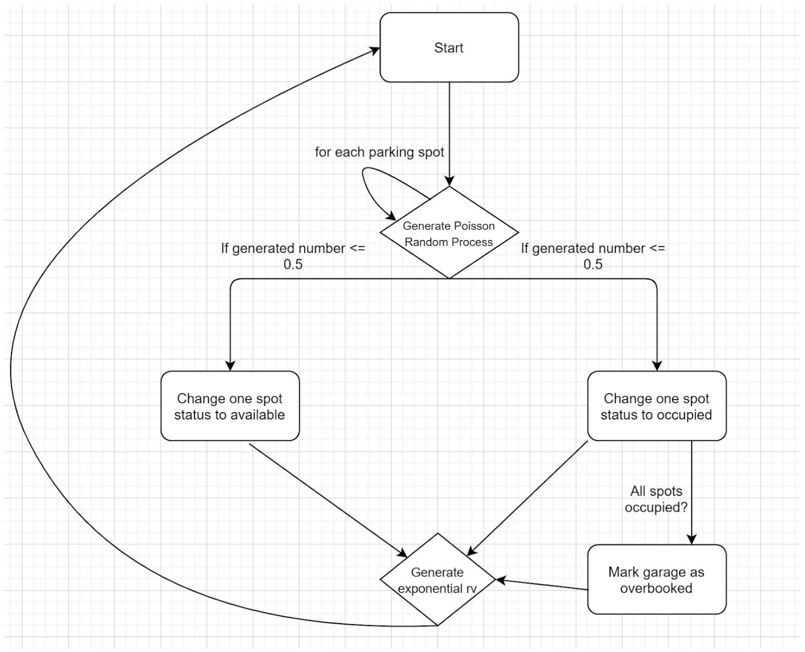
$Pr(n) = \frac{e^{-\lambda\Delta t}(\lambda\cdot\Delta t)^n}{n!}$ and $E\{n\} = \lambda\cdot\Delta t$

Where the same model is used to simulate the departures. Then to simulate the time between arrivals, an exponential random number is generated where $rx(u) = \frac{-ln(u)}{\lambda}$ - u represents one hour of time and lambda is the average number of arrivals per hour.

The logic flow for the simulation of the random process is described below:

1. Change the status of one spot from “Available” to “Occupied.” If there are no available spots, mark as an “Overbooked” event.
2. Generate a random value rx as defined above. After a time t(rx) = 60rx minutes, perform step 1 again.

This is displayed in the following activity diagram:


Start
for each parking spot
Generate Poisson Random Process
If generated number <= 0.5
If generated number <= 0.5
Change one spot status to available
Change one spot status to occupied
All spots occupied?
Generate exponential rv
Mark garage as overbooked

Data Structures

Customer Registration Subgroup

The customer group needed a way to be able to store the information of an account so that the user can refer to it when making a reservation, looking up their bill, and editing their account. In order to do this, we used a MongoDB database that stored information in “documents”. In those “documents”, the information is stored in a JSON (Javascript Object Notation) format as follows:

Firstname : String
Lastname : String
Email: String
Username: String
Password: String

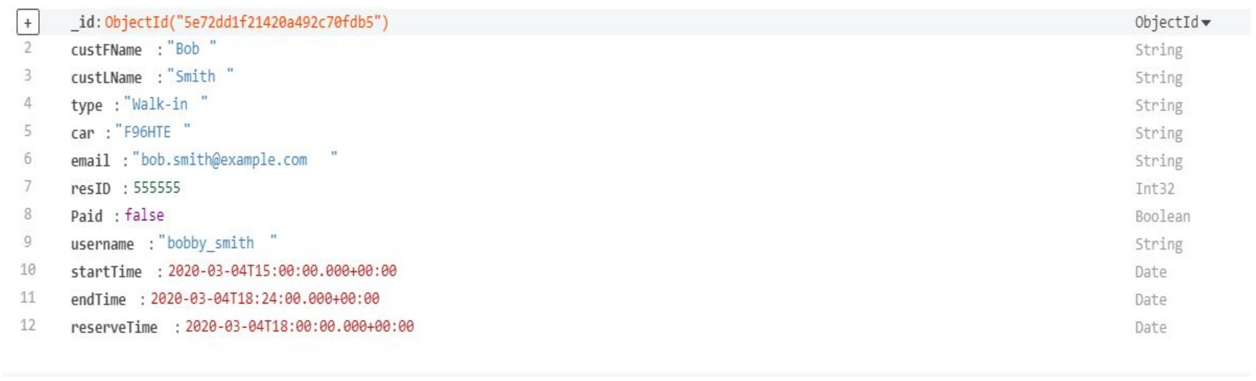
This format of data collection is basically a key,value pair which allows us to pull data from different documents based on a certain unique key that is associated with the user which in this case would be their email. We believe that this will be the most beneficial way to ensure fast data retrieval as well as data logging. We have also included other non required fields in a user’s account such as their credit card number and other associated info in case they would like to have a card associated with their account when paying their bill.

Another reason we like this approach is the fact that having a Schema like the one supported in MongoDB allows us to create embedded data for a customer’s account to make it even more personal. For example, instead of having an “email” field, we could have a “contact” field and within that field we could store an email and a phone number. This gives more flexibility to the user without sacrificing the structure of other models within the backend!

Managerial / Administrator Subgroup

The Managers need to be able to create parking reservations and delete or alter existing ones. In order to do that, we needed to store the reservation information somewhere. We wanted a backend that has a database that is easily searchable, even when looking for specific information, and is flexible such that relevant information can be found through the use of any number of unique identifiers. Also, it is flexible in the sense that managers can easily create, alter, and/or delete reservations stored inside the database. The backend also needed to seamlessly integrate with the frontend of the website, to ensure minimal user effort. Thus, we decided to use MongoDB as our backend since we are already using other members of the MERN stack (Node and React) for our frontend. MongoDB Compass to create a SmartPark database. Within SmartPark, we created a data collection called “Reservations”. The collection stores reservation information as .JSON files in the Documents tab. We have created a Schema in our file res.js, so that any reservation information that is inputted through the Reservation page on the SmartPark website gets stored directly into the MongoDB Reservations database as a Document. We set up the Schema so that the reservation data is matched to certain object fields. Each field in the document is formatted as “key: value” with the “value” being of data type

String, Int32, Boolean, etc. Each document is also automatically assigned an ObjectID by MongoDB. For example:


```
_id: ObjectId("5e72dd1f21420a492c70fdb5")        ObjectId
custFName  : "Bob  "                             String
custLName  : "Smith  "                           String
type  : "Walk-in  "                              String
car  : "F96HTE  "                                String
email  : "bob.smith@example.com   "              String
resID  : 555555                                  Int32
Paid  : false                                    Boolean
username  : "bobby_smith  "                      String
startTime  : 2020-03-04T15:00:00.000+00:00       Date
endTime  : 2020-03-04T18:24:00.000+00:00         Date
reserveTime  : 2020-03-04T18:00:00.000+00:00     Date
```

This type of formatting allows us to use the Schema tab in MongoDB to analyze the current data collection. Analyzing the schema allows our group to observe the frequency, types and ranges of the fields in Reservations, which is useful data for our parking garage Statistics page and its corresponding graphs and tables. The "key: value" format also provides managers with easy access when looking for specific information in each document. For example, normally a manager would look for a customer's resID when searching for their reservation in the system. However, if a resID has not been issued (because a reservation was never made), a customer can usually also be identified by their vehicle's license plate number or by the email they provide. Given a unique identifier, a manager is given the flexibility to find other data objects such as the customer's SmartPark account username or their reservation time. This system also allows for easy sorting of the reservation data, i.e. sorting by date of reservation.

Elevator Operation Subgroup

The elevator modules need to temporarily store data when fetching it from the database, and prior to saving to the database. The volume of data moving at any given time is small. An example data package would comprise an object with the following fields

Name: "CustomerName"
Reservation Status: resStatus
Reservation Details: resDetailsObject

The Elevator Operation Subgroup determined that such a small quantity of data can most efficiently be stored in a JavaScript array. The ease of using a built-in datatype far surpases any performace gained derived from a faster, but potentially more complex solution. The guiding principle in choosing the built-in JavaScript array type was simplicity and reliability, not speed.

As with the Customer and Management groups, the Elevator must also have access to MongoDB, however the elevator will only receive and view these values and decide what to do based on the results of these values. The elevator terminal will not add or delete any values, only update values. The ground-floor walk-in terminals, however, will be based on the customer registration system and will therefore need to add and update customer login information.

User Interface Design and Implementation

Customer Registrations Group

The Customer Group's user interfaces are now going to be coded separately from each other. We have divided the main components into 3 manageable parts: Login and Sign Up + Main Account Page, Editing Account Information + Billing & Making a Reservation.

Login and Sign Up

The layout of this page has also not changed significantly but enough to make it more user friendly, we have eliminated the separate page to choose whether they are a new or returning customer but made them both into 1 main page, that has divisions for both login and sign up on either side.

Main Account

The layout of this page has changed from 3 buttons to 4 buttons, namely: Making a new Reservation, Editing an Existing Reservation, Billing and Editing Account information. The last two have been separated (since earlier) to make finding them easier from the main page. It also became aware to us that editing your own account information would be a much more important feature that should be given as an option on the main page, than previously thought.

Making a Reservation

This page was not initially included in the draw-ups from the previous report, the user will be asked to pick a duration of time to park; day, week, month, etc. After this a calendar will appear to choose the dates to park on, this will have the previously chosen restriction. Then, the user is asked to confirm the dates and the system checks for their availability. If the availability is found then the user is asked to choose times between which they can park.

Editing Account Information

This page brings the user to all the current demographic information that the central server has on that particular user including their first and last name, email address, password, etc. The user can then edit this information by typing in the corresponding boxes and click a confirm button to push all of the updated information in the central server, this also ensures that this information is saved and the user and the system both acknowledge a change in the information held by both parties.

Billing

The billing page displays all the interactions between the charges and refunds on the customer's account according to the previously stated business policies.

Managerial / Administrator Subgroup

Dynamic Pricing Page

The user interface for the dynamic pricing model page if intended to be simple and intuitive for the user. The admin may select from 3 options: View/Edit current price model, Create new price model and View historic results.

View/Edit Current Price Model

The View/Edit current price model page allows the admin to view a graph of the projected revenue vs. the revenue generated from the base hourly parking rate. On this same page the admin can alter the base rate, base rate multiplier, and the two parking occupancy percentage modifiers. The admin cannot alter the parking percentage weights on this page. This is intentional. The parking percentage weights are an integral part of the parking price model and should only be changed after a lot of consideration, and after accruing quite a bit of parking data to base the decision off of. Therefore, the admin is limited to the entry of variables that have a more short-term impact on the revenue.

The View/Edit page costs one click to access, there are three fields that offer tab-between access and a single submit button to make the new values live and refresh the graph. This creates a streamlined process and maximises “ease-of-use”.

Create New Price Model

The Create New Price Model page is where the admin will have access to altering the “percentage weights” of the price model. These should only be changed after months worth of data have been collected to see how actual parking occupancy rates compare to the theoretical occupancy percentages derived from the Uber research data of parking rates in urban zones. The admin should track the occupancy rates over a period of months, average it and enter the accurate occupancy percentages by hour into the appropriate fields.

The page can be accessed with a single click, the fields are tab-accessible and a single button makes the new weights live and updates the graph. This is an easy-to-use page, but the knowledge of when and in what way to make these changes to the price model would require training outside the scope of the software itself.

View Historic Performance

The View Historic Performance interface allows the admin to access the historic actual revenue for a given day and graphs it in relation to the projected revenue for the admin’s price model on that given day. This provides the admin with essential information regarding the accuracy of the price model and gives perspective on what may, or may not need to be changed.

The page requires only a single button to access, and then a simple date selection tool is offered for data selection. The date selector is modeled after the common date picker that is ubiquitous across the internet.

The need for only a single button to access the desired data makes this a very simple and intuitive feature for the admin to access.

Admin Login

This page allows managers to login to access the administrative features. This will verify their credentials and navigate them to the managers home page where they can view and interact with the different features for managers. The Admin Login link redirects to an Auth0 page that then prompts the user to enter their email and password. If the login credentials can be verified, the user is then redirected to the home page which includes tabs for each administrative feature.

Reservation Page

This page will allow the manager to configure different aspects of the reservation process. Managers can confirm details like reservation price, cancellation fee, enabling rain check etc. As an extension of the login page, the managers are the only ones who can change these details, and then those changes are then passed down for the customers to adopt. Many of these options will be configured with a text box that explains its corresponding parameter attached to a place where the managers can edit these details and to publish these changes. Additionally to editing, managers can also override any reservations made by the customers and cancel reservations as they seem fit.

Pricing Page

The pricing page will allow the manager to configure pricing either dynamically through the dynamic pricing model or with fixed inputs. A toggle switch will be added; when the setting is set to manual, a pricing menu similar to the one shown in the mockups of report 1 will be displayed. Otherwise, the manager will configure pricing through the dynamic page described above.

Statistics Page

The layout of the page looks the same as the mockups. We used react-bootstrap for the buttons and input group, and react-chartjs for the graph. The user can click a button to choose over what time interval they want to view the statistics for (Daily, Weekly, Monthly, by Year). They can click the year they want to view from the by Year dropdown menu. Otherwise, they have to input the exact time interval they want to view into the search bar next to the buttons. By showing statistics for only the specified time interval, we can keep the page from being cluttered and the graphs from being too difficult to read due to numerous data points and microscopic scale sizes. The organization, although requiring an extra step, improves the user experience greatly--enough to offset the user effort. The graph is also more interactive and convenient than we had originally planned it to be. The user can hover their cursor over the individual bars in the bar graph, and a bubble with all the information the bar represents appears. This decreases user effort as they can easily read and access the information the graphs are providing.

Garage Configuration Page

The layout of this page is quite simple, but its functionality is what is most crucial. The managers will be asked how many floors does their garage have and how many parking spots does each floor have. After that the page will render a table format showing the user how the spots are numbered and where the road will be as well. This data will be how managers can configure their garage for customers to view and reserve.

Elevator Operation Group

Elevator Terminal

The elevator group's user interface went through very minor changes overall. As specified in report 1, the elevator group was initially going to allow for the user to register for a SmartPark account through the elevator terminal. This functionality is instead moved to the walk-in terminals, which will share most of the design of the customer interface when visited on the SmartPark website. Instead, when the user's license plate cannot be scanned the user is prompted with two push buttons, one which asks if the user would like to register as a walk-in customer - and thereby must exit the elevator to obtain a reservation through the walk-in terminal - and the other which prompts the user for a reservation number. This takes away some additional inputs necessary in the elevator terminal: simplifying one of the keyboard interactions with a push button interaction.

Moreover, an additional push button will be added to each interaction to confirm a user's inputs. This will add a quality of life improvement to the elevator terminal in the event that the user accidentally enters incorrect information.

Lastly, instead of prompting the user for his or her username and password, the user will instead be prompted for his or her membership number when the license plate could not be scanned. The membership number is a unique identifier given to each customer of SmartPark. This will simplify the keyboard inputs, as instead of using a full keypad a numpad will be used instead. The user will now have less to enter depending on the lengths of his or her username and password, as well as less room for error due to less values to input.

The elevator group is striving to make user interaction with the customer as simple and efficient as possible. By limiting customer inputs to only push buttons and a numpad while the elevator's cameras and license plate scanners do all of the other work, this will maximize customer ease of use. In report 1, the maximum number of user inputs was the case where the user's license plate could not be recognized and the user did not have a valid membership number and by extension SmartPark account. In the original design, the user would stay in the elevator and register for an account. After being discussed with the group, it was decided that in order to keep vehicles entering the elevator and parking efficiently, these customers should be asked to leave the elevator and register for an account either using his or her smart device or the walk-in terminals. This significantly cuts down the worst case user interaction in the elevator, now being the case where the user's license plate could not be recognized and the user has a valid membership number. In this case, only an interaction between a push button and entering the membership number on a numpad is necessary.

On the administrative side of the elevator terminal, no user interaction with the terminal will be necessary. The elevator terminal, when booted up for the first time each day, will perform a check in order to determine that the devices necessary for functionality are connected. If the devices are all correctly connected, the elevator terminal will proceed to the customer interaction interface. When the devices are not connected, an indicator will be displayed next to the listed device that is not currently connected. Until the device is successfully connected, the elevator terminal will remain on this screen.

The user interfaces are implemented by using a combination of React, CSS, and Bootstrap. React is used to compile all of the Javascript, CSS, and Bootstrap together into one package. CSS and Bootstrap are used in tandem to make the website appealing and eye-catching, while Javascript is used for all of the decision making necessary for the user interface to function properly.

Walk-In Terminal

The walk-in terminal design had little to no changes from its first iteration. As with the elevator terminal, the walk-in terminal will now receive a confirmation button for user inputs. While this will slightly increase the amount of user inputs required to complete a transaction on the walk-in terminal, in the event that the user inputs incorrect information more time will be saved in the long run. In the old scheme, if the user inputted incorrect information, they would have to complete the transaction with incorrect information and then begin a new inputting the correct information. In order to alleviate this lack of margin for error, the confirmation button will be added as a means for the user to catch any potential mistakes before advancing.

Like the elevator terminal, the walk-in terminal interface will be created using a combination of React, CSS, and Bootstrap.

Design of Tests

Unit Testing

Customer Registration Subgroup

Test-Case Identifier: **Use Case Tested:** **Pass/Fail Criteria:** **Input Data:**	TC-Register UC-1 The test passes if the data user enters saves to the MongodB cluster. Firstname, Lastname, Email, Username, Password
Test Procedure	**Expected Result**

Step 1:Input data into the fields to register for an account. Step 2: Check MongodB	When the user inputs the data, if the data is stored successfully when we check the data cluster in MongoDB all the user input data should be stored with a unique id. If all the data saves this means the user has created an account and can proceed to Login.

Test-Case Identifier: **Use Case Tested:** **Pass/Fail Criteria:** **Input Data:**	TC-Login UC- 2 The test passes if the user enters the correct username and password and is able to access their SmartPark account Username, Password
Test Procedure	**Expected Result**
Step 1: Create an account and enter the wrong username and password Step 2: Put in the correct username and password	The system creates a warning window stating which field has an incorrect data type entered into it. The system prompts the user to reenter the correct data. The user should be taken to their account where they can access their reservations, payment information and other account details.

Test-Case Identifier: **Use Case Tested:** **Pass/Fail Criteria:** **Input Data:**	TC-Make Reservation UC-4 The test passes if the user enters a date and time of their choosing and picks a spot to reserve.and receives a confirmation message that their reservation was successful as well as a unique identifier for their reservation. Date, Time, Spot
Test Procedure	**Expected Result**

Step 1: Pick a day, time and spot Step 2: Check for Confirmation	If the spot is unavailable then the system should prompt the user to to choose another spot. Once an available spot is chosen the System should send the user a confirmation message and a unique identifier for their reservation.

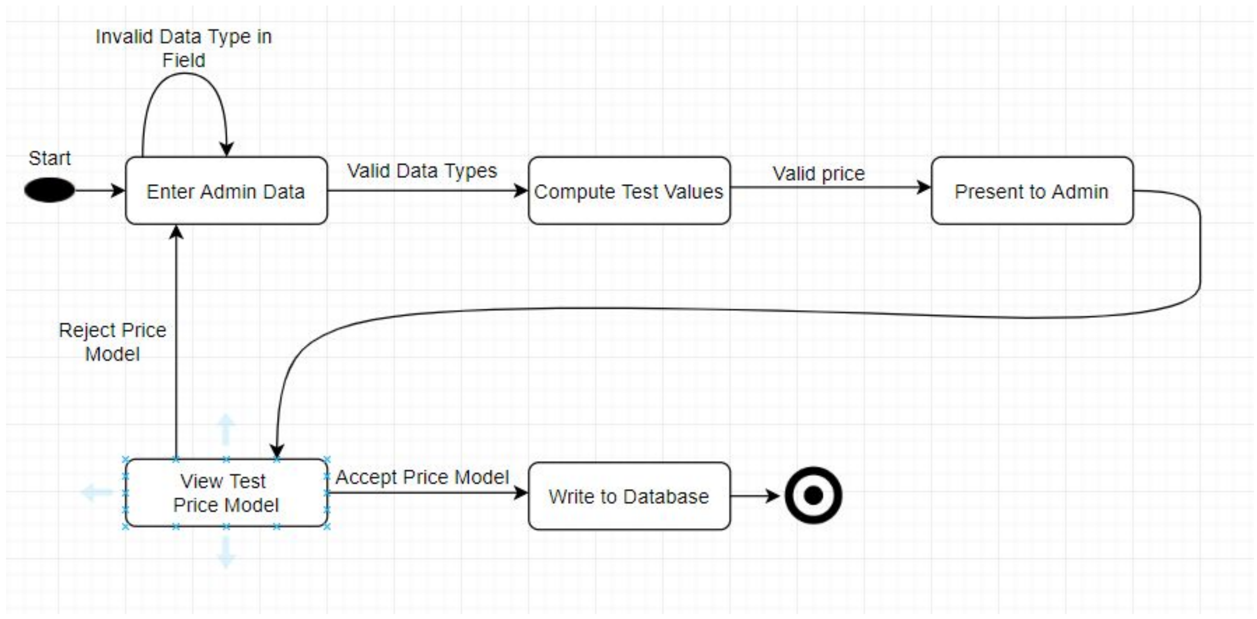
Test-Case Identifier: **Use Case Tested:** **Pass/Fail Criteria:** **Input Data:**	TC-Pay Bill UC- 7 The test passes if the user enters the correct credit card information and is shown a message that they paid their bill successfully. Credit Card Information
Test Procedure	**Expected Result**
Step 1: Enter incorrect credit card information and CVV Step 2: Put in the correct information	The system creates a warning window stating which field has an incorrect data type entered into it. The system prompts the user to reenter the correct data. If the user has entered the correct credit card information then they will receive a message that they paid their bill successfully and a receipt is sent to the email address they have on their account.

Managerial / Administrator Subgroup

Dynamic Pricing State Diagram and Test Tables

The dynamic pricing model will need to be tested for the following cases: incorrect input from the administrator, incorrect output data output from the mathematical price model and failure to update the database to the administrator's desired price model.

The input and computation state diagram:



From this state diagram we can derive the following test case tables:

Test-Case Identifier: **Use Case Tested:** **Pass/Fail Criteria:** **Input Data:**	TC-Dynamic Pricing UC-10 The test passes if the admin enters the correct data types in the available field and receives an accurate price model based on that data. Base Rate, Occupancy Percentage, Minimum Threshold, Maximum Threshold, Base Multiplier
Test Procedure	**Expected Result**
Step 1: Enter incorrect data types into the admin accessible fields. Step 2: Enter the correct data into	The system creates a warning window stating which field has an incorrect data type entered into it. The system prompts the admin to reenter the correct data. The system notifies the admin that a new model is being created with the provided values.

Set Pricing

Test-Case Identifier:	TC-Set Pricing-Front
Use Case Tested:	UC-10
Pass/Fail Criteria:	The test passes if the admin enters valid values for hourly, peak-time, and overstay rates. Boundary cases such as no values entered for a specific input will also be considered.
Input Data:	hourlyRate, walkInRate, cancelFee, overstayRate, guaranteedPrice
Test Procedure	**Expected Result**
Case 1: Administrator enters values for the pricing and presses submit Case 2: Administrator enters an invalid number	Case 1: The user interface indicates that the changes have been accepted. Case 2: System indicates that the value is invalid

Test-Case Identifier:	TC-Set Pricing-Back
Use Case Tested:	UC-10
Pass/Fail Criteria:	The test passes if valid values for hourly, peak-time, and overstay rates are successfully entered into the database.
Input Data:	hourlyRate, walkInRate, cancelFee, overstayRate, guaranteedPrice
Test Procedure	**Expected Result**
Case 1: Values for hourly, walk-in, cancellation, overstay, and guaranteed pricing are sent to the database via a post request	Case 1: The database is updated with the new prices. This change is reflected on MongoDB Compass.

Edit Garage Layout

Test-Case Identifier: **Use Case Tested:** **Pass/Fail Criteria:** **Input Data:**	TC-Edit Layout-Front UC-9 The test passes if the admin enters valid number of spots per floor and number of floors values and the system is able to render a view of the garage spotsPerFloor, numFloors
Test Procedure	**Expected Result**
Case 1: Administrator enters a valid number of spots and number of floors Case 2: Administrator enters an invalid number	Case 1: The system renders the number of spots and number of floors entered by the administrator Case 2: System indicates that the value is invalid

Test-Case Identifier: **Use Case Tested:** **Pass/Fail Criteria:** **Input Data:**	TC-Edit Layout-Back UC-9 The test passes if the values entered by the administrator are entered into the garage database spotsPerFloor, numFloors
Test Procedure	**Expected Result**
Case 1: A valid number of spots and number of floors are sent to the database via a post request	Case 1: The database is updated with the new amount of floors and spots and is reflected on MongoDB Compass

View Garage

Test-Case Identifier:	TC-View Garage-Front
Use Case Tested:	UC-11
Pass/Fail Criteria:	The test passes if the reservation page can successfully display the number of available spots and current reservations given fixed data for output (no database)
Input Data:	Available Spots, Current reservations
Test Procedure	**Expected Result**
Case 1: Administrator requests to view available spots	Case 1: The system displays all available spots on the user interface based on fixed input

Test-Case Identifier:	TC-View Garage-Back
Use Case Tested:	UC-11
Pass/Fail Criteria:	The test passes if the system can successfully retrieve records from the database given fixed search criteria (no user input)
Input Data:	Available Spots, Current reservations
Test Procedure	**Expected Result**
Case 1: A fixed search query is entered (for example, all occupied spots)	Case 1: The system responds with all entries that match the fixed search criteria

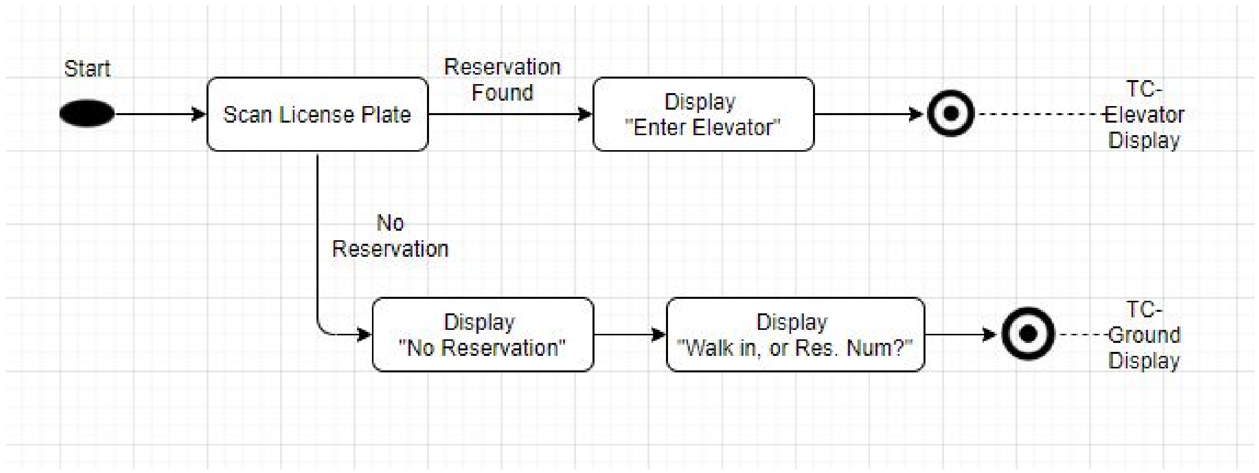
View Garage Statistics

Test-Case Identifier:	TC-View Garage Stats-Front
Use Case Tested:	UC-14
Pass/Fail Criteria:	The test passes if the reservation page can successfully display statistics in a graph given fixed inputs and search criteria
Input Data:	Date Range
Test Procedure	**Expected Result**
Case 1: Administrator requests to garage statistics for a certain date range	Case 1: The system displays the graph from fixed data given a certain date range (no database)

Test-Case Identifier:	TC-View Garage-Back
Use Case Tested:	UC-14
Pass/Fail Criteria:	The test passes if the system can successfully retrieve records from the database given fixed search criteria (no user input)
Input Data:	Date Range
Test Procedure	**Expected Result**
Case 1: A fixed search date range is entered	Case 1: The system responds with all entries from the database that match the fixed search criteria

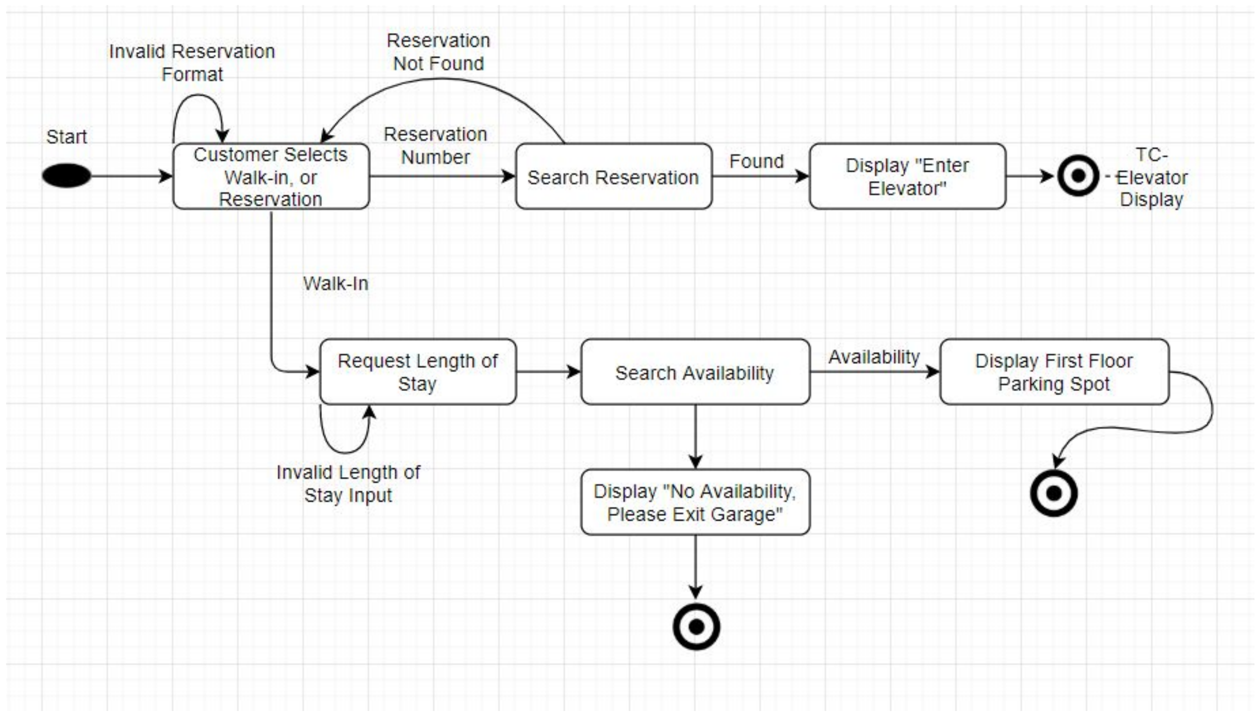
Elevator Operation Subgroup

Elevator Scan License Plate



Test-Case Identifier:	TC-Scan License Plate
Use Case Tested:	UC-15
Pass/Fail Criteria:	Pass Criteria 1: The test passes if a scanned license plate is associated with a reservation in the database and the customer is prompted to enter the elevator. AND Pass Criteria 2: The test passes if a scanned license plate cannot be associated with a reservation in the database and the customer is prompted appropriately by the elevator terminal.
Input Data:	License Plate Number
Test Procedure	**Expected Result**
Step 1: Enter a license plate number known to have a reservation on file.	The system prompts the user via the ground floor terminal to enter the elevator.
Step 2: Enter a license plate number known to not have a reservation on file.	The system prompts the user via the ground floor terminal to select whether they are a walk-in customer, or wish to enter a reservation number.

Ground Floor Terminal


Invalid Reservation Format
Reservation Not Found
Start
Customer Selects Walk-in, or Reservation
Reservation Number
Search Reservation
Found
Display "Enter Elevator"
TC-Elevator Display
Walk-In
Request Length of Stay
Search Availability
Availability
Display First Floor Parking Spot
Invalid Length of Stay Input
Display "No Availability, Please Exit Garage"

Test-Case Identifier: **Use Case Tested:** **Pass/Fail Criteria:** **Input Data:**	TC-Ground Floor Terminal, Reservation UC-17 Pass Criteria 1: The test passes if a reservation number that is entered is recognized, and the system prompts the customer to enter the elevator. AND Pass Criteria 2: The test passes if a correct reservation format is entered, but no matching reservation is found in system and the system informs the customer that no reservation is found. AND Pass Criteria 3: The test passes if an incorrect reservation number format is entered and the terminal notifies the customer of incorrect format and requests re-entry. Reservation Number
Test Procedure	**Expected Result**

Step 1: Select “Have Reservation”. Enter a reservation number known to be associated with a valid reservation.	System will recognize the reservation number and display which parking spot is assigned to the customer.
Step 2: Select “Have Reservation”. Enter an incorrectly formatted reservation number.	System should reject the input and prompt the customer to re-enter their reservation number.
Step 3: Select “Have Reservation”. Enter a correctly formatted reservation number that has no associated reservation.	System should inform the customer that no such reservation exists and ask if they would like to become a walk-in.

Test-Case Identifier:	TC-Ground Floor Terminal, Walk-In
Use Case Tested:	UC-17
Pass/Fail Criteria:	Pass Criteria 1: Customer selects walk-in on terminal. Terminal prompts for length of stay. If space is available, the terminal prompts customers where to park. If space is not available, the terminal prompts the customer to exit the parking garage. AND Pass Criteria 2: Customer selects walk-in on terminal. Terminal prompts for length of stay. If a customer inputs incorrect time format, the terminal notifies the customer and requests re-entry.
Input Data:	Walk-in, Length of Stay
Test Procedure	**Expected Result**

Step 1: Set the garage to available for the availability test, set it to unavailable for the rejection test.	No system interaction at Step 1.
Step 2: Select walk-in and enter length of stay in incorrect format	The system should reject the input and request reentry of length of stay.
Step 3: Set ground floor to available for length of stay.	No system interaction at Step 3.
Step 4: Select walk-on and enter length of stay in correct format.	System should inform customers that parking is available and direct them to their ground floor spot.
Step 5: Set ground floor to <u>not</u> available for length of stay	No system interaction at Step 5.
Step 6: Select walk-on and enter length of stay in correct format.	System should inform customers that parking is <u>not</u> available and request they exit the parking garage.

Test Coverage

Customer Registration Subgroup

Test coverage for the Registration, Login, Reservation and Bill Payment cases will be tested with both Specification testing and Code coverage. Code coverage level testing will be performed on the frontend and backend as it is being created. After the frontend and backend are prepared, the specification testing will ensure that all use cases needs are met and everything is functioning properly.

Managerial / Administrator Subgroup

Dynamic Pricing Test Coverage

Test coverage for the dynamic pricing model will be tested with both Specification testing and Code coverage. Code coverage level testing will be performed on each sub module as it is being created. After the subcomponents are prepared, the specification testing will ensure proper operation from the input space.

Test Coverage for Other Cases

The test coverage for other use cases will consist of both specification testing and code coverage. We will include all components of the front-end React user interface as well as back-end Express APIs and their interaction with the database. We will also perform peer review to make sure all specifications are met and all use cases are fulfilled. We can estimate that test cases we have described above will cover at least 80% code coverage since every component will be tested with boundary conditions. Since our component is intended to cover only managers interaction with the application, we will not be performing load testing.

Elevator Operation Subgroup

The test coverage of the frontend UI of the elevator terminal, the simulation, the database setup incorporating the spot administration and the dynamic pricing model will be done in collaboration with other groups. We will be peer reviewing to ensure correct specifications, ideas and methodologies. We will also be testing based on Specification and Code Coverage on all the different components to ensure there are no faults in implementation and no logical flaws in the code. We will be testing individual components and incorporating all logical scenarios into the code with error case testing to ensure errors are handled carefully and well, and the UI gracefully exits the screen.

Integration Testing Strategy

Overall Strategy

Integration testing across subgroups will begin after each subgroup's contribution has been reviewed and code coverage for that component is complete. We will be taking a bottom up strategy for integration. Since both the customer and manager group contributions depend on inputs from the Elevator Group, we will start by ensuring the Elevator components are performing as specified. This means first checking the database and back-end components to make sure the responses are correct and consistent. Next, we will test components from the manager and customer subgroups that interact directly with the elevator subsystem. For the customer subgroup, this will include the interface that determines if a customer can make a reservation based on spot availability. For the manager subgroup, the component that shows current spot status will be integrated. The manager event calendar does not depend on other components and does not have any dependents; thus it can be integrated at any time. The penultimate part that will be integrated and tested will be the reservation creation component of the customer group. The manager components depend on data from the customer group's reservations, and will thus be integrated into testing last.

Customer Registration Subgroup

The integration testing strategy for the customer group will begin by splitting the frontend and backend portion of the website. Through each step of developing the frontend and backend they will be tested for their respective user cases. Once both components are functional individually they will be combined through github. This strategy ensures minimal errors throughout the development of the website.

Managerial / Administrator Subgroup

The integration testing strategy for the manager group will first include unit tests for all Express APIs and React components. Once a unit test is complete, we will integrate that unit with its corresponding back-end or front-end partner. This can be described as a top-down approach. We are first testing the React components that have no dependent systems.

Elevator Operation Subgroup

The integration testing strategy for the elevator subsystems will begin by unit testing each component individually to ensure that the respective modules accept correct input and provide accurate output. This stage must be cleared before integration can commence.
Once discrete modules have been verified we will begin by sequentially adding modules to the logical chain of an overarching state diagram and testing system cohesion by creating various inputs that will result in known system states. The pass criteria for the system will be based on which system state should be active given a particular set of inputs. We will also add incorrect input to ensure that if an incorrect input is added, the chain of events will not move forward until something accurate is added. In any event of a test failure, whoever is responsible for the section that failed will look over the code and will make any necessary edits. If one member is

having trouble with a particular, other members of the group will also look into it. We will utilize a horizontal testing strategy, since all of our components rely on another component, and feedback from one component will be used in testing another. It will be bottom-up. It will follow this general idea:

1. Camera state: Check if license plate is registered in the system. If it is recognized in the system, the user should be taken to the entry way terminal and treated as a registered customer. If the license plate is not recognized in the system, the user will be taken to the entry way terminal and treated as a non registered user. If these events take place, the test will pass. If not, the test will be a failure
2. Entry way terminal:
    a. If the user is a registered customer, they will be prompted to the elevator and the test will pass. If they are not prompted to the elevator and they are a registered we will consider the test a failure
    b. The the user is a non-registered user, they will be directed to a screen asking them if they would like to be a walk in customer or if they have a reservation number and the test will pass. If they do not see this screen, the test will fail.
3. Post-entry way (User is not registered):
    a. If the user selects to enter a reservation number the system will check if their reservation number is in the system. If it is not in the system, they will be told they have no reservation and the test will pass. If they are in the system, they will be directed to the elevator and the test will pass. If their reservation is not in the system but the user is directed to the elevator, then the test will fail. If the user has a reservation but is not directed to the elevator, then the test will fail
    b. If the user selects to be treated as a walk-in customer, they will be asked for their intended length of stay. If they are asked this, the test passes. If not, the test fails. The system will then check if there is space available. If there is none available, the user will be prompted to leave and the test will be a success. If they are not, the test will fail. If there is space, the user will be directed to the elevator and the test will pass. If they are not directed to the elevator but there is space, the test will fail.

Our strategy will require us to create certain license plates and have certain scenarios in order to implement each test case listed above.

Project Management

1. Customer Registration Subgroup

Report 2 Part 1: This week the Customer Registration subgroup figured out the logistics in each of the use cases, with the connection to each other database and what information would be needed for each use case to function properly. This entailed communicating with the other groups about how they would pass on information to our functions and use cases for each operation. Along with this, we used draw.io to make the system sequence diagrams for each of the use cases. This allowed for better project management and allowed the team members to work individually while asking others to contribute and give opinions where necessary.

Along with this report, the subgroup also began to code the beginning interfaces to each of the pages of the SmartPark website (mobile-ready application) as per the Gantt map made in the Report 1 submission. We still have yet to figure out how to link the individual pages and only allow access to the following/preceding pages after logging in.

Report 2 Part 2: This week the Customer group met 2 times to discuss the combination of the skills in this project. We were able to successfully divide the coding portion of the project in preparation for the demo specifically into 3 parts consisting of: login/sign up interface, making a reservation and billing. Each member was allowed to pick that alongside what part they wished to work on this project. Our group was able to start this earlier on in the week and finish this report by progressively working on it throughout the week. After setting up the document and assigning roles to ourselves. The class diagrams were easy to draw up in the presence of the other group members which allowed for quick turnaround time and confirmation for any rising questions.

Report 2 Part 3: This week the Customer group met online 2 times this week due to the changing situations presented by the virus. We set up our backend with routes so that the current url would allow a user to see their account and edit information. We also discussed what was necessary for the front end implications of our part of the project and we coordinated with the managerial/administrative subgroup so that we didn't conflict with any of the information that they needed from the customer group. Implementing the MERN stack was integral to our learning of the requirements needed for the project and we hope to keep improving upon the models and code that we have so far.

Overall, this report was completed in a shorter amount of time due to the document being set up earlier in the week and working more frequently within the subgroup. We will focus on implementing this technique more in the coming weeks, maybe even working on the code in designated times due to the efficiency of the combined group.

2. Managerial / Administrative Subgroup

Report 2 Part 1: The manager subgroup has solidified the features that we wanted to incorporate and decided which features would take priority over others. We have decided to use the MERN stack to streamline our code and provide a reliable, modern codebase. Additionally, we have communicated with other groups and have decided what types of data will be used as arguments and returned back.

Along with this report we have begun coding a basic page to use as a framework for the rest of the pages that will each execute its own function. We still need to code the specifics that will differentiate each page by its function and make sure that these pages are integrated with each other under the manager page and that it's integrated with the customer and elevator teams products as well.

Report 2 Part 2: The manager subgroup has a strong understanding of what needs to be done and what we want to accomplish. During the next week we will be discussing what are the most pressing issues and objectives that need to get done and how to split that work up. These objectives will be built upon the foundation of our existing home page and the finished project will be integrated with the Elevator and Customer sub group's projects as well. We plan to use the break due to the coronavirus to our advantage by working together on the different objectives.

Report 2 Part 3: The manager group had multiple calls this week about our front end code and its implementation. Additionally we also worked on routing our pages together and connecting to the MongoDB database. The issue we were facing was trying to make some front end code to be as flexible as we want it while also coordinating with the other groups and how their code will interact with ours. To address these issues we invited members of the other groups to our calls to learn from how they have set up their code and databases and briefed them on everything that we have done. Therefore, there is more coherence and understanding as to where each group is in their progress.

Finishing this report ended with less stress than usual. That's probably due to this document being set up earlier and that everyone was committed to the project despite the busy midterm season.

3. Elevator Operation Subgroup

Report 2 Part 1: The elevator operation subgroup each worked on a part of the UML interaction diagrams as displayed above. This week, the elevator operation subgroup had a Discord call discussing plans for how the project would be broken up. It was decided that each member in the elevator subgroup would be responsible for a deliverable every two weeks before the first demonstration. If there are any questions regarding what needs to be added or suggestions from other members of the subgroup, the members of the group will not hesitate to reach out and contact their peers.

The elevator project was decided to be broken down into four sub-components for the first demonstration: a section consisting of user intractability, a section focusing on dynamic pricing and statistics, a section focusing on a basic simulation, and a section focusing on the backend of the elevator interface, specifically parking spot statuses. The subsystems may not be able to interact with one another by the time of the first demo; this intractability is planned for the second demo. Each of these subsections will cover the requirements first, and given enough time will be built up and developed into fully realized systems.

Report 2 Part 2: The Elevator group worked together by communicating through text and in person to discuss the possible architectural styles that we would need to employ. We also got in a call to discuss how we were going to implement the different sub project details that are required for this report. Each member was able to contribute to the decision making process and the collaboration of our teamwork is what we have in this section of our report. Everyone contributed equally and worked together even though each one of us had exams and other commitments.

We have already started to work on coding and implementing the project, we have divided the work into the dynamic pricing model, the simulation, the database and the front end UI for the elevator subsystem; each sub group member is working on one of these sub projects.

Primarily, this report was finished on a good timeline where there were not a lot of disgruntled team members even during midterm season. We split out work, finished it by the time of deadline and worked together to check the work that was updated all through the week.

Report 2 Part 3: The Elevator group worked through midterm season to get this report out of the way. We started working on the report beforehand so that we could use the spring break to work on majorly implementing the project. We primarily communicated through our project group Discord server to talk about the splitting of the project report and descriptions and also about how we were going to be implementing the project.

Little to no issues occurred in regards to merging the final copy of the report. Because the report was all done using Google Documents, each group was able to collaborate and merge in the format that the other project groups were using. This ensured that in the end, each section for each group shared the same format.

The discussion of implementation of the project was essential because we wanted to ensure uniformity of the different components that we were creating. The simulation and the front end UI have similar designs because they were made under the elevator terminal.

We also managed to implement the initial stages of the database in terms of spot administration. We used MongoDB to incorporate and implement the spot taken and not taken design in the parking garage.

The Dynamic Pricing Model has been implemented with the front end UI established with graphs and connections to the backend. We are using this as a business model as well to maximize profits as we are running the Parking Garage.

Project coordination and progress report, projected milestones and breakdown of responsibilities is further elaborated below.

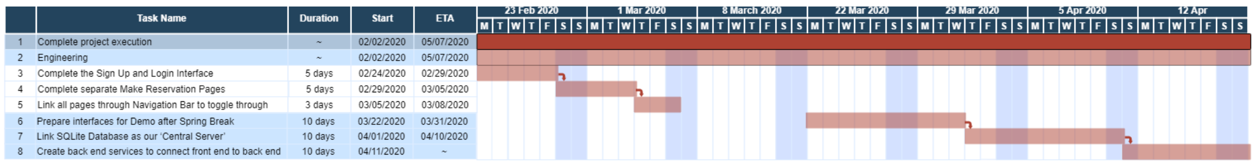
Plan of Work

1. Customer Registration Subgroup

Project Coordination and Breakdown Report

The Customer Subgroup has effectively distributed the work between ourselves along with breaking down our own subproblem into pieces that we could later integrate.

Projected Milestones



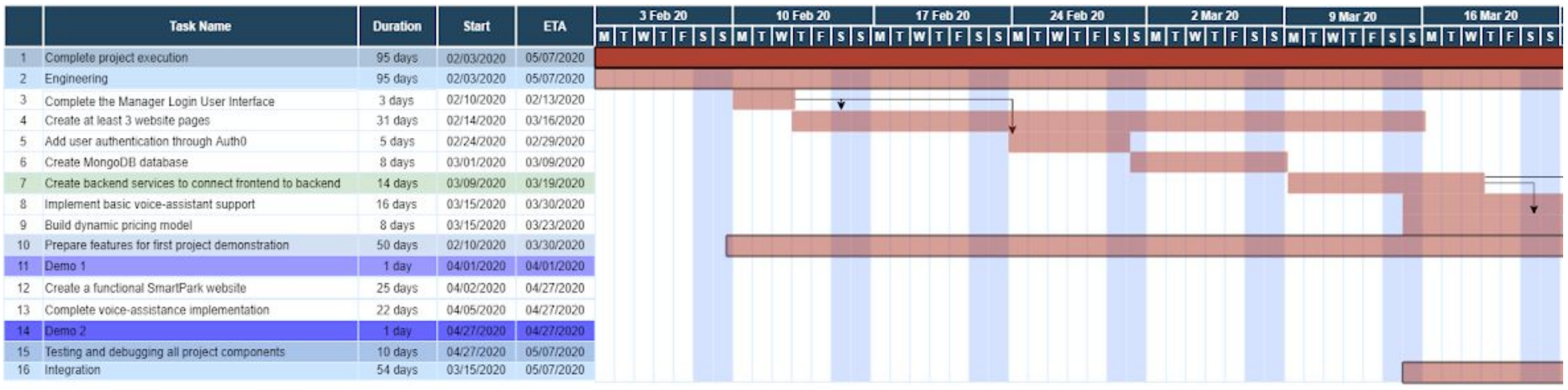
	Task Name	Duration	Start	ETA
1	Complete project execution	~	02/02/2020	05/07/2020
2	Engineering	~	02/02/2020	05/07/2020
3	Complete the Sign Up and Login Interface	5 days	02/24/2020	02/29/2020
4	Complete separate Make Reservation Pages	5 days	02/29/2020	03/05/2020
5	Link all pages through Navigation Bar to toggle through	3 days	03/05/2020	03/08/2020
6	Prepare interfaces for Demo after Spring Break	10 days	03/22/2020	03/31/2020
7	Link SQLite Database as our 'Central Server'	10 days	04/01/2020	04/10/2020
8	Create back end services to connect front end to back end	10 days	04/11/2020	~

Breakdown of Responsibilities

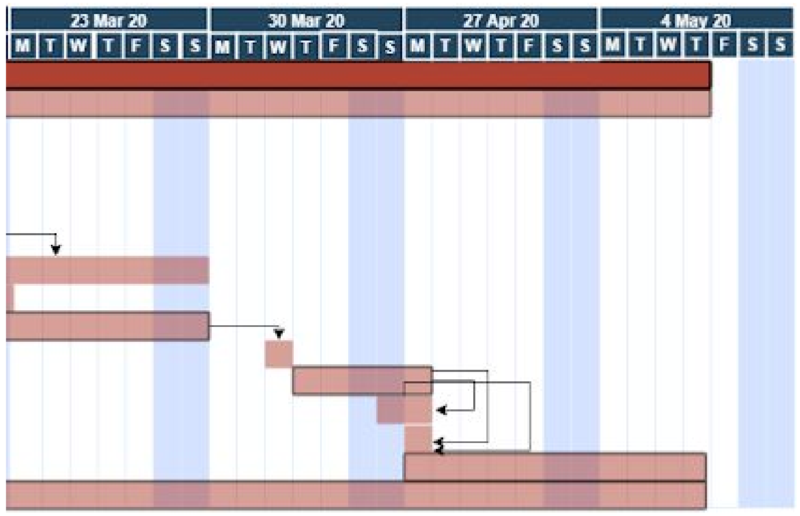
Name	Responsibility
Neha Nelson	- Sign up and Login + Main Account Page
Brian Ogbebor	- Billing + Editing Account Information
Param Patel	- Making + Editing a Reservation

2. Managerial / Administrative Subgroup

Projected Milestones



	Task Name	Duration	Start	ETA
1	Complete project execution	95 days	02/03/2020	05/07/2020
2	Engineering	95 days	02/03/2020	05/07/2020
3	Complete the Manager Login User Interface	3 days	02/10/2020	02/13/2020
4	Create at least 3 website pages	31 days	02/14/2020	03/16/2020
5	Add user authentication through Auth0	5 days	02/24/2020	02/29/2020
6	Create MongoDB database	8 days	03/01/2020	03/09/2020
7	Create backend services to connect frontend to backend	14 days	03/09/2020	03/19/2020
8	Implement basic voice-assistant support	16 days	03/15/2020	03/30/2020
9	Build dynamic pricing model	8 days	03/15/2020	03/23/2020
10	Prepare features for first project demonstration	50 days	02/10/2020	03/30/2020
11	Demo 1	1 day	04/01/2020	04/01/2020
12	Create a functional SmartPark website	25 days	04/02/2020	04/27/2020
13	Complete voice-assistance implementation	22 days	04/05/2020	04/27/2020
14	Demo 2	1 day	04/27/2020	04/27/2020
15	Testing and debugging all project components	10 days	04/27/2020	05/07/2020
16	Integration	54 days	03/15/2020	05/07/2020



Breakdown of Responsibilities

Team Member	Responsibilities
Swetha	**Front End** - Reservation Page - Admin Login Page - Home Page **Back End** - Manager - Auth0 authorization - Reservation Manager - Reservation - RainCheck - Walk-in - Confirmed Reservation - GuaranteedReservation
Aniqa	**Front End** - Statistics Page - Graphs - Date Search - Calendar Page - Calendar View - Event Search **Back End** - StatsManager - EventManager - Event
Jeffrey	**Front End** - Garage View - Overview - Edit Layout - Image Render **Back End** - Overview - EditGarageLayout - Image Render
Charles	**Front End** - Pricing Page - Graphs

	- Date Search **<u>Back End</u>** - Pricing Page - Dynamic Pricing Algorithm - PriceManager

Swetha will coordinate the integration testing. All team members will perform integration tests and unit tests for the classes and modules they are each responsible for as listed above. Jeffrey and Aniqa will help Swetha test the Reservation subsystem by creating reservation test cases and deleting/editing them. The Garage View subsystem will be tested by everyone in the group as they edit the garage layout and explore the virtual garage render. Everyone in the Manager Group has tested Manager access by logging in with their premade administrative login credentials. The entire Parking Garage group will view the Calendar while the Manager Group creates Event test cases.

3. Elevator Operation Subgroup

Project Coordination & Progress Report

As demo 1 swiftly approaches, the elevator group has planned and is working on completing four separate tasks: creating a functional front-end interface for the elevator, designing functions in order to communicate with the database- a connection between the front-end and the back end, a dynamic pricing model, and a means to simulate a vehicle going through the elevator process. For the elevator operation subgroup, this will cover the following use cases:

UC-14 - View Garage Usage Statistics: for the dynamic pricing model, the garage usage statistics will be utilized to create the dynamic pricing model and derive pricing plans based on this data.

UC-15 - Scan License Plate: built from a combination of the front-end interface implementation and the back-end functions connecting to the database. The front end will call the license plate scanning function while displaying a screen which says the license plate is being scanned. While this is occuring, the backend function searches the database to try and find a matching license plate. If the license plate is found, this value is returned to the front end of the elevator terminal and then begins to search for a reservation. Otherwise, an error value is returned to the front end and execution is shifted to a different branch, where the user is prompted for their membership number.

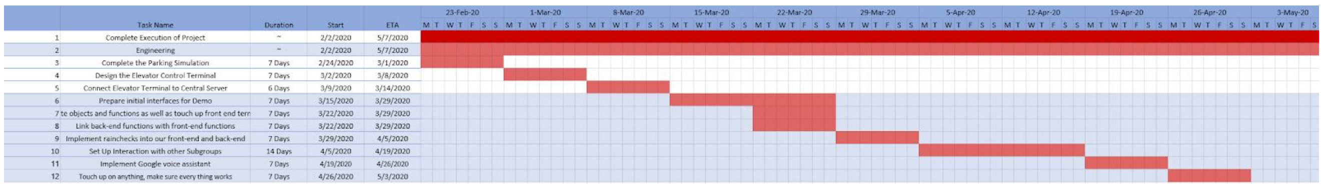
UC-17 - Elevator Terminal Output: the entirety of the front end of the elevator interface. The elevator terminal output is the display that the user interacts with when they enter the elevator. The elevator terminal must be able to receive data from the backend and know what to do based on the obtained values.

UC-18 - Update Parking Spot Status: a function of the backend of the elevator terminal in combination with the simulation. When the simulation has the car enter or exit the parking spot, the weight sensor in the parking spot activates a function in the backend which updates the spot status in the database. This update can then be retrieved by various other components of the

SmartPark application, such as by managers to check on spot availability, customers to choose a parking spot, or the elevator terminal to determine if a rain check needs to be issued.
UC-19 - Park: the entirety of the simulation process. The simulation process will be controlled by using push buttons in the first iteration in order to make decisions which the front end will display.

All of these use cases are currently being worked on. The goal is to have each of these use cases implemented by the first demo, and work on improving and adding onto them as the semester continues. For the first demo, these use cases will be kept fairly simplistic and will be built on for the second demo. As of right now, use case 14 is nearly complete, with use case 17 close behind.

Projected Milestones



	Task Name	Duration	Start	ETA
1	Complete Execution of Project	~	2/2/2020	5/7/2020
2	Engineering	~	2/2/2020	5/7/2020
3	Complete the Parking Simulation	7 Days	2/24/2020	3/1/2020
4	Design the Elevator Control Terminal	7 Days	3/2/2020	3/8/2020
5	Connect Elevator Terminal to Central Server	6 Days	3/9/2020	3/14/2020
6	Prepare initial interfaces for Demo	7 Days	3/15/2020	3/29/2020
7	te objects and functions as well as touch up front end tern	7 Days	3/22/2020	3/29/2020
8	Link back-end functions with front-end functions	7 Days	3/22/2020	3/29/2020
9	Implement rainchecks into our front-end and back-end	7 Days	3/29/2020	4/5/2020
10	Set Up Interaction with other Subgroups	14 Days	4/5/2020	4/19/2020
11	Implement Google voice assistant	7 Days	4/19/2020	4/26/2020
12	Touch up on anything, make sure every thing works	7 Days	4/26/2020	5/3/2020

Breakdown of Responsibilities

Member	Responsibility
Charles	● Coordinating the dynamic pricing model with the Manager group that will be used in the elevator terminal as well ● Setting up class objects with functions associated with the dynamic pricing model ● Setting up the routers and HTTP requests (backend)
Nicholas	● Front-end for the elevator terminal ● Working with React.js libraries and using JSX for the UI ● Libraries working with includes reactstrap
Disha	● Working with Manager group on Google Voice Assistant and Google Action ● Front-end using react for scanning car license plates and associated functions

	● Connecting different front-end components (navbar) as well as connecting to the back-end ● Working on the skeletal form of the simulation outlining different scenarios.
Thomas	● Backend for spots and license plate/users and associated routers, classes and functions ● Connecting back-end to front-end ● Working with Manager group to have consistency with the database that both groups will be drawing from

References

- Draw.io - UML Diagrams
- SOLID Design Principles - https://en.wikipedia.org/wiki/SOLID
- Composition over inheritance (aka Composite reuse principle) - https://en.wikipedia.org/wiki/Composition_over_inheritance
- Don't Repeat Yourself - DRY - https://en.wikipedia.org/wiki/Don%27t_repeat_yourself
- Inversion of Control - IoC (aka Hollywood Principle) - https://en.wikipedia.org/wiki/Inversion_of_control
- You Aren't Gonna Need It - YAGNI - https://en.wikipedia.org/wiki/You_aren%27t_gonna_need_it
- Law of Demeter - LoD (aka Principle of Least Knowledge) - https://en.wikipedia.org/wiki/Law_of_Demeter
- Principle of Least Astonishment - PoLA - https://en.wikipedia.org/wiki/Principle_of_least_astonishment
- Minimum Viable Product - MVP - http://en.wikipedia.org/wiki/Minimum_viable_product
- HTTP - https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview
- GUI Design: Sun Microsystems, Inc. *Java Look and Feel Design Guidelines*. Mountain View, CA, 1999. Available at: http://java.sun.com/products/jlf/ed2/book/