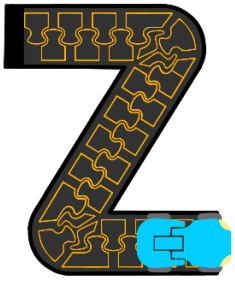

ZippyPark

An Optimized Automated Parking Garage

14:332:452 Software Engineering

Project Report 2

Team 2

Samantha Moy

Atmika Ponnusamy

Samantha Cheng

Kylie Chow

Andrew Ko

Parth Patel

Shreya Patel

Nandita Shenoy

Piotr Zakrevski

Submitted March 22, 2020

CONTRIBUTION BREAKDOWN

All team members contributed equally.

Contents

Section 1: INTERACTION DIAGRAMS

UC-4: Create Reservation

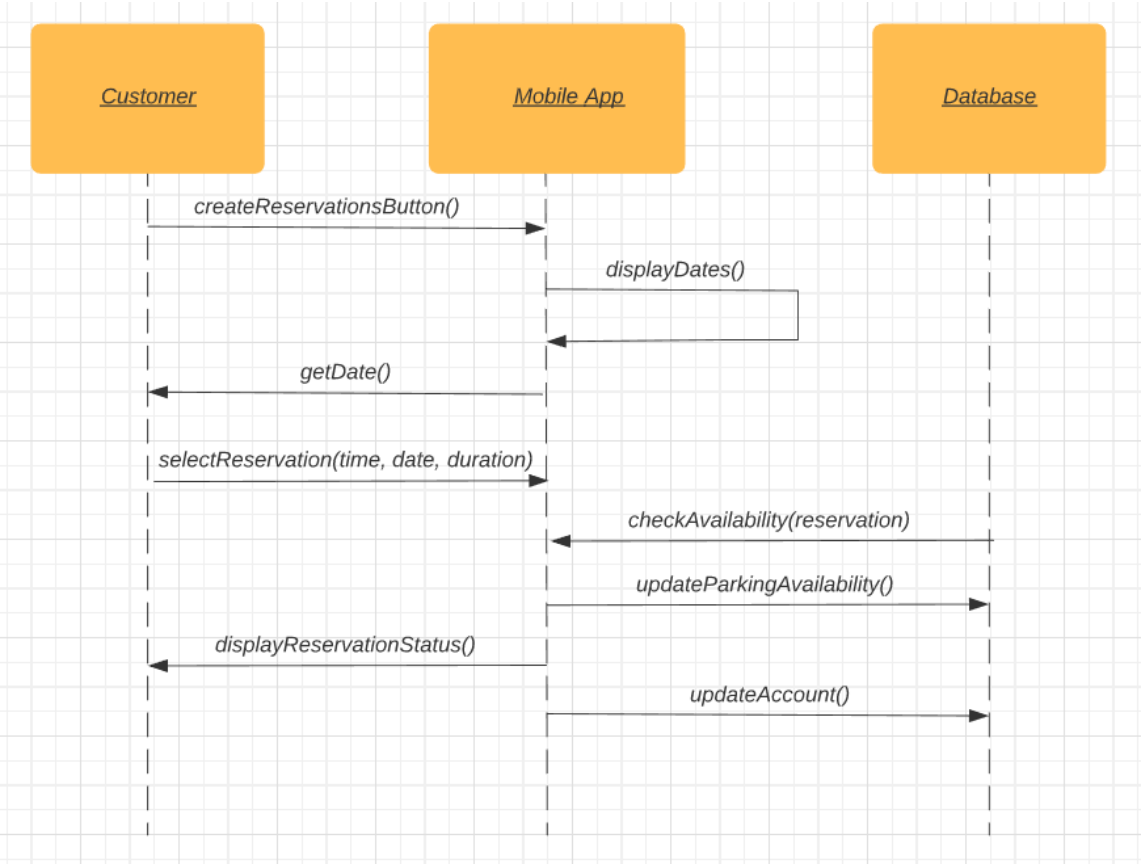

Figure 1. Interaction Diagram of UC-4.

UC-4 describes the process of creating a reservation. The interaction diagram of UC-4 was derived from the system sequence diagram from Report #1 (Figure 4) and is shown in Figure 1. No new domain concepts or objects were introduced. Concepts from the Report #1 domain model were used.

The mobile app is responsible for communication between the database and the customer. The app provides a centralized hub to accept input and display output on an organized user interface. As a result, the app will be managing the majority of the information. Because it already handles most of the processing, the app can readily provide the information to the customer and pull/push to the database. However, there is consequently more responsibility placed on the mobile app. Although separating the data processing and interface aspects would reduce the number of responsibilities on any given object, this strategy allows for more efficient communication and a smoother customer experience. The idea is to streamline communication to

limit latency during use while also limiting the number of dependencies between too many objects. By having fewer objects, the system is less likely to fail due to any single component, and the customer receives quick responses from the database. The final design allows customers to create their new reservation by selecting the desired date, time, and duration and then receive confirmation quickly. In the meantime, the mobile app manages the incoming information, checks availability with the database, and updates the customer's account information.

UC-5: Edit Reservation

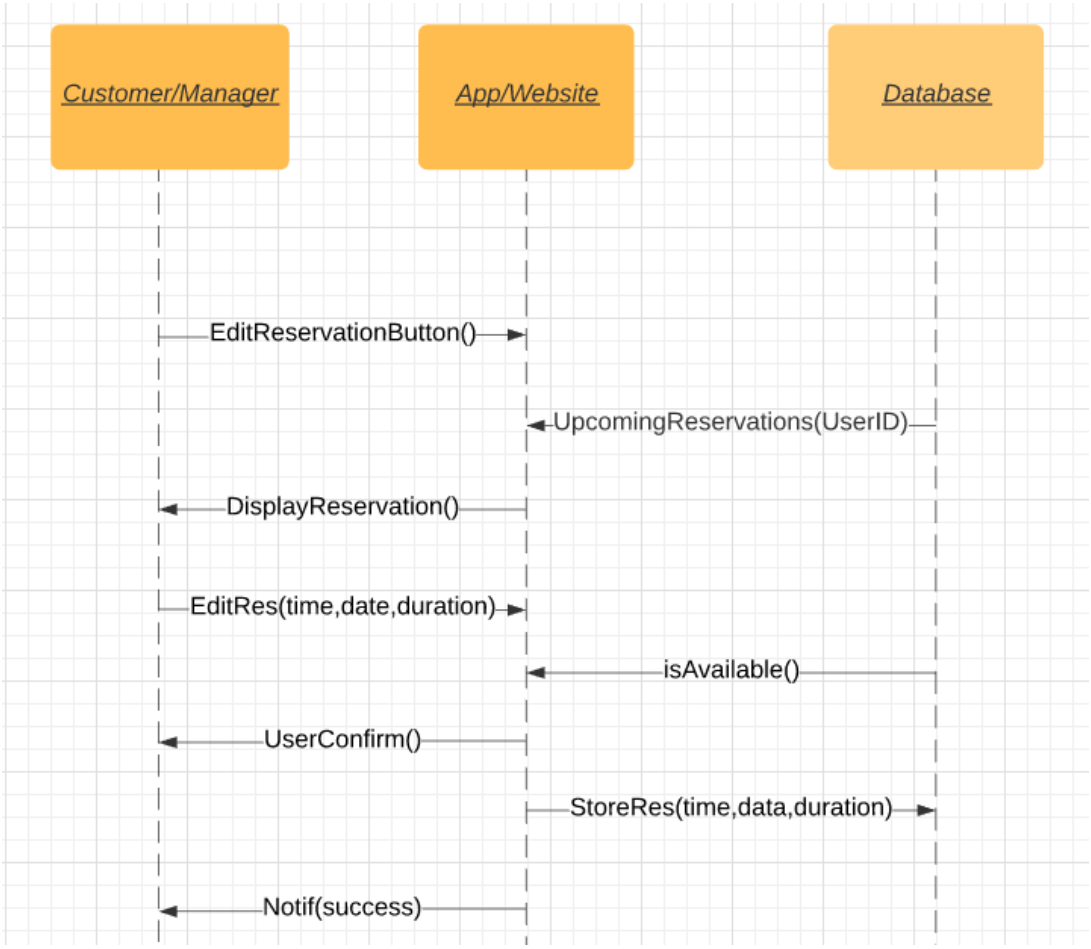

Figure 2. Interaction Diagram of UC-5.

UC-5 describes the process of editing a reservation. The interaction diagram of UC-5 was derived from the system sequence diagram from Report #1 (Figure 5) and is shown in Figure 2. No new domain concepts or objects were introduced. Concepts from the Report #1 domain model were used.

For UC-5, we assigned the responsibility of editing reservations to both user interfaces – the customer's app and the manager's website. Customers may make reservations through the app, and managers may do the same through their special manager website or portal. The app/website allows whichever user to interact with the database that holds all the information regarding reservations. Furthermore, allowing for an actor between the database and the customer/manager ensures that no one actor is communicating or computing too much. In this scenario, the database has the responsibility of sending and modifying information about a customer's reservations. A Publisher-Subscriber pattern is used here with the website/app being the publisher and the customer/manager being the subscriber. The subscriber inputs information, such as their user identification and the edits to a reservation. The publisher makes changes given the information from the subscriber; it communicates with the database to change the reservation as desired.

UC-7: Enter Garage

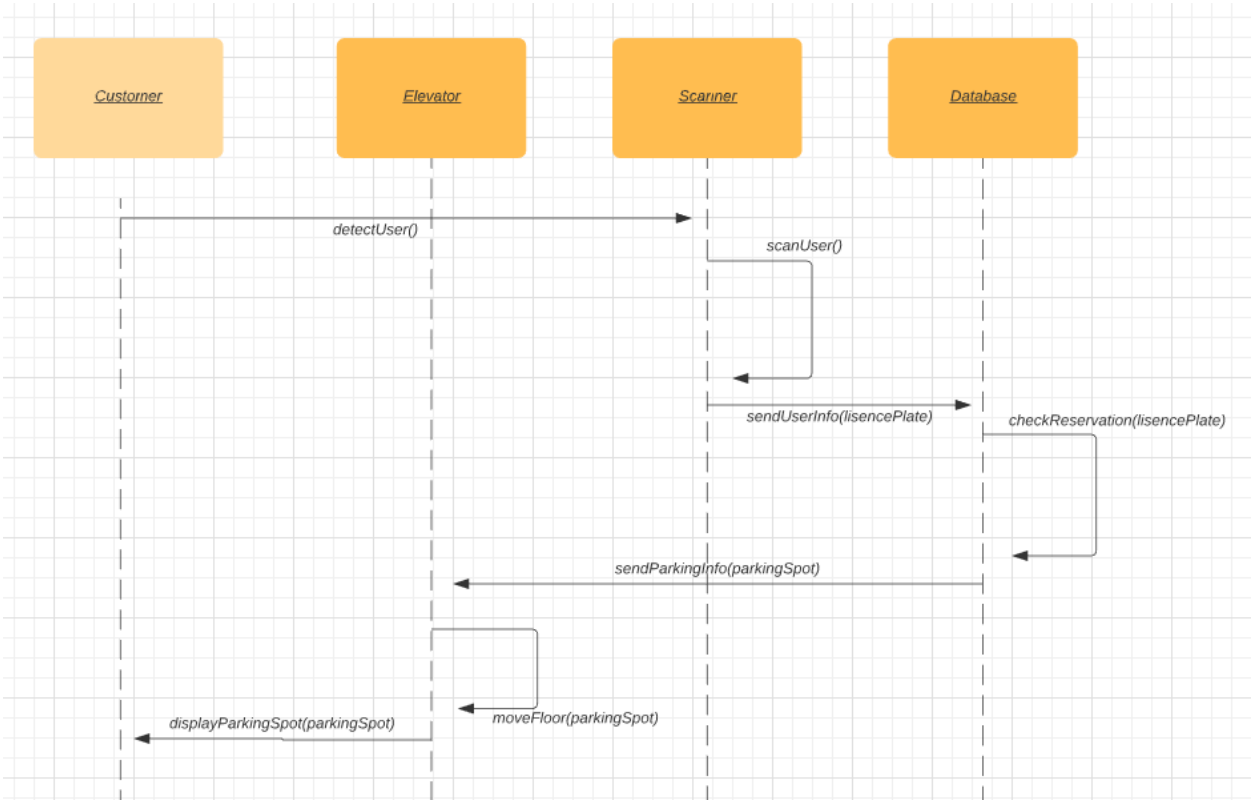

Figure 3. Interaction Diagram of UC-7.

UC-7 details the process of a customer entering the garage. The interaction diagram of UC-7 was derived from the system sequence diagram from Report #1 (Figure 6) and is shown in Figure 3. No new domain concepts or objects were introduced. Concepts from the Report #1 domain model were used.

This process starts with the customer driving onto the elevator, which the scanner detects. The scanner is responsible for obtaining and then sending the customer's data (license plate number) to the database. We have decided on this structure as this method requires minimal interaction from the customer's part. Since the only job of the customer is to drive onto the garage, they do not have to enter in any information as the scanner will automatically do it for them. The database is then responsible for finding and confirming the customer's existing reservation based on the data sent from the scanner. The database reports to the elevator an available parking spot. The elevator moves to the appropriate floor and displays what parking spot the customer should navigate to. This design further simplifies the customer's role as all verification is done through the database. The customer is then informed exactly where they should park and how to arrive at that location, making the process both easy for the customer and efficient for the system.

UC-10: Display Statistics

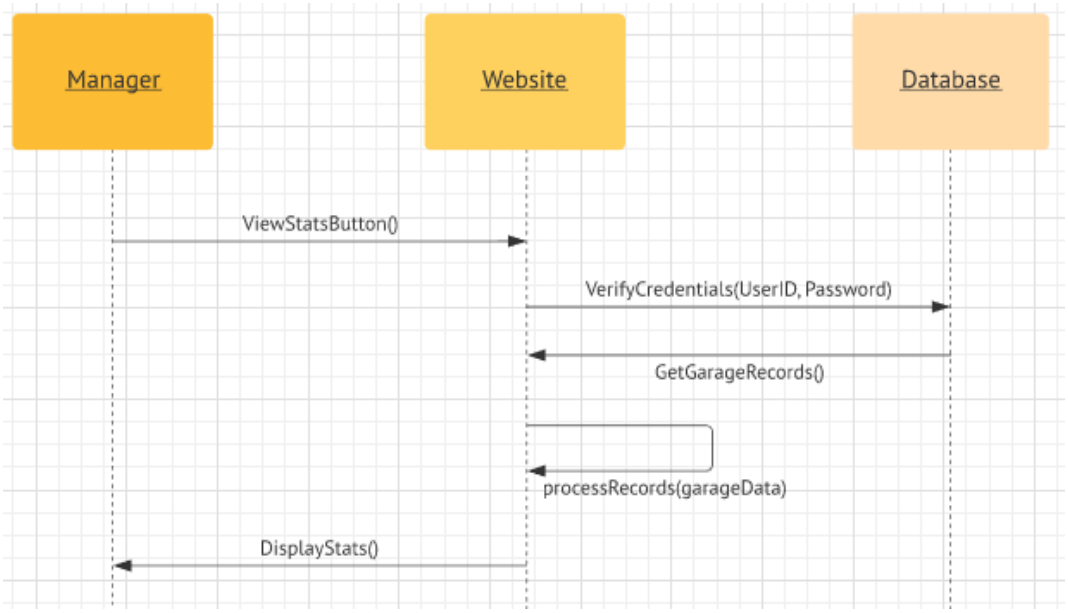

Figure 4. Interaction Diagram of UC-10.

UC-10 enables managers to view a variety of parking statistics for their parking garage. The interaction diagram of UC-10 was derived from the system sequence diagram from Report #1 (Figure 7) and is shown in Figure 4. No new domain concepts or objects were introduced. Concepts from the Report #1 domain model were used.

The website is responsible for pulling all parking data from the database and processing it to create a clean and easy to navigate page of parking statistics and metrics. We have decided on this structure so that the parking manager can quickly view relevant information about their garage on one page, then navigate to other pages (i.e. the dynamic pricing model editor) and make changes with relative ease. This way, the end-user has to put in the least amount of effort to manage and diagnose their garage. In this layout, the app/website handles all of the data processing and displays it directly on the site when complete to minimize latency between the initiating command and the final display. Once fully compiled, the garage manager will be able to view graphs of parking volume, daily revenue, and quantities of each reservation type over time intervals ranging from one week to one month. Along with the graphs, the manager will find a table of useful metrics such as the mean, median, mode, range, standard deviation, and variance of the parking datasets. The park manager will be able to view all of this information on a single site which will help him/her make proper business decisions for their garage.

Section 2: CLASS DIAGRAM AND INTERACE SPECIFICATION

2a) Class Diagram

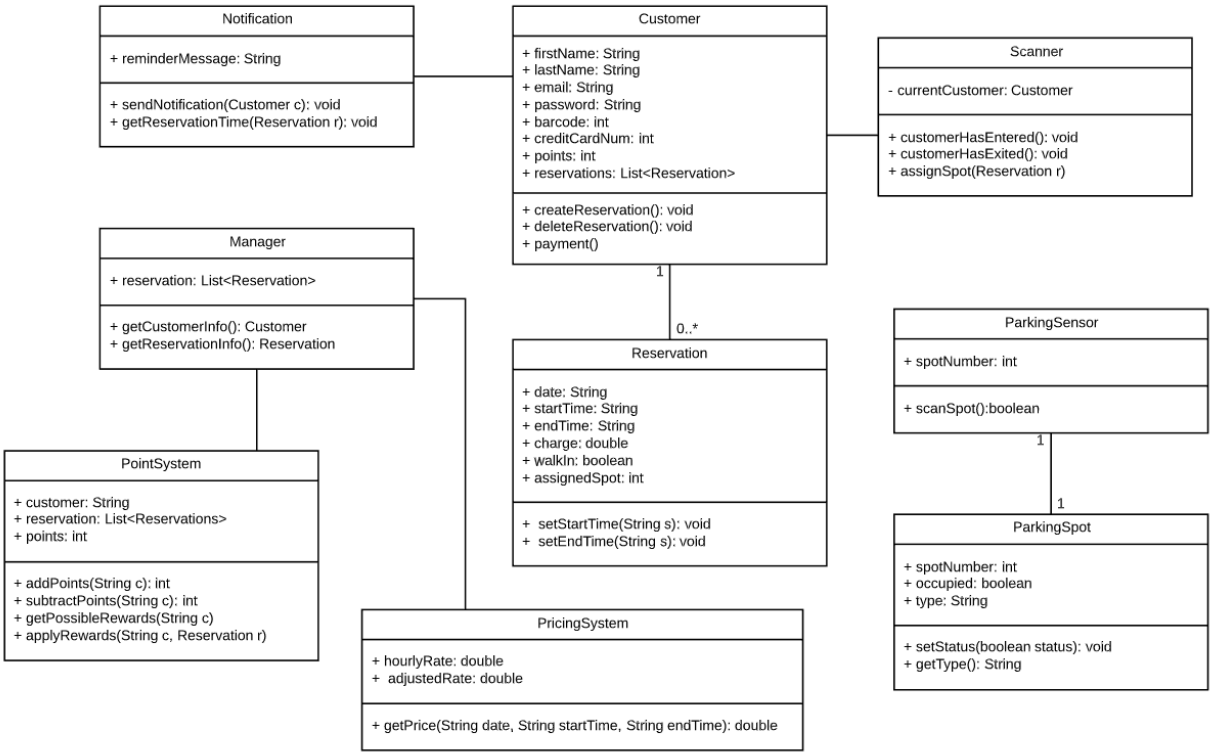

Figure 5. Full class diagram.

Each class in the Class Diagram originated from an abstract concept of the Report #1 domain model. An explicit outline of which classes/objects came from which domain model concept is further detailed beneath the table in “Section 2c: Traceability Matrix.”

2b) Data Types and Operation Signatures

The classes shown in the class diagram are enumerated below with descriptions of the class itself, its attributes, and its operations.

1. **Customer**

   Controls customer account information

   a. Attributes

      i. String firstName: customer’s first name

ii. String lastName: customer's last name
iii. String email: customer's email address
iv. String password: customer's account password
v. String phoneNum: customer's phone number
vi. int barcode: customer's unique barcode
vii. String licenseNum: customer's registered license plate number
viii. String creditCardType: customer's credit card type
ix. int creditCardNum: customer's credit card number for payment
x. String expDate: customer's credit card expiration date
xi. String CVV: customer's credit card CVV
xii. int numPoints: customer's number of reward points
xiii. List<Reservation> reservations: list of customer's reservations

b. Operations

i. createReservation(): creates a new reservation for the customer
ii. deleteReservation(): deletes an existing reservation for the customer
iii. payment(): uses credit card information to pay for reservation

2. Parking Sensor

Senses parking spot occupancy

a. Attributes

i. int spotNumber: number associated with a parking spot
ii. String type: type of parking spot

b. Operations

i. scanSpot(): scans parking spot to determine occupancy

3. Parking Spot

Tracks parking spot occupancy

a. Attributes

i. int spotNumber: number associated with a parking spot
ii. String type: type of parking spot
iii. boolean occupied: tracks whether spot is occupied or not

b. Operations

i. setStatus(): sets parking spot status to occupied or not occupied

4. **Scanner**

   Tracks customer entrances and exits

   a. Attributes

      i. Customer currentCustomer: customer that the scanner is tracking

   b. Operations

      i. customerHasEntered(): flags if a customer enters the parking garage

      ii. customerHasExited(): flags if a customer exits the parking garage

      iii. assignSpot(): assigns a parking spot to the current customer based on reservations

5. **Manager**

   Views reservations

   a. Attributes

      i. List<Reservation> reservation: list of all customer reservations

   b. Operations

      i. getCustomerInfo(): retrieves customer information

      ii. getReservationInfo(): retrieves reservation information

6. **Reservation**

   Creates a reservation

   a. Attributes

      i. String date: date of reservation

      ii. String startTime: start time of reservation

      iii. String endTime: end time of reservation

      iv. int barcode: scanned barcode associated with reservation

      v. double price: price of reservation

      vi. boolean walkIn: tracks whether the customer is a walk-in or not

      vii. boolean deleted: tracks whether reservation has been deleted or not

      viii. int assignedSpot: customer's assigned spot upon entering parking garage

   b. Operations

      i. setStartTime(): sets the start time of a reservation

      ii. setEndTime(): sets the end time of a reservation

7. **Point System**

   Tracks and changes customer points

   a. Attributes
      i. String customer: current customer profile
      ii. List<Reservations> reservation: list of current customer's reservations
      iii. int points: number of customer rewards points
   b. Operations
      i. addPoints(): adds points to customer profile
      ii. subtractPoints(): subtracts points from customer profile
      iii. getPossibleRewards(): views possible rewards given the number of customer points
      iv. applyRewards(): applies selected rewards to customer profile

8. **Dynamic Price System**

   Changes the parking price rate

   a. Attributes
      i. double hourlyRate: regular hourly rate for parking garage price
      ii. double adjustedRate: adjusted hourly rate for parking garage price
   b. Operations
      i. getPrice(): gets price of parking reservation given the date and start and end times

9. **Notifications**

   Sends notifications to the customer

   a. Attributes
      i. String reminderMessage: reminder message for customer to notify about the upcoming reservation
   b. Operations
      i. sendNotification(): sends notification to customer with reminder message
      ii. getReservationTime(): gets reservation date and time to notify customer with specific time

2c) Traceability Matrix

This matrix maps the software classes to the domain concepts. Further explanation of the mappings is given below for each enumerated domain concept.

		Domain Concepts				
		Database	Mobile App	Manager Portal	Scanner	Parking Spot Sensor
Software Classes	Customer Reservation	✔	✔	✔		
	Management Options	✔		✔		
	License Plate Scanner	✔			✔	
	Spot Sensor	✔				✔
	Payment	✔	✔	✔		
	Notification System	✔				
	Reward System	✔	✔	✔		
	Gate Entrance	✔				
	Gate Exit	✔				
	Database Management	✔		✔		
	Database Connection	✔	✔	✔	✔	✔

Table 1. Traceability Matrix of Software Classes & Domain Concepts.

All of the classes originated from the Report #1 domain concepts. See below for further explanation.

1. Database
    a. (All classes are derived from this concept because they all interact with the database).
2. Mobile App
    a. Customer Reservation – the mobile app has the capability to create, edit, and delete customer reservations
    b. Payment – the mobile app has customers input a payment method
    c. Reward System – the mobile app allows customers to trade points in for rewards under the reward system
    d. Database Connection – the mobile app must have database connection to interact with the database

3. Manager Portal
    a. Customer Reservation – the manager portal has the capability to create, edit, and delete customer reservations
    b. Manager Options – the manager portal allows managers to view statistic, set prices, and edit customer info and reservations
    c. Payment – the manager portal sets prices and can change payment methods
    d. Reward System – the manager portal sets the conditions of the reward system
    e. Database Management – the manager portal can make changes to the database (specifically the customer info and reservations)
    f. Database Connection – the manager portal must have database connection to interact with the database
4. Scanner
    a. License Plate Scanner – the license plate scanner is utilized to read license plates
    b. Database Connection – the scanner must have database connection to interact with the database
    c. Gate Entrance – the scanner tracks the entrance time of a vehicle
    d. Gate Exit – the scanner tracks the exit time of a vehicle
5. Parking Spot Sensor
    a. Spot Sensor – the spot sensor is utilized to indicate whether a spot is occupied or vacant
    b. Database Connection – the sensor must have database connection to interact with the database

Section 3: SYSTEM ARCHITECTURE AND SYSTEM DESIGN

3a) Architectural Styles

Our project will follow an “event-driven” architectural style. This is a style that focuses on how the system works as it responds to the events that occur. In our case, events are described as any significant change that is made in relation to the parking garage. For example, a change may be editing a reservation time or changing the occupancy status of a parking spot. These events result in the changing of the data held in the database. In the instance of editing a reservation, we would see the records change to hold the values of the new reservation as a result of the communication between the website/app and the database. With the parking spot more communication is required as the weight sensor must determine the spot to be “occupied” and not “vacant” and then send this information to the database.

The event-driven style is based on event emitters and event consumers. Additionally, event channels may be involved to connect the two. Emitters may also be referred to as agents and the consumers are also known as sinks. An agent is essentially the component that detects the change or the fact that an event occurred. In our example of parking the car, the emitter would be the weight sensor that determines that a car has taken up a certain spot. The sink is what reacts to the event that occurred. Following our example, the database is the collector that receives a notification of this change and reacts by storing “occupied” in a spot that was formerly “vacant”. In general, our project’s specific architectural style will always refer to the database as the sink since it is the actor that always ends up taking note of the changes. In most cases, we would see the app/website as the event channel that works to communicate between the event emitter and the event consumer.

3b) Identifying Subsystems

Our website runs on the typical client server technology where the server handles requests from the client through an Android app. The server or database is able to collect data through either one or multiple devices, depending on the usage. The server interacts with the user interface which is the app, the manager portal, and the hardware devices such as our scanners and parking sensors. The server receives information from the app which is then updated and

verified using the hardware. The history of that information is updated to the manager portal. Using the app, manager portal, and the hardware devices the database is able to store a full history of every customer's profile information and reservation. An outline of our subsystems is shown in Figure 6.

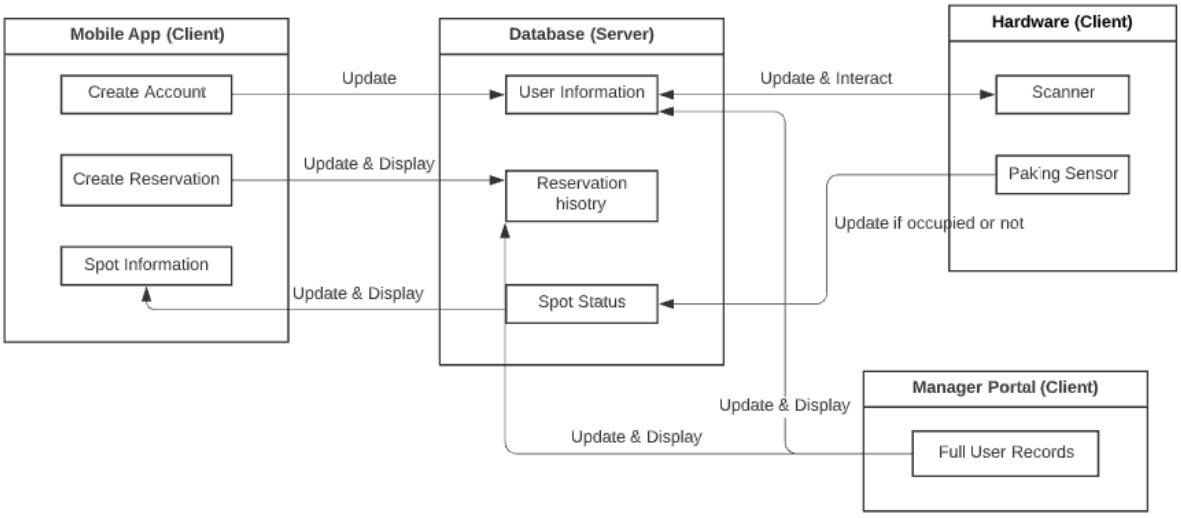

Figure 6. UML Package Diagram of Subsystems.

3c) Mapping Subsystems to Hardware

There are multiple hardware components required for the system to function smoothly. The main computer will host the server/database and can receive requests and interact with clients. Such requests can come from customers who are using either the app or the web browser from a personal device. Any customer can request an action at any time, meaning many hardware devices are being used by the system at once, or very few. Of course, the computer hosting the server/database must always be functioning for the system to work.

Another notable client would be a manager, who can also interact with the server/database from a personal device. Just like the customer, the manager can link his hardware device to the system, request certain actions, and then unlink his device from the system, making the hardware non-essential to the continuous uptime of the system.

The last notable client is the garage itself, where the hardware consists of a scanner and a sensor. The scanner both sends information and requests from the server, while the sensor only

sends information to the server. Unlike the other clients in this system, the garage hardware is essential to the system and needs to be running at all times for the system to continuously function. Figure 6 has a visual representation of these systems.

3d) Persistent Data Storage

A MySQL database is used to store all information that is required for use by all the components of this project. This database is comprised of seven separate tables:

1. CustomerInfo: stores basic customer data required to construct an account (i.e. first name, last name, email, phone number, license plate number, vehicle registration number, credit card information, as well as an auto-generated unique user barcode which serves as the user's identification number).
2. ParkingSpots: a map of the current status of each parking spot in the garage (i.e. spot number, type [handicap, walk-in, VIP, reserved], and status [occupied or vacant]).
3. Reservations: contains a record of currently active reservations (i.e. date, start time, end time, user barcode, assigned spot number, and total charge amount).
4. Walk-ins: contains a record of current walk-in customers who are parked at the garage (i.e. date, start time, user barcode, assigned spot number, and total charge amount). The end time is not stored in this table, since we don't know exactly when the walk-in customer will leave, so this information is stored in the Records table.
5. Records: contains a history of all reservations that have ever been placed at the garage (i.e. date, start time, end time, user barcode, assigned spot, charge, cancelled/not).
6. Payment: contains the pricing scheme used to calculate a customer's fee (i.e. base price and multiplier for each hour of the day).
7. Points: contains the reward/penalty scheme to calculate a customer's new point balance with each action (i.e. action taken and associated number of points added/detracted).

All information stored in these tables will be modified and/or accessed by the mobile app, manager portal, and scanner systems.

3e) Network Protocol

The system uses the common network protocol HTTP, or hypertext transfer protocol, for data communication on the manager portal website. Communication between clients and servers will be done specifically using the REST, or representational state transfer, architectural style. The most common HTTP methods used for this system are GET, POST, PUT, and DELETE for system users to obtain, add, and delete information from the MySQL database.

3f) Global Control Flow

Execution Orderness:

The parking garage system is both procedure-driven and event-driven. The entire process of staying at the garage includes several steps that customers will generally follow. They will log in to the system, or register if they do not have an account, and then they will make a reservation by selecting a date, start time, and end time. Once the reservation date arrives, customers are scanned in, stay for a predetermined period of time, and leave once their reservation expires. These steps make for a procedure-driven system. However, multiple scenarios can trigger events throughout the use of the system. For example, the parking process is initiated once the camera detects an approaching vehicle. The system is waiting for this event to occur before checking for reservations in the database. An unsuccessful license plate scan can signal to a separate barcode code scanner to turn on as an alternate source for verification. Finally, the system can trigger point deductions once a timer detects that a customer has overstayed their booked reservation.

Time Dependency:

The system operates in real-time, regularly comparing the current time to the reservations of incoming customers. This includes timers that track how long individuals stay during their reservations. The timers are used to ensure that customers arrive and leave on time. If a customer overstays their reservation, then the timer triggers a point penalty to their account.

Concurrency:

The parking garage system does not directly implement multiple threads to handle simultaneous inputs and outputs. Because the system is centralized around the mobile app and

the website communicating with the MySQL database, concurrent requests will be handled by the server. Multiple reservations that begin or end at the same time are processed by the database such that they are properly linked with the appropriate accounts.

3g) Hardware Requirements

1. Application/Website Hosting Server
    a. 1 mb/s connection speed
    b. 100 mb disk space
2. Database Host
    a. 10 gb disk space
    b. 4 gb memory
    c. 100 mb/s transfer speed
3. Garage Hardware*
    a. Camera/Scanner
    b. Parking IR Motion Sensors
    c. Raspberry Pi Model 3A
    d. 32'' Displays for Parking Instruction

At the input layer, customers and garage managers will interact directly with the mobile application or website. Both of these will be hosted on local servers with the design specifications listed above.

All of the data entered by the customers and managers will then be stored on the database server hosted by the Rutgers Engineering Computing Services (ECS). System administrators will be given permission to read and write directly to the database specified above.

Inside the garage, a scanner will read in the customer's license plate or QR code. Following a successful scan, the system utilizes a Raspberry Pi with a display to instruct the customer of their specific parking location. Finally, the system monitors occupied and vacant parking spots with infrared motion sensors.

**Note – the garage hardware will be simulated by GUIs for the purpose of this course.*

Section 4: ALGORITHMS AND DATA STRUCTURES

4a) Algorithms

The system has simple algorithms that control parking spot selection, the pricing scheme, and the point reward structure.

Ideally, a system should limit the number of steps a user must take to interact with the system. Therefore, to achieve a minimalist design, the system selects a parking spot when a customer makes a reservation or enters the garage as a walk-in. If the customer requires parking in one of the special spots (handicap, VIP, etc.), the system will select a spot within the designated area. The system selects a spot for the customer by using a MySQL "Update" statement. The first row with a "Status" of "Vacant" is selected for the customer, and the status is changed to "Occupied" in the "Parking Spots" and "Reservations" tables.

Parking spot prices are dependent on multipliers. On the manager website, the parking garage manager can set a base price per every 15 minutes and multiplier values for different times of the day. Theoretically, a larger multiplier would be assigned during rush hours. A customer's total fee is determined by the following equation:

$$Customer\ Fee = Base\ Price * Number\ of\ 15\ minute\ intervals * Multiplier$$

The point reward structure is similar to the pricing scheme in the sense that the manager can adjust the number of points rewarded and deducted with each customer action. As a customer fulfills a task, a MySQL "Update" statement will adjust the number of points that customer has in the "Customer Info" table accordingly. For instance, a manager may assign ten points to the action of making a reservation. Therefore, when a hypothetical customer with barcode ID, 1234, makes a reservation, the value in the customer's "Points" column will increase by ten. The example MySQL statement is shown:

$$UPDATE\ CustomerInfo$$
$$SET\ points = points + 1$$
$$WHERE\ barcode = 1234$$

4b) Data Structures

None of the subsystems use data structures. However, all subsystems interact with a MySQL database on the server provided by Rutgers ECS. This database has multiple tables that store all of the information that our subsystems access and modify using MySQL statements such as “Select,” “Update,” “Insert,” and “Delete.” See the section “Persistent Data Storage” for more detailed descriptions of each of the database tables.

Note that contrary to past projects, the parking garage is not simulated as a two-dimensional array. Instead, there is a database table named “ParkingSpots” with three columns, “SpotNum,” “Type,” and “Status.” For the sake of this explanation, let us pretend that the garage has 30 spots in total. “ParkingSpots” is populated with 30 entries numbered one to 30. The first 5 entries have Type “Handicap,” the next are “Walk-In,” the next are “VIP,” and the rest are “Reserved.” The “Status” field is changed to “Vacant” or “Occupied” based on the occupancy status of that spot at the present time. Therefore, this table serves as a “real-time view” of the garage. Each parking spot in the physical garage is assigned a number that corresponds to a row with that “SpotNum” value. We chose to represent the garage in a fashion that does not mirror the physical building so that our software can easily adapt to different garages. Parking garages can be square, rectangle, or L-shaped, as well as differ in overall capacity. It would have been difficult to adapt a two-dimensional array to these various scenarios, while it is simple to add more parking spot entries to our table.

Section 5: USER INTERFACE DESIGN & IMPLEMENTATION

The interface design we had initially proposed consisted of several screens depicting a web application, however we have made several modifications to our project since then and these design choices are illustrated in the new mockups shown below. The significant change is that customers will use a mobile phone application instead of a web application. All other changes are mainly stylistic alterations, such as choosing a new color palette. The original mockups were already designed to reduce the user effort, so our updated interface was created with the old setup in mind. These new mockups depict the activities which will make up the customer's mobile application.

Use Case #1

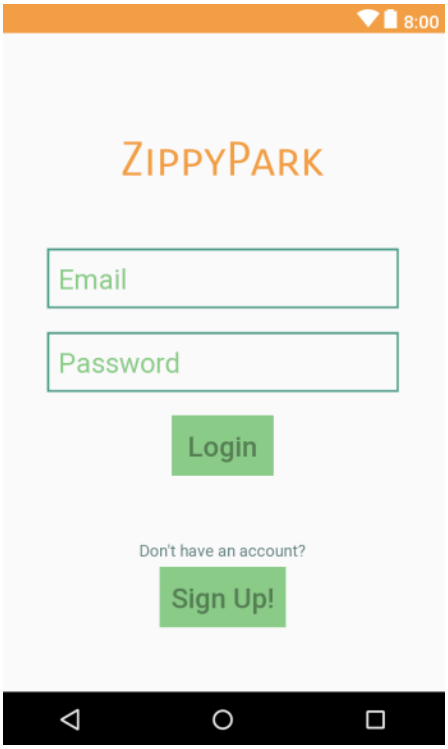

Figure 7. Customer UI - Login Page.

Login Page

Upon initial installation of the app, the user will be asked to login with the email and password associated with their existing account. In the case that they do not have an account, they are directed to the "Sign Up!" button at the bottom of the screen which leads to the registration form activity.

Worst case: 1 button click, variable number of keystrokes

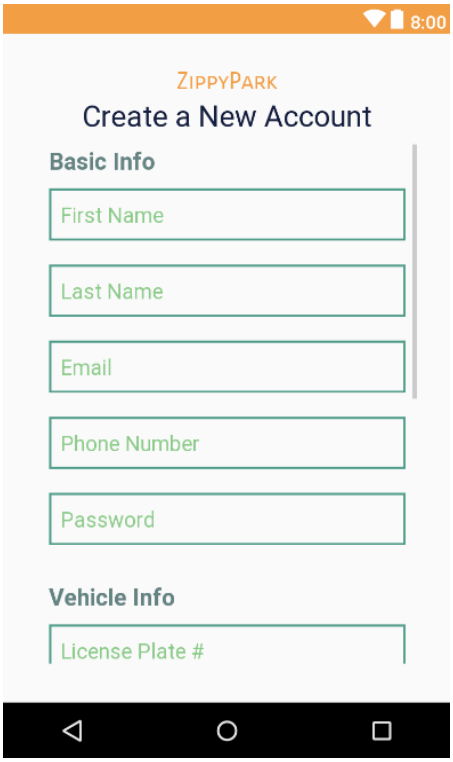

Figure 8. Customer UI - Create Account Page (part 1).

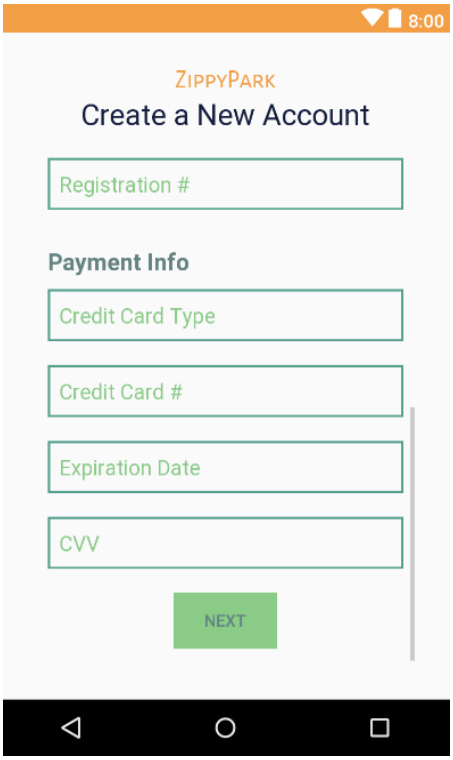

Figure 9. Customer UI - Create Account Page (part 2).

Registration Form

This activity contains a scrolling list, consisting of text fields for the user to fill out with the information necessary in order to create an account. Each text field contains a hint, indicating to the user what information is required to be inputted into that field. Once the form has been completed, the user can press the "NEXT" button, which will lead to the Home Screen. All fields will be checked for properly formatted responses. In the case that the "NEXT" button is pressed but certain fields are found empty or improperly formatted, then the user will be prompted to correct these errors and the new account will not be created until all errors have been resolved.

Worst Case: 1 click, variable number of keystrokes

Use Case #2, 3, 4, 5, 6

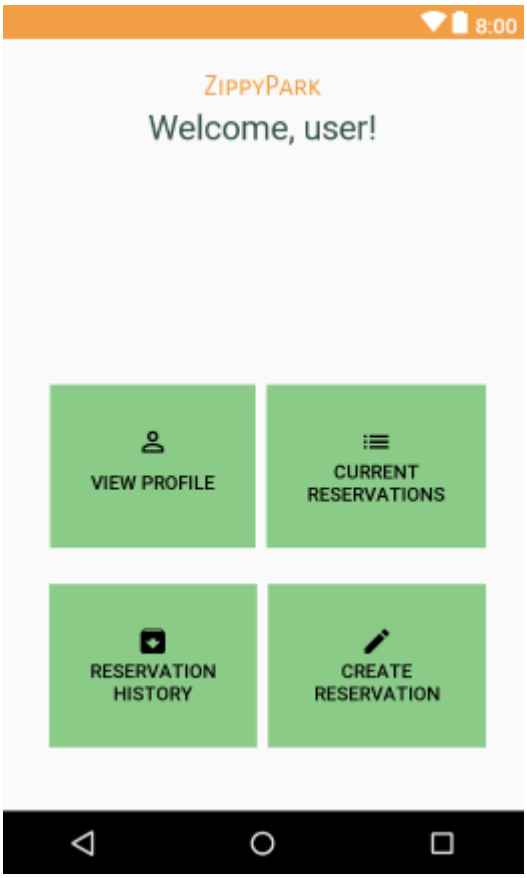

Figure 10. Customer UI - Home Screen.

Home Screen

The Home Screen of the ZippyPark application serves as the user's main dashboard for all key actions. At the top of the screen, the app name is displayed along with a text that will be customized to display the customer's first name. Below that they can select to perform a specific action. Each button will lead to a separate activity where the user can complete that action.

Worst Case: 1 button click

Section 6: DESIGN OF TESTS

Use Case Testing

For the sake of thorough planning, the following are the test plans for all of the use cases we intend to implement for the final product. For the first demo, we will implement and test the most essential use cases, which will be denoted with an asterisk*.

***UC #1 – Create Account**

A user should be able to register for an account by entering a valid email address and creating a unique password. These would be the credentials that would be needed every single time the user plans to log in to their account. They will also be asked to enter payment information and add at the very least details for one car. They would also be assigned a unique barcode attached to their account.

Test Coverage: For the email address we will initiate this test with 10 test cases randomly generated. For the password, we will initiate the test with 2 kinds of inputs for the password. One of them will have valid inputs, while the second one will have invalid inputs for the password, such as too short of a length, which would be rejected. There will be 2 test cases for the payment information; the first one will be tested by adding a valid input for the credit card number and the other one will have an invalid input for the credit card number.

***UC #2 – Edit Account**

A user should be able to edit their account whenever they need. They should be able to change their password, update their credit card information, and add or delete car license plate registrations.

Test Coverage: For the password change we will test it in 2 different ways. One of them will be done by the app asking for the old password and then the new one, and checking if the password updated. In the other case, an invalid password would be provided, and we will check that the original password is retained. We would once again require 2 test

cases for the credit card information. The first one would involve updating the credit card number with a valid credit card input to check if the previous credit information is replaced, and the second would have an invalid credit card input to check if the older information is still retained.

***UC #3 – Delete Account**

The garage owner should have the ability to delete an account through the manager portal. This would be done in the case of an inactive account.

Test Coverage: A test case for this scenario would be to attempt to delete an account as the manager. Once the account is deleted a notification of success will be received. The database will no longer hold information on this account with the exception of the records table.

***UC #4 – Create Reservation**

A user should be able to create a reservation for a parking spot for a selected time. They can select a date and time by looking at the calendar which shows an updated version of the amount of reservations there are currently. The user would get an email notification for their confirmed reservation. They would also have the amount deducted from their bank account after the reservation is confirmed.

Test Coverage: We will be testing if after the reservation is made, the user gets a notification for the reservation and is available to be viewed in their reservation history. We will also be testing if the customer's fee charge is sent to their credit card service by awaiting a confirmation message after a transaction. Another test for the reservation is to show the calendar times as unavailable if they try to reserve during a time where the entire garage has already been previously reserved.

UC #5 – Edit Reservation

A manager or customer should be able to edit an upcoming reservation whenever they need. The user can move the date of their reservation or change the start time or end time of the reservation. A customer would receive an email notification for the new reservation and likely a fee change.

Test Coverage: A test case for this scenario would be to create a reservation and then attempt to change the time, date, or duration. If the desired time is available, the changes will be made and the database updated with the new information in place of the old. If the duration is changed, the user will see and increase/decrease in what they owe. If not available, the database is not updated-The user may use a different button to delete the reservation if they choose.

UC #6 – Delete Reservation

The manager or customers should be able to delete reservation at least 1 hour before the reservation time to get a full refund. They would receive an email notification for confirmation for the deleted reservation.

Test Coverage: For the reservation we will have a test case where we will delete an upcoming reservation before the 1-hour limit to see if a notification is sent via email to the user and if it is omitted from the user's reservation history. We will also be checking if the user is provided a full refund. In another test case we will delete an upcoming reservation only 15 minutes prior to the reservation time to see if a partial refund is given back to the user. The databases are updated to no longer hold information regarding these reservations with the exception of the records which show the reservations as cancelled.

***UC #7 – Enter Garage**

A user should be able to easily drive onto the elevator with the scanner being able to read the user's license plate. The elevator should then move to the correct floor and the user should be prompted to which parking spot to navigate to.

Test Coverage: There are multiple scenarios that need to be covered. The user may or may not have a reservation, the system of the database and the scanner must determine this by looking for existing reservations under the user's license plate. Both scenarios will be tested. If the user has a reservation the elevator should move to an appropriate floor and direct the user to an empty spot. If the user has no reservation, the elevator should remain on the first floor and check if any spots are available. If a spot is available, the user should be directed to it otherwise, the user should be told to exit the garage. The test coverage takes in different license plates as inputs and outputs the location the user should go to.

***UC #8 – Exit Garage**

A user should be able to easily drive out of the garage from any floor. When the user drives out, the system should be able to tell what user drove out, and then compile a list of statistics including total duration of the stay, type of reservation, and if the user was late in leaving or not. This information should then be visible to the user in his past reservations.

Test Coverage: To test whether the statistics are working correctly, a list of different users will enter and exit the garage. As each user exits the garage, the system should be able to scan which user left the garage, when they parked, when they left, and what their reservation status was (duration, not reserved, etc.). From this, the output for each user should be a comparison of the total time parked versus the reserved time and this information should then be sent to the database. The number of outputs should match with the number of inputs, users entered the garage, and data from one user should not be mixed with data from a different user.

***UC #9 – Update Spot Status**

The system should be able to tell when a spot is currently occupied by a user and when it is not. Every spot in the garage should be monitored while the system is running. The sensors at each spot should also be able to tell at what time a user parks in a spot, and then transitions that spot to an occupied state. The sensors should be able to tell at what time a user leaves a spot, and then transitions that spot to a free state, as well. These times should then be stored in the database.

Test Coverage: To test whether the parking spot's availability is being updated correctly we will "park" a car at a certain spot and see if that spot is now marked as occupied in the database. If we try to park another car at this spot, it should not be possible. When the car "leaves" the spot, we will check if that spot is now marked as free in the database. If we try to now park another car at this post, it should succeed. One final test is to fill up the entire walk-in floor, and then try to park another car at that floor. In this scenario the system should report that the floor is entirely occupied.

UC #10 – Display Statistics

The parking manager should be able to view parking trend data. This feature is provided by the managerial website whose access is limited to parking administration. The managerial website has a single page dedicated solely to garage statistics. The parking garage records are taken and that data is compiled into statistics which can be viewed in the form of graphs and other data models for certain time frames determined by the manager.

Test Coverage: Testing the statistics display will require the database to be populated with parking garage records for a long span of time. Populating data for the length of two years will be sufficient. If the manager chooses to view parking data for the month, they may specify that choice by selecting the option to view the data for the month. They may also choose to view data for the year or specify a start and end date for the data which they choose to view. The website page should display different data models for each of these cases.

UC #11 – Points Management

The parking manager should be able to adjust the reward scheme from the managerial website. This feature includes setting the number of points added/detracted for each action, as well as managing trading for awards. An optional feature is allowing the manager to view, add, or take away points from a customer manually. This feature would be in place in case the automatic point calculation fails.

Test Coverage: To test the reward scheme, the manager can set a condition for ten points to be exchanged for a certain service. A test customer account should be created, from which an action is executed. Ten points should be automatically loaded on the test account. From the customer account, the points should be able to be exchanged. On the managerial website, the manager should view that the points have been exchanged and the test account's point balance should be zero. To test the points management feature, the manager may navigate to the points management page of the managerial website and select this test account. The manager may deduct points, which would result in the balance being displayed as zero on both the manager and customer's side.

UC #12 – Payment

Customer fees should be calculated for the duration of their stay. Payments for reservations are made prior to the reserved time. Payments for walk-ins are made after the customer has left the garage. Additional charges are incurred for staying past a reservation time. These charges are sent to the customer's credit card service and the customer receives a receipt via email. The garage manager should be able to adjust the pricing scheme of the garage, specifically the base fee and multipliers which are reflective of the dynamic pricing model utilized in this system.

Test Coverage: To test whether payments are being deducted correctly, a test account must first be created. Then, multiple reservations during various time periods can be made to test that the dynamic pricing algorithm is working with the payment feature. It should be seen on the managerial website that these charges have been made before the

reservation time. These reservations should have different costs per hour depending on for which time period they are made. To test the fee payment, we will check the account into its reservation and wait until it is after the reserved exit time to check the account out. This should show another payment occurring after this action. For the walk-in scenario, we can simulate a walk-in appointment by checking an account into an open spot. We should see a payment as soon as the simulated walk-in leaves.

Integration Testing

Integration testing is performed through big-bang integration testing, where all modules are combined and functionality is verified after testing is completed. This is practical because our system is small and the convenience of this testing is a large benefit. Because smaller modular testing will be completed on each use case, errors within a use case will already have been debugged before integration testing.

Integration testing is split up into two main sections: testing involving data, and testing not involving data.

- Tests with data:
    - Garage → Manager Website
        - Upon simulating a car entering and exiting a parking spot, test that the manager sees the correct number of open parking spots on the website.
        - Upon a car exiting the parking garage, test that the appropriate reservation is moved from "Active Reservations" to "Records" on the manager website.
    - Customer App ↔ Manager Website
        - Upon creating or updating a customer profile or reservation, test that the manager sees this change in the appropriate webpage.
        - Upon updating a user's profile, reservation, or points from the manager end, test that the customer sees this change on their app.
    - *Data Validation*: Data is stored in a MySQL database. This can be checked during integration testing. We can also validate data in the form of XML files. Thus, whenever a change involving data is made, we check whether the XML files are

generated correctly, if they hold the correct data, and if they are being correctly transferred between modules.

- Tests without data:
    - *Cross-browser compatibility*: Though the customer app will be done through Android only, the manager website is viewable on different browsers. We will test that the website remains consistent and viewable on popular browsers such as Google Chrome, Safari, Firefox, and Internet Explorer.
    - *Verifying interface links*: In both the customer app and the manager website, interface links will be tested for all tabs and buttons that require redirection to a new tab. For example, clicking on the statistics tab on the homepage of the manager website should open the statistics page.

Additional Testing

Algorithms:

Parking Spot Selection

A user should automatically be assigned a parking space by the app/website upon entering the garage based on their parking preference. The system should communicate with the database to find a free spot in either the regular, handicap, or VIP parking sections.

Test Coverage: The functionality of this algorithm will be tested by requesting spots in all three sections while they are empty. Then, parking spots will be requested when all sections are all partially filled. Finally, parking spots will be requested when all sections are filled which should result in an error.

Dynamic Pricing

Parking spot prices should automatically change based on the time of day according to the dynamic pricing model. The model should be directly affected by changes made in the base price and multipliers set by the garage manager.

Test Coverage: The functionality of this algorithm will be tested by making reservations during rush hours, noon, and night-time. The parking quotes given by the app will be monitored to confirm that they match the pricing equation. In-app parking prices will also be monitored to see if they change with changes in base price and multipliers.

Point Rewards

A user should gain points for preferred behavior and lose points for infractions. The system and database should communicate to record changes in a user's points.

Test Coverage: The functionality of this algorithm will be tested by creating a test customer account and executing positive and negative actions, as well as reward exchanges. The customer account will be monitored to ensure that the point balance reflects the action performed.

Non-Functional Requirements:

REQ #31 – Accounting for Disability

The system should recognize if the account making a reservation has opted for disability features. During registration, customers have the option to request additional accommodations for disability and then verify their status using their unique ID code. For these users, the system will automatically reserve parking spots designated for disabled persons.

Test Coverage: We will be testing this by creating a reservation with two different types of accounts. The first type will be a normal account without the extra disability features.

The second type of account will be an account that has requested for disability accommodations. During these tests, we will be checking to ensure that accounts with disability features are provided with handicapped parking spots. However, only these accounts should be given handicapped parking spots for their reservations. For all other accounts, the reserved parking spot should not be handicapped.

REQ #32 – Storing into the Database

When the system interacts with the database, it should read from and write to the appropriate tables. To stay organized, the database is divided into seven tables ranging from account information to garage records. When a new account is created, a new reservation is made, or an existing reservation is edited or cancelled, the system should update the correct database table to accurately reflect the changes.

Test Coverage: We will observe the mobile app and website's interactions with the database during any actions the customers can take. These tests include creating and editing account information, making or cancelling reservations, and arriving or leaving a parking spot. We will test that all of the actions result in an entry in the appropriate table of the database, and that a copy is also logged in the records section of the database.

REQ #35 – Accepting Barcodes

When a customer arrives and their license plate is unrecognized, they will have the option to provide a barcode to verify their identity. This code is available on the profile page of the app and is linked to their account information.

Test Coverage: We will create a set of barcodes and connect them to profiles at a one-to-one ratio. When these codes are scanned into the system, they should be recognized and the system should display the respective account's pending reservations. However, we will also test sets of codes that are not linked to the system. In these tests, the system

should respond by displaying that the code is unrecognized and instruct the customer to either try again or exit.

User Interface Requirements

REQ #53 – Viewing Registered Customers (Manager)

When a manager logs into the system using an authorized account, they have access to a list of registered customers. This should pull all of the entries from the Customer Info table of the database and display them in an organized format.

Test Coverage: To test this, we will manually enter ten example customers into the database. Then, logged in as a manager, we will press the button to pull up registered customers. The app should access the database and output the list onscreen. We will repeat this multiple times with different customer information to represent unusual scenarios, such as customers with the same name. In each test, the database should pull and display the list of customers for the manager. For similar entries, the accounts will be differentiated by their unique user IDs.

REQ #55 – Edit Reward/Penalty System (Manager)

When a manager is logged in with an authorized account, they have access to the settings for the reward and penalty system. On this page, the manager can modify the rewards that customers can exchange their points for. They can also edit the penalties that occur when a customer dips below a threshold value, also changeable by the garage manager. The app refreshes to reflect these changes so that customers do not continue to use outdated information.

Test Coverage: We will test this functionality using three scenarios. In the first, the manager account will edit the reward/penalty settings and save the changes. Then, a customer account will open the points page on the app, where the updated settings should be displayed. In the second scenario, the customer account will already have the points

page open. If the manager account updates the settings during this time, the customer account will have outdated information. So, the system should inform the customer that the settings have been updated and prompt the customer to refresh the page. In the third scenario, a customer account will be at -15 points, while the threshold for a penalty will be -20. Should the manager adjust the threshold to -10, the customer will not be instantly penalized. Instead, the system will inform the customer that the penalty will occur the next time the customer incurs negative points due to bad behavior.

Section 7: PROJECT MANAGEMENT

General Workflow

After the first Software Engineering lecture, our roster was finalized and sent to Professor Marsic and Kartik Rattan. We agreed to appoint Samantha Moy as the project leader and Atmika Ponnusamy as the configuration manager. Mondays at 10:00am are reserved for full team meetings, scheduled as necessary. Sub-group meeting schedules are decided among members of each group. On Sundays before full-team meetings, Samantha M. will send a meeting agenda to the team. She records meeting notes on the agenda while leading the meeting. Members give updates on progress, discuss future work, and assign responsibilities. Our main form of communication is via Facebook Messenger. Samantha M. sends full-team reminders about meetings and deadlines through Messenger. We also use Slack to simplify communication as separate work on sub-projects begins.

Throughout the course of this semester, our team aims to collaborate efficiently by utilizing file sharing resources that are available to us. These resources, which include Google Drive and GitHub, will enable us to operate as one unit, while still allowing each member to make their own individual contributions to the project materials.

A Google Drive folder of resources will be updated throughout the semester. The folder includes contact information, calendars with deadlines, project resources, meeting agendas, and meeting minutes. This folder will serve as the primary tool for project management and will be overseen by Samantha M.

Team members who possess expertise regarding GitHub serve as configuration managers. Each sub-group has a configuration manager for their sub-project, and Atmika serves as the overall configuration manager. These individuals are responsible for ensuring that all code submitted to the project's central repository is cohesive and well-written, both syntactically and semantically. Each team member possesses a local working directory, as well as their own public repository from which the code enters a review process: the configuration manager reviews the code, and if they decide it is acceptable it is committed to the project's central repository. Otherwise, the code is rejected with feedback in order to be improved. See the section, "Breakdown of Responsibilities," for specific information about the division of work.

7a) Merging Contributions

Our team takes several steps to ensure seamless report integration at the end of each week. After a report section is submitted, Samantha M. immediately begins preparing for the next submission. For example, for Report 2 – part 2, she adds a document on the team Google Drive called "Report #2 – part 2 (draft)" to the folder labeled "Report #2." There is a folder for Report 1, 2, and 3. Every folder holds relevant writing pieces and report submissions. Samantha M. researches the upcoming report section and, in the draft document, outlines the requirements and possible ideas to consider. She divides the section into evenly-sized parts, and each teammate is asked to sign up for one part. If the week's report section requires diagrams, Atmika shares a template with the team so that everyone's diagram style is the same. Team member's finish writing their pieces on the document by 11:59pm on the Thursday before the deadline. Samantha M. spends Friday and Saturday compiling everyone's work. She checks that the content is correct, while also ensuring consistency and uniformity in the terminology and formatting.

7b) Project Coordination & Progress Report

As of February 27th, the shared database has been implemented. Customer accounts and reservations can be made directly to the database.

As of March 7th, Group 1 was designing the Android application screens so that customers can make accounts and reservations through the app instead. Group 2 was creating the scanner interface. Group 3 was working on the manager authentication screen. However, over the following two weeks, all subgroups encountered difficulty connecting to the database through external means (i.e. via Android app), as well as while off-campus, due to the strict Rutgers security setup. After many setbacks, we were able to devise a solution with the Rutgers ECS to establish connection so that we are able to work on the project as classes are being conducted remotely. Because of the delay, the timeline was restructured, but steady progress is still being made.

As of March 14th, remote connection difficulties were resolved, and the workflow is anticipated to be smoother from now on.

As of March 20th, use cases 7, 8, and 9 have been implemented. Since Group 2's work is completed, these team members will transition into Group 3.

By March 22nd, use cases 1, 2, 3 should be completed. Group 1 and 3 are currently working on these cases. With the postponement of Demo 1, additional use cases may be finished in time.

The only anticipated challenges are further delays from handling project-work and other coursework during this unusual time we are all attempting to adapt to (remote-class-instruction).

7c) Plan of Work

Our team has held a total of five full-team meetings since January 27th. Over the last few weeks, we have thoroughly outlined our project, sub-teams, responsibilities, and timeline. We have also designed and implemented our shared infrastructure and team communication systems. Since we have transitioned into sub-team work, we no longer need to meet weekly as a full-team. However, we occasionally gather on pre-scheduled days for full-team check-ins, such as in preparation for demos. We will also be in contact weekly when assembling the report portion that is due each week.

Initially, our team was apprehensive given our overall lack of software engineering experience. However, our team leaders have met with professors, advisors, and the Rutgers Engineer Computing Services (ECS) to gain insight into a project of this magnitude. The team has worked hard to use this advice to establish a strong foundation for the project during the first few weeks; we have had a clear business plan, basic software infrastructure, and knowledge of the technologies required since early in the semester.

We have fully transitioned into sub-project work, where each group is self-regulated. However, everyone will still abide by team deadlines and milestones that were set at the beginning of the semester.

The dates listed below are subject to change as the team and course require.

Note - the table shows a few recently passed, as well as upcoming, deadlines; not all of the deadlines are shown. See the Gantt chart and calendar of deadlines for more specific dates, as well as later ones.

Date	**Milestone**
March 5	Full team meeting & demo 1 prep.
March 21	Finish minimum demo 1 code.
March 24	Prep demo Q&A.
March 27	Finish demo 1 documentation. Finish additional demo 1 code.
March 29	Finish filming demo 1 video clips.
March 30	Finish compiling demo 1 video. Send demo materials.
March 25	Demo 1
April 6	Finish minimum demo 2 code.
April 6	Full team meeting & demo 2prep

Table 2. Upcoming team deadlines.

Located in our shared group resources folder, we have a Gantt Chart that shows the high-level overview of this project. This spreadsheet is maintained by Samantha M.

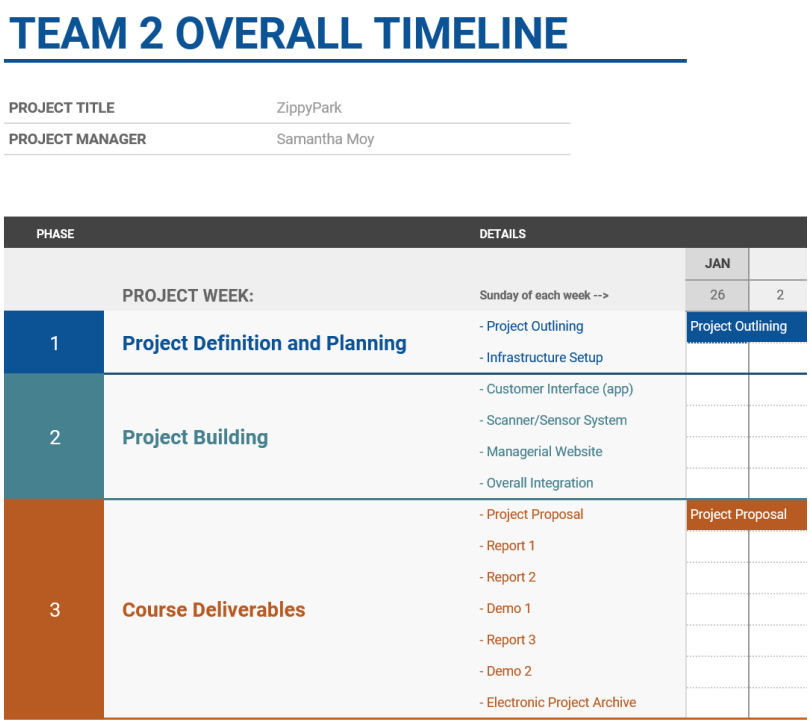

Figure 11. Preview of Gantt Chart (part 1).

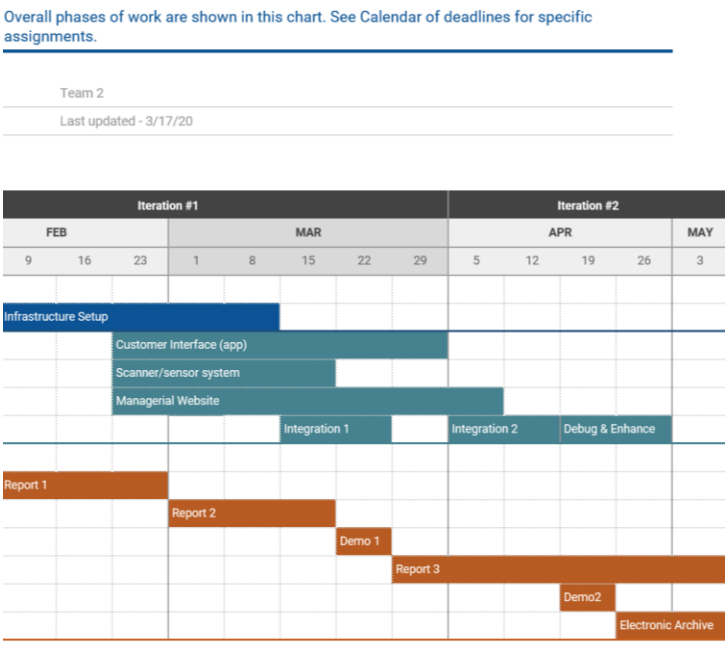

Figure 12. Preview of Gantt Chart (part 2).

The same spreadsheet may also be viewed here: https://tinyurl.com/team2Gantt

Additionally, we have a calendar of deadlines that is maintained by Samantha M. This spreadsheet serves as a simple way to view more specific project deadlines and milestones.

LEGEND
Course deadlines
Meetings
Writing deadlines
Suggested sub-project deadlines

Figure 13. Legend for calendar of deadlines.

FEBRUARY						
Sunday	**Monday**	**Tuesday**	**Wednesday**	**Thursday**	**Friday**	**Saturday**
26	27	28	29	30	31	**1**
Email team roster	Full Team Meeting			Finish writing your piece	Proposal assembled & submitted	
					Prof office hours about sub-projects	
2	**3**	**4**	**5**	**6**	**7**	**8**
Proposal due	Full Team Meeting			Finish writing your piece	Report assembled & submitted	
9	**10**	**11**	**12**	**13**	**14**	**15**
First report - p1 due	Full Team Meeting	Meeting with Sabian - ECS Server		Finish writing your piece	Report assembled & submitted	
	Establish project infrastructure					
	Git setup					
16	**17**	**18**	**19**	**20**	**21**	**22**
	Full Team Meeting					
First report - p2 due	All groups display environment			Finish writing your piece	Report assembled & submitted	
23	**24**	**25**	**26**	**27**	**28**	**29**
First report - full due	Group 1 screen setups			Finish writing your piece	Report assembled & submitted	
	Group 2 setup					
	Group 3 setup					

Figure 14. Preview of February calendar of deadlines.

MARCH						
Sunday	**Monday**	**Tuesday**	**Wednesday**	**Thursday**	**Friday**	**Saturday**
1	**2**	**3**	**4**	**5**	**6**	**7**
				Full Team Meeting - checkpoint		
Second report - p1 due				Finish writing your piece	Report assembled & submitted	
				Prep demo pieces (Group 1 & 2)		
8	**9**	**10**	**11**	**12**	**13**	**14**
Second report - p2 due		~MIDTERM~				Finish writing your piece
						External DB connection setup
15	**16**	**17**	**18**	**19**	**20**	**21**
Report assembled & submitted	~~~~~~~~~~~~~~~~ SPRING BREAK ~~~~~~~~~~~~~~~~					All groups finish demo code
22	**23**	**24**	**25**	**26**	**27**	**28**
Second report - full due		Prep Q&A doc.	Full Team Meeting - demo prep			
					All groups finish demo docs	
29	**30**	**31**	1	2	3	4
	Finish overall video					
Subgroup demo videos.	Send demo docs to TA					

Figure 15. Preview of March calendar of deadlines.

APRIL						
Sunday	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday
29	30	31	1 WebEx Demo 1	2	3	4 Finish report feedback updates
5	6 Full Team Meeting - checkpoint Implement demo feedback.	7	8	9 Finish writing your piece	10 Report assembled & submitted	11
12 Third report - p1 due	13 All groups prep for demo	14	15	16	17	18
19	20	21	22	23	24	25
26	27	28	29	30	1	2
	---------- Second demo ----------					

Figure 16. Preview of April calendar of deadlines.

MAY						
Sunday	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday
26	27	28	29	30 Finish writing your piece	1 Report assembled & submitted	2
3 Third report - full due Reflective Essay due	4	5	6	7 Electronic project archive due	8	9
10	11	12	13	14	15	16

Figure 17. Preview of May calendar of deadlines.

The same spreadsheet may also be viewed here: https://tinyurl.com/team2calendar.

7d) Breakdown of Responsibilities

Together, our team agreed on the overall project business plan and shared infrastructure. Samantha M. and Atmika researched and setup the shared infrastructure to lay a strong foundation for the team's sub-projects. Our team of nine members are divided into three groups of three. Each group is mainly responsible for a single mini project with specific functional features and qualitative properties, listed below. Each group sub-topic encompasses several use cases. Of course, there may be inter-group collaboration if a member's specific expertise is ever required. Some members may also take on additional tasks as needed (see below).

Note – while each teammate is responsible for different project deliverables (both coding use cases and written pieces), care is taken to ensure that everyone does an equal amount of coding and writing.

<u>Shared Infrastructure</u> – Samantha Moy, Atmika Ponnusamy

- Rutgers ECS server
- MySQL database with 7 tables
    - Customer Info
    - Parking Spots
    - Reservations
    - Walk-Ins
    - Records
    - Payment
    - Points
- 3 sub-projects will interact with the database (see the following)

<u>Sub-Groups</u>

1. **Customer Services** – Samantha Moy, Atmika Ponnusamy, Shreya Patel

   Technologies Used: Android Studio

    a. UC-1: Create Account
    b. UC-2: Edit Account
    c. UC-3: Delete Account

d. UC-4: Create Reservation
e. UC-5: Edit Reservation
f. UC-6: Cancel Reservation

2. **Arrival & Departure Operations** – Andrew Ko, Parth Patel, Piotr Zakrevski

   Technologies Used: JavaFX, Scene Builder, JDBC

   a. UC-7: Enter Garage
   b. UC-8: Exit Garage
   c. UC-9: Update Spot Status

3. **Manager Tasks** – Kylie Chow, Samantha Cheng, Nandita Shenoy;

   *Andrew Ko, Parth Patel, Piotr Zakrevski ***

   Technologies Used: Visual Studio, MEAN Stack

   a. UC-3: Delete Account
   b. UC-5: Edit Reservation
   c. UC-6: Cancel Reservation
   d. UC-10: Display Statistics
   e. UC-11: Points Management
   f. UC-12: Payment

***Note – when Group 2 finishes, they will transition into Group 3 due to the smaller size of the Group 2 project.*

<u>Additional Tasks</u>

1. Project Management – Samantha Moy
   a. Manage group deadlines and weekly tasks (set by professor and by the group).
   b. Update group calendars and documents.
   c. Run team meetings and write agendas.
   d. Assemble and submit weekly reports/deliverables.
   e. Coordinate integration of sub-group work.
2. Configuration Management – Atmika Ponnusamy
   a. Manage GitHub account.
   b. Ensure the team maintains good project practices when integrating deliverables.

c. Educate team on best coding practices and project infrastructure.
d. Troubleshoot any issues with group infrastructure (server, database, GitHub).
e. Perform integration testing,

Section 8: REFERENCES

[1] Inrix. (2017, July 12). *Searching for Parking Costs Americans $73 Billion a Year*. Inrix Research. https://inrix.com/press-releases/parking-pain-us/

[2] Chen, C., Deng, C., Feng, X., Liao, S., Musale, S., Sakhuja., R. (2018, December 9). *Auto Park Parking Garage Automation*. Software Engineering Project: Parking Garage/Lot Automation. https://www.ece.rutgers.edu/~marsic/books/SE/projects/ParkingLot/2018f-g3-report3.pdf

[3] Choudhury, S., Ngo, T., Nguyen, D., Nguyen, K., Patel, N., Tran, L. (2019, December). *Blockchain and Docker Assisted Secure Automated Parking Garage System.* Software Engineering Project: Parking Garage/Lot Automation. https://www.ece.rutgers.edu/~marsic/books/SE/projects/ParkingLot/2019f-g4-report3.pdf

[4] Gallego, Guillermo, and Garrett Van Ryzin. "Optimal dynamic pricing of inventories with stochastic demand over finite horizons." *Management science* 40.8 (1994): 999-1020.

[5] Tian, Qiong, et al. "Dynamic pricing for reservation-based parking system: A revenue management method." *Transport Policy* 71 (2018): 36-44.