Park Me

Final Report

Team 3:

Pink Unicorns 

Members:

Nolan Roe

Robert Fuller

Hugo Tavares

Adrian David

Cecilia Dominguez

Instructor:

Professor Daryl Posnett

URL: parkmecsus.wordpress.com/

Breakdown of Responsibilities

All team members contributed equally.

Table of Contents

Summary of Changes

Revisions to Report one:

- Removed and condensed some of the Use Cases
- Updated the Traceability table due to Use Case revisions
- Added more content in the Customer Statement of Requirements
- Reduced and simplified the Actors and Goals section
- Simplified the System Operation of Contacts
- Added more terms in the Glossary

Revisions to Report two:

- Updated the user interface design and Implementation

Customer Statement of Requirements

Our main goal is to enable students to easily find a parking spot with ease. We will provide students with an app to easily tell whether parking spots are vacant or non-vacant and provide a navigation system. We will leverage location tracking and routing utilizing mobile device based technologies for audio interactions and available mapping technologies.

Parking at Sacramento State is a hassle. After arriving at the school, one can expect spending 5-15 minutes locating a vacant spot to park. At peak times, this time can become even greater. And at the beginning of the semester when people are trying to add classes, the parking situation can get so bad that students get so frustrated that they turn around and go home because they can't find a vacant spot. According to source, students that don't make it to class on time on average have GPA's that are # points lower than their punctual

counterparts and students that don't make it to class have GPA's that are # lower. This is a huge problem and needs to be resolved.

Currently, in an effort to ease the strain on the current parking lot system at Sac State, they have decided to build an entirely new 5 story parking structure that will cost around $25 million to complete. There is also an app on the Sac State App that provides "real time" parking availability that has proven to be completely worthless and does not provide good enough information to be useful. And finally, there is temporary road work sign that is used at the North entrance of the school to display which lots are full and recently they've actually removed the sign. Clearly this situation is a mess. The system is disorganized and instead of spending $25 million to build new structures, we believe that more connected and responsive parking lots will be able to help parking enforcement and students come to a more logistic friendly resolution when it comes to parking.

The current system uses manually inputted data entered by parking enforcement to display a percentage of how full each lot is. This leaves a lot of room for human error and laziness as lot info has been shown on the Sac State App to go unedited for hours. What we are proposing is coming up with a real-time system to not just monitor the percentage of how full each lot is, but with a preciseness of exactly which spots are still vacant. This information can then be passed on to parking enforcement and students alike to guide students to vacant spots.

Each parking spot will be equipped with a module that can be used to detect whether a vehicle is present at the spot with 99.99% percent accuracy. The module will use a HC-SR04 Ultrasonic sensor and will connect to a microcontroller. The microcontroller will be able to analyze the data coming into the sensor and be able to determine if a car is present or not. The microcontroller will then send information to a remote web server where the data will be stored. This data can then be used by parking enforcement and students/faculty alike to tell where parking is available.

The average student/faculty user will be able to use the app on their own to find a spot. The home screen of the app will display a heat map of Sacramento State and all the parking lots/structured on campus. From there a user can do a sort, for example a faculty member can limit their query to faculty only lots. Or if they'd like they can click on a parking lot/structure to get a view in real time of where parking spots are available within that parking lot/structure. Once they have a map of what spots are open, all that's left is to drive to that spot and park there. Once they park in a vacant spot, the sensor gets triggered at that spot and the app updates, making the spot appear as "not vacant" on app.

Another scenario is student/faculty may want to park in closest proximity possible to a desired building. Actual commute time on foot is another major consideration that needs to be addressed when designing a system used to optimize logistics. There might be a sizable amount of parking available at lot 7 but let's say the user's first class is at 9am at the opposite end of campus in Solano Hall. A better place to park would be Lot 1. There's a 0.5 mile difference and a 8 minute difference in walking time from lot to classroom! What a user would be able to do is pick what classroom they want to be closest to. The app will then return a list of spots it recommends parking at based on availability and odds that a parking spot will still be available by the time the user gets the parking lot/structure.

Sometimes the app itself can be very difficult to navigate through. The user opens the app, finds it confusing, and out of frustration may decide not to use it ever again. That's why our app will be very user friendly. It will mainly focus on finding a vacant parking spot with a few other options that the user may wish to use, such as the sorting feature mentioned earlier. The user can easily choose whether to toggle these other options on or off. The app will be able to find a parking spot with just a few taps after first being set up. Say the user is in a hurry; the user will be able to open the app and within one to three taps, the app has already found a vacant parking spot for the user.

The database will have outbound communication with the cell phone app along with a possible desktop application. Using a homegrown proprietary REST API in conjunction with the SQL database, the app and web interface alike will have the capability of data extraction through JSON. An example implementation of the getLot method on the REST API would be to display the lot info through JSON. From there, the app and web interface alike can parse the data and generate kml overlays to be used on the google maps map. Possible additional implementation would be to have the web server generate these kml files, but the speed practicality of this method is yet to be determined.

The system as a whole will largely be event based. The database becomes updated once it receives stimulus of a change in state from the sensors via the internet. From there, signs and displays will be updated based on the new overall state of the database. The web interface and mobile phone app will also be updated based on these events but may be updated incrementally to save bandwidth. The mobile app and interface are also event based systems in their own respect. A click to see the detailed state of a lot is an event trigger by the user. A request to receive directions to a certain spot is also an event created by the user.

Glossary of Terms

Technical Terms:

App – A mobile application where customers can access the system to find and navigate a parking spot.

Database – Entity that stores all the system's information.

Espresso – A testing framework for Android used to test the application interface.

Geo-coordinates - number based coordinate system used in geography that enables every location on Earth.

Google Maps - Google Maps is a web mapping service developed by Google integrated to find the current location of the user.

Raspberry PI - microcomputer used to communicate the sensor information with the database.

Rest API - web services are a way of providing interoperability between computer systems on the Internet.

Sensors – A device that is placed on the floor of every parking spot that detects if that spot is occupied or vacant.

Server - computer program that manages access to a centralized resource or service in a network.

System - a set of connected things or parts forming a complex whole

Use case - is a list of actions or event steps typically defining the interactions between a role and a system to achieve a goal.

Website – An interface that the customer can use to register, or see more information about the application.

Non-Technical Terms:

Driver – The primary consumer/client in the system, who is in need for parking space.

Enforcer – an employee/admin that maintains the lot/garage.

Garage - a building or shed for housing a motor vehicle or vehicles.

Hands Free - navigation directions are provided through the application.

Navigate – plan and direct the route or course of a vehicle by using a mobile device.

Vacant - refers to parking spaces available for use.

Vehicle – A thing that is used to transport people (I.E: car).

Functional Requirements Specifications

Priority Description of Requirements

Identifier	P.W.	Description
REQ 1	**3**	System locates user's current location to optimize search.
REQ 2	**2**	System provides user with real time data showing what parking lots are the fullest.
REQ 3	**5**	System shall provide user with lot options to choose from.
REQ 4	**5**	System accesses the database to locate vacant parking spaces.
REQ 5	**4**	System provides user option to navigate to the parking space.
REQ 6	**3**	System is able to reroute the user to a different location if the spot is taken by someone else.
REQ 7	**1**	System can provide user with data demonstrating expected flow derived from date, time, and lot.
REQ 8	**5**	System shall record all occupied and vacant spots to the database on real time.

Stakeholders

The following people are the ones who might benefit and get involved with the app:

-Students

-Faculty staff/members

-Parking enforcers

-Database manager

Actors and Goals

Driver (Initiating Actor)

- A Driver can be a university student, faculty member, or campus staff. Drivers are the primary consumers of the system. A Driver's goal is to find the nearest available parking space based on their approach to campus or destination on campus.

Parking enforcer (Initiating Actor)

- An enforcer can be a parking enforcement employee or an admin. They maintain and scout the garage for safety and checks the validity of the permit in every vehicle. They can also manually override a certain spot/s if they need to close it for maintenance or reservations.

Sensor (Participating Actor)

- A Sensor provides the self-report technology the system relies on to determine the occupancy status of an individual space in a parking lot. The Sensor provides real-time status updates of either occupied or vacant.

Navigator (Participating Actor)

- The Navigator guides a driver to a vacant parking spot. The Navigator provides several important services: first, it tracks the current location of a driver; second, it provides a route to a parking lot; third, it provides audio directions to the driver.

SpaceFinder (Participating Actor)

- The SpaceFinder provides the system with real-time tabulation of the total number of vacant spaces in each parking lot on campus, and locates the best parking lot and available space.

Use Cases with their Casual Descriptions

Use Case 1 - ParkMe Valet:

- Automates trip from start to finish.

Use Case 2 - ParkMe Near:

- Enables the user park nearby a specific destination.

Use Case 3 - Manual Override:

- Allows a parking enforcer to change the status of a parking spot by manually closing and opening it through the system.

Fully Dressed Use Case Descriptions

Use Case 1 - ParkMe Valet

Related Requirements: REQ 1, 2, 4, 5, 6, 8

Initiating Actor: Driver

Goal: User wants to find closest possible parking spot from their current location.

Participating Actors: Sensor, SpaceFinder, Navigator

Preconditions: A parking spot must be available, and all services must be up and running.

Postconditions: User is given directions on how to get to the closest and optimal parking spot.

Main Success Scenario:

1. User opens app with location permissions enabled
2. Google Maps is able to locate phone
3. User must select ParkMe Valet option
4. Database returns all possible parking places
5. Google Maps chooses closest and optimal spot
6. Google Maps uses navigation to direct user to the parking place

Extensions: Spot gets taken by another vehicle before user has arrived at desired parking spot.

Use Case 2 - ParkMe Near

Related Requirements: REQ 1, 2, 3, 4, 5, 6, 7, 8

Initiating Actor: Driver

Goal: Lets the user select a certain lot and park nearby a specific destination.

Participating Actors: Sensor, SpaceFinder, Navigator

Preconditions: A parking spot must be available, and a certain spot must be selected.

Postconditions: User is given directions to their selected lot and search for the closest and optimal parking spot.

Main Success Scenario:

1. User opens app with location permissions enabled
2. Google Maps is able to locate phone
3. User must select ParkMe Near option
4. Database returns all possible parking places
5. User must select the desired lot
6. Google Maps chooses closest and optimal spot
7. Google Maps uses navigation to direct user to the parking lot and the spot afterwards

Extensions: Spot gets taken by another vehicle before user has arrived at desired parking spot.

Use Case 3 – Manual Override

Related Requirements: REQ 2,3,4,7,8

Initiating Actor: Parking enforcer

Goal: User has the access to manually change the status of a certain spot.

Participating Actors: Sensor

Preconditions: Parking lot must be vacant.

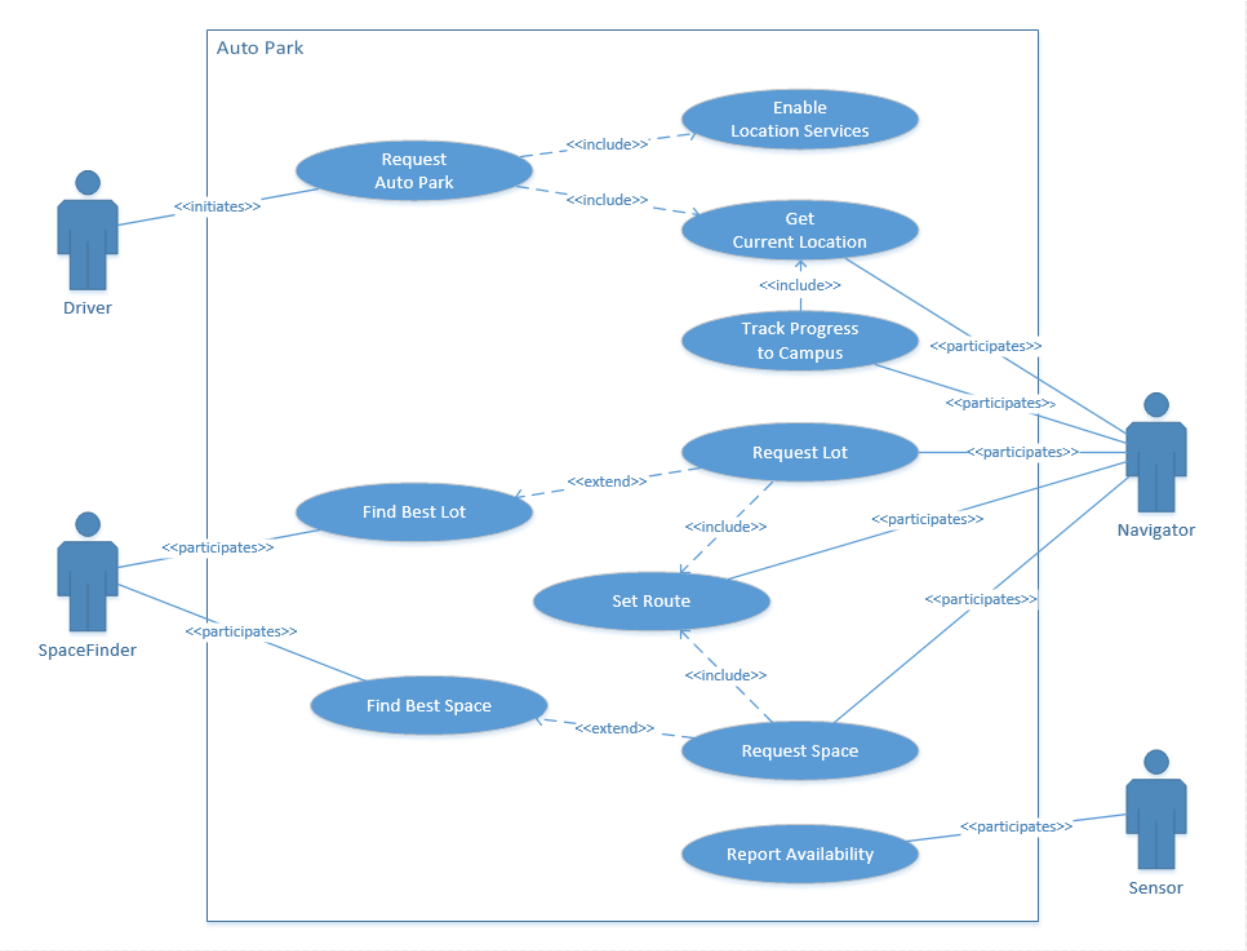
Postconditions: A parking lot will be closed/open.

Main Success Scenario:

1. Parking Enforcer will choose a certain spot or spots.
2. Parking Enforcer can manually change the status of a spot if it needs to be down for maintenance or reservation.
3. The Sensor will display green if it's open, and red if it's closed.

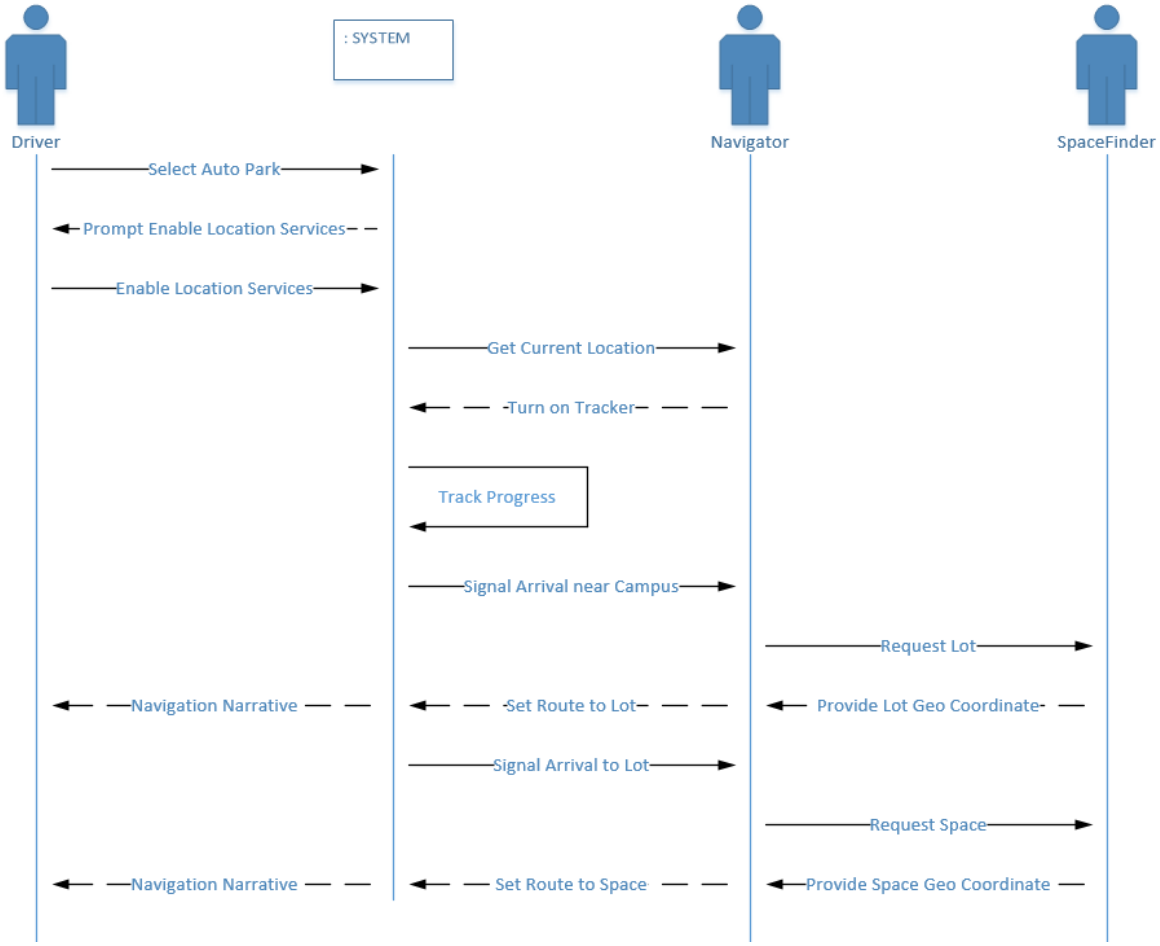
Use Case Diagrams

ParkMe Valet automatically routes the driver to the best parking space. The driver simply starts the app and relies on driving instructions provided by the Navigator, while ParkMe Near to my destination lets the driver select a building on campus so that it can prioritize parking lots in its vicinity.

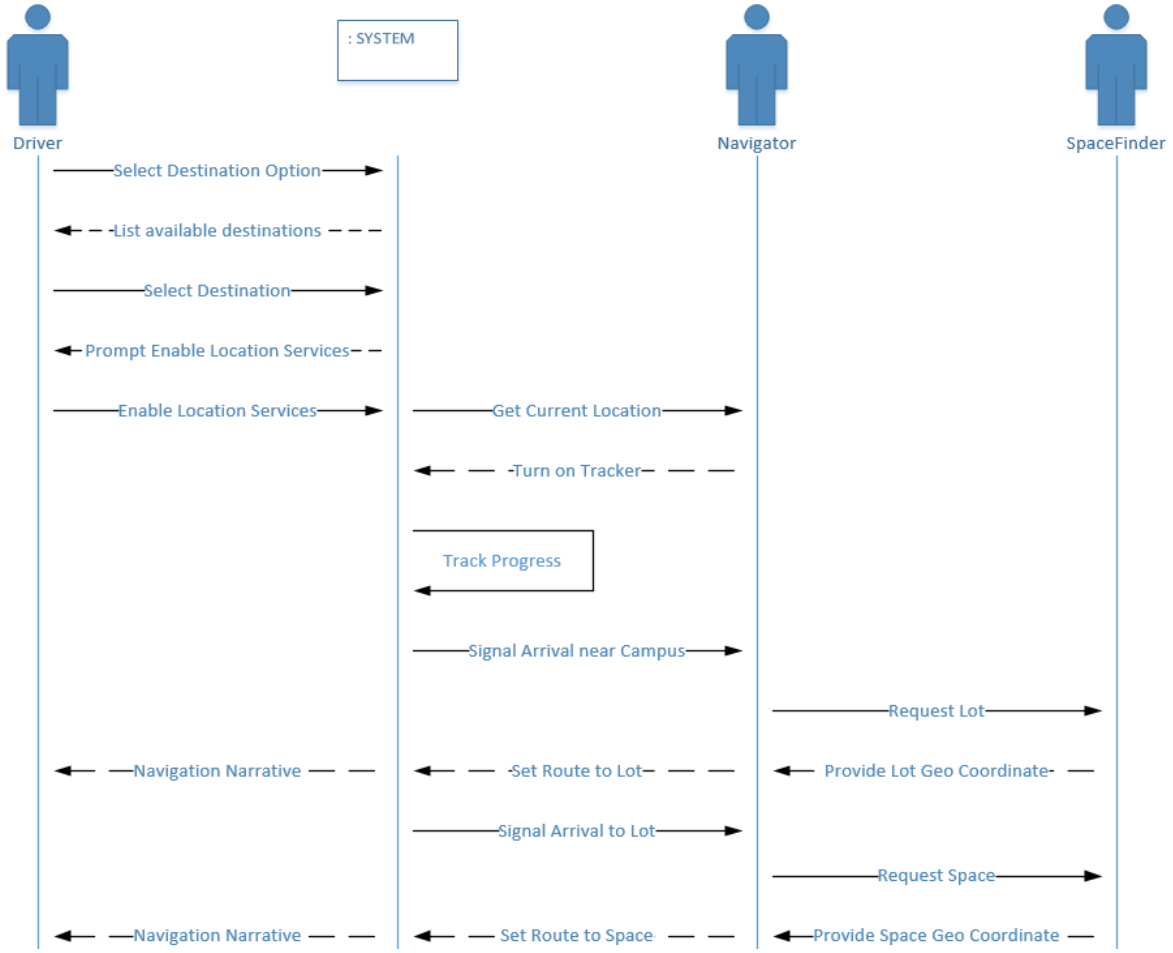


System Sequence Diagram

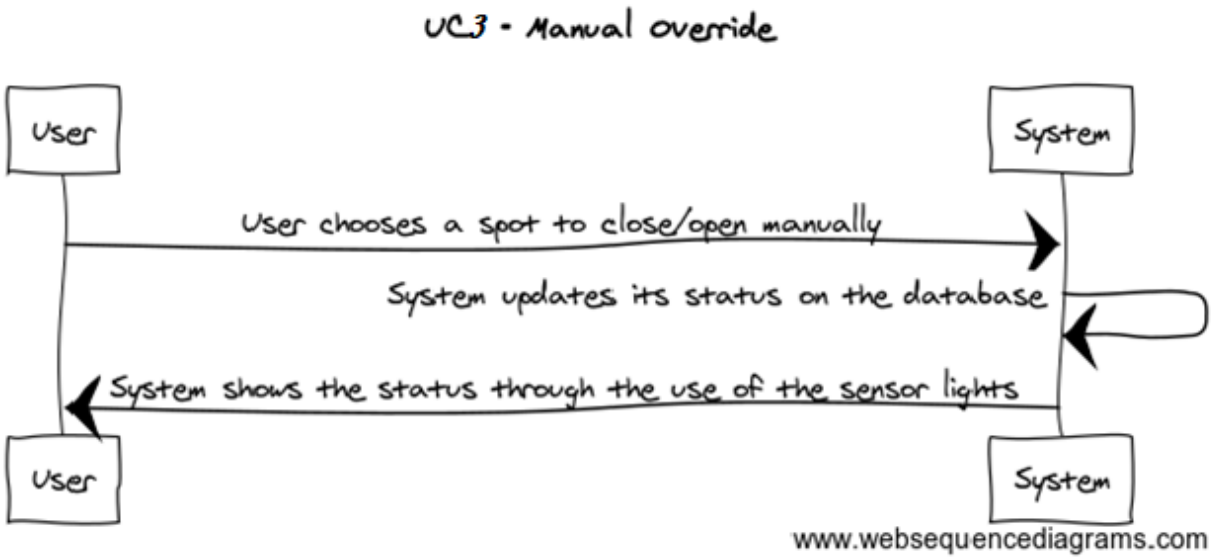
UC 1 – ParkMe Valet



Use Case 2 – ParkMe Near



Use Case 3 – Manual Override



Traceability Table

Functional Requirements	Priority Weight	UC 1	UC 2	UC 3
REQ 1	3	X	X	
REQ 2	2	X	X	X
REQ 3	5		X	X
REQ 4	5	X	X	X
REQ 5	4	X	X	
REQ 6	3	X	X	
REQ 7	1		X	X
REQ 8	5	X	X	X

Non-Functional Requirements

Identifier	P.W.	Description
REQ 9	**4**	System can be tested to check such that it works properly and that there are no bugs.
REQ 10	**4**	The system allows for a hands free experience through navigation.
REQ 11	**4**	The database is stored on site to decrease the chance of data loss.
REQ 12	**3**	Web pages will have a simple design to enhance user experience. Each page will have a similar template to prevent confusion.
REQ 13	**2**	System shall keep track of the preferred lots previously selected.
REQ 14	**5**	System shall immediately end the navigation if the user parked somewhere else with a sensor
REQ 15	**3**	System will immediately end the navigation with one press of a button.
REQ 16	**5**	System shall navigate the user to the parking spot with precision and accuracy.
REQ 17	**5**	The system asks the user if they wish to navigate to the nearest parking lot.
REQ 18	**4**	The system asks user to choose a vacant parking space from the list. The list should be ordered from nearest to farthest vacant parking space.

Effort Estimation using Use Case Points

ParkMe will be straightforward, two to three taps are all we need to do, and it will navigate the user to an empty spot. Utilizing the location service provided by a mobile device,

ParkMe tracks the driver's progress toward campus, and when s/he is close to campus ParkMe checks with the SpaceFinder to find the closest parking lot with the highest availability. Once the driver enters the parking lot the Navigator requests the geo coordinates for and available space and automatically routes them to spot. The key difference between ParkMe Near and ParkMe Valet is that the driver can have the system prioritize parking lots close to his/her destination.

Manual Override's function gives the parking enforcer the ability to open and close spots for other purposes. The user must make sure that the parking spot is vacant before they can manually change the status of the spot.

Due to the few use cases presented in the report, the use case points won't be as high as we will be expecting, but since the ParkMe Near and ParkMe Valet functions has a lot of complexity, the final use case points will be assumed to be higher than the unadjusted use case weights.

Technical Complexity

Technical Factor	Description	Weight	Perceived Complexity	Calculated Factor (Weight * Perceived Complexity)
T1	Distributed system	2	2	4
T2	Performance objectives	1	3	3
T3	End-user efficiency	1	3	3
T4	Complex internal processing	1	4	4
T5	Reusable design or code	1	2	2
T6	Easy to install	0.5	3	1.5
T7	Easy to use	0.5	4	2
T8	Portable	2	5	10
T9	Easy to change	1	3	3
T10	Concurrent use	1	2	2
T11	Special security features	1	3	3

T12	Provides direct access for third parties	1	3	3
T13	Special user training facilities are required	1	0	0
	Technical Factor Total			**40.5**

TCF = Const-1 + Const-2 × Technical Factor Total = **0.6 + (0.01)(40.5) = 1.005**

$$C_1 + C_2 \cdot \sum_{i=1}^{13} W_i \cdot F_i$$

Use Case Weights

Use Case	Description	Category	Weight
1 ParkMe Valet	Complex User Interface. 6+1 steps for main success scenario. Three participating Actors (Sensor, Spacefinder, Navigator)	Complex	**15**
2 ParkMe Near	Complex User Interface. 7+1 steps for main success scenario. Three participating Actors (Sensor, Spacefinder, Navigator)	Complex	**15**
3 Manual Override	Simple User Interface. 3 steps for main success scenario. One participating Actors (Sensor)	Simple	**5**

UUCW = 1 × Simple + 0 × Average + 2 × Complex = **1×5 + 0×10 + 2×15 = 35**

Environmental Factors

Environmental Factor	Description	Weight	Perceived Complexity	Calculated Factor (Weight * Perceived Complexity)
E1	Beginner familiarity with the UML-base development	1.5	1	1.5

E2	Some familiarity with application problem	0.5	2	1
E3	Some knowledge of object oriented approach	1	2	2
E4	Lead analyst capability	.05	2	1
E5	Motivated Team Member	1	3	3
E6	Stable Requirements Expected	2	4	8
E7	Part Time staff	-1	3	-3
E8	Difficult Program Language	-1	3	-3
	Environmental Factor Total			**10.5**

ECF = Const-1 + Const-2 × Technical Factor Total = **1.4 + (-0.03)(10.5) = 1.085**

$$C_1 + C_2 \cdot \sum_{i=1}^{8} W_i \cdot F_i$$

$$UCP = UUCP \times TCF \times ECF$$

From the above calculations, the UCP variables have the following values:

Unadjusted UCP: UUCP = 35

Technical Complexity Factors: TCF = 1.005

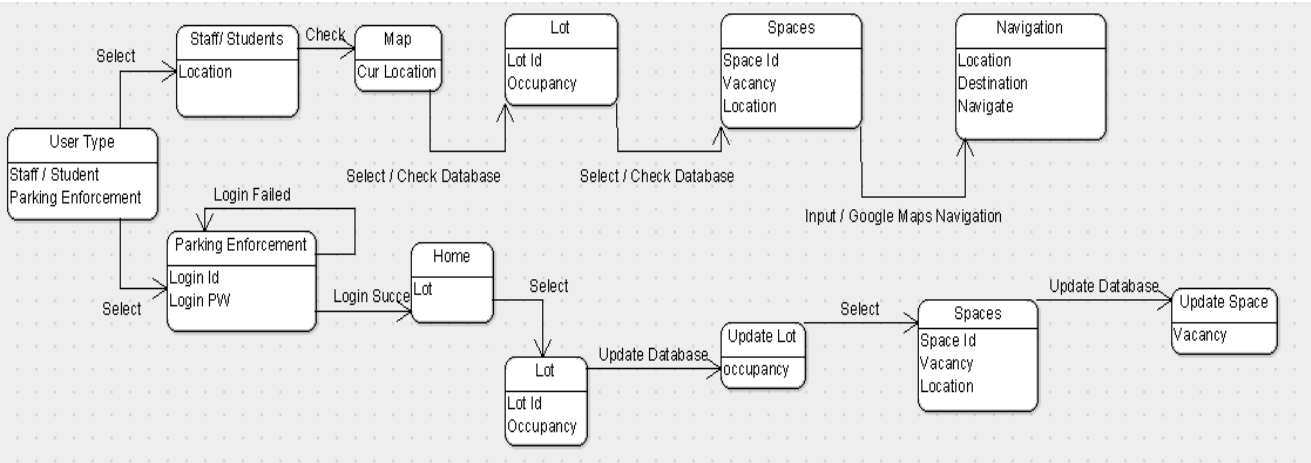
Environmental Complexity Factors: ECF = 1.085

For the ParkMe case study, the final UCP is the following:

UCP = 35× 1.005 × 1.085 = <u>38.16 or 38 use case points.</u>

Domain Analysis

Domain Model



System Operation Contacts

Operation	ParkMe Valet
Preconditions	- A parking spot must be available - A certain spot must be selected.
Postconditions	- User is given directions on how to get to the closest and optimal parking spot.

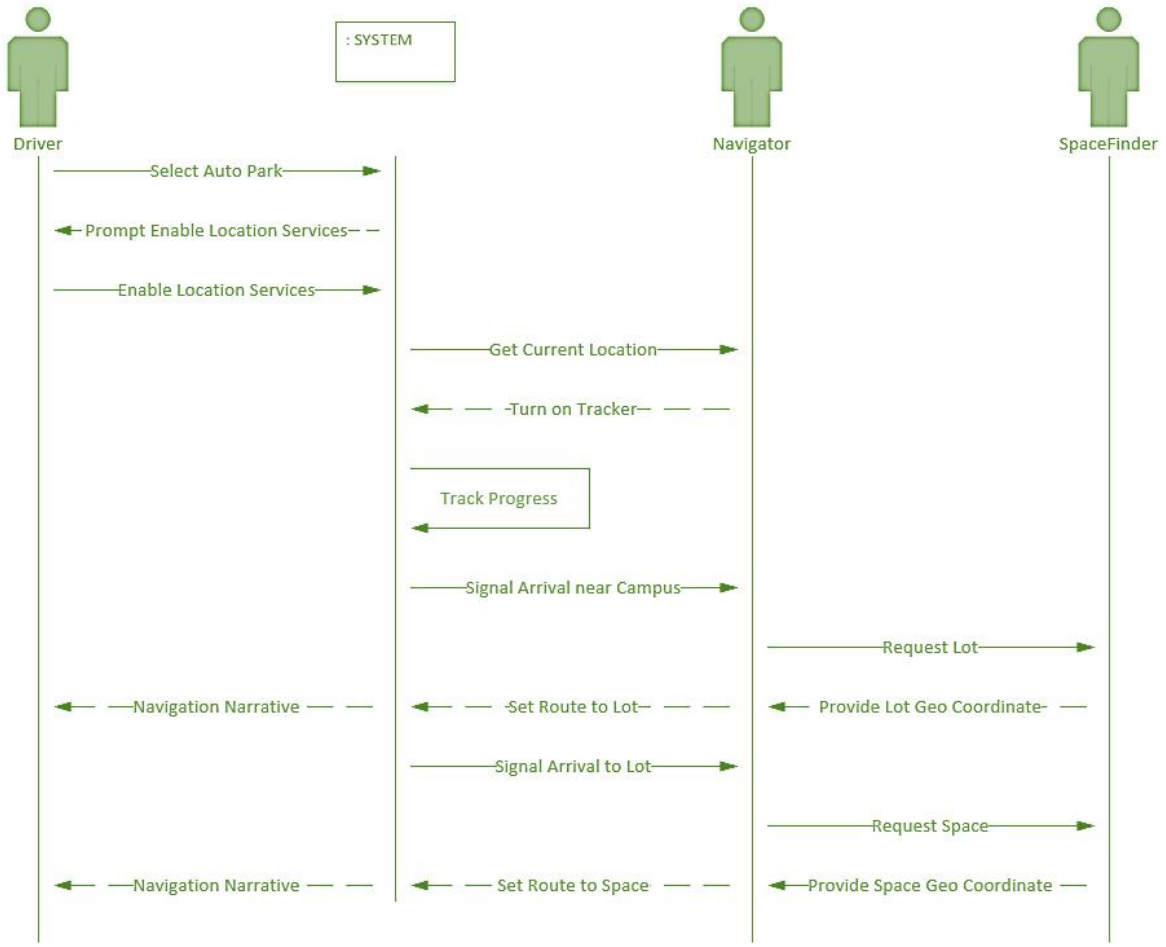
Operation	ParkMe Near
Preconditions	- A parking spot must be available - A certain spot must be selected.
Postconditions	- User is given directions to their selected lot and search for the closest and optimal parking spot.

Operation	Manual Override
Preconditions	- Parking lot must be vacant.

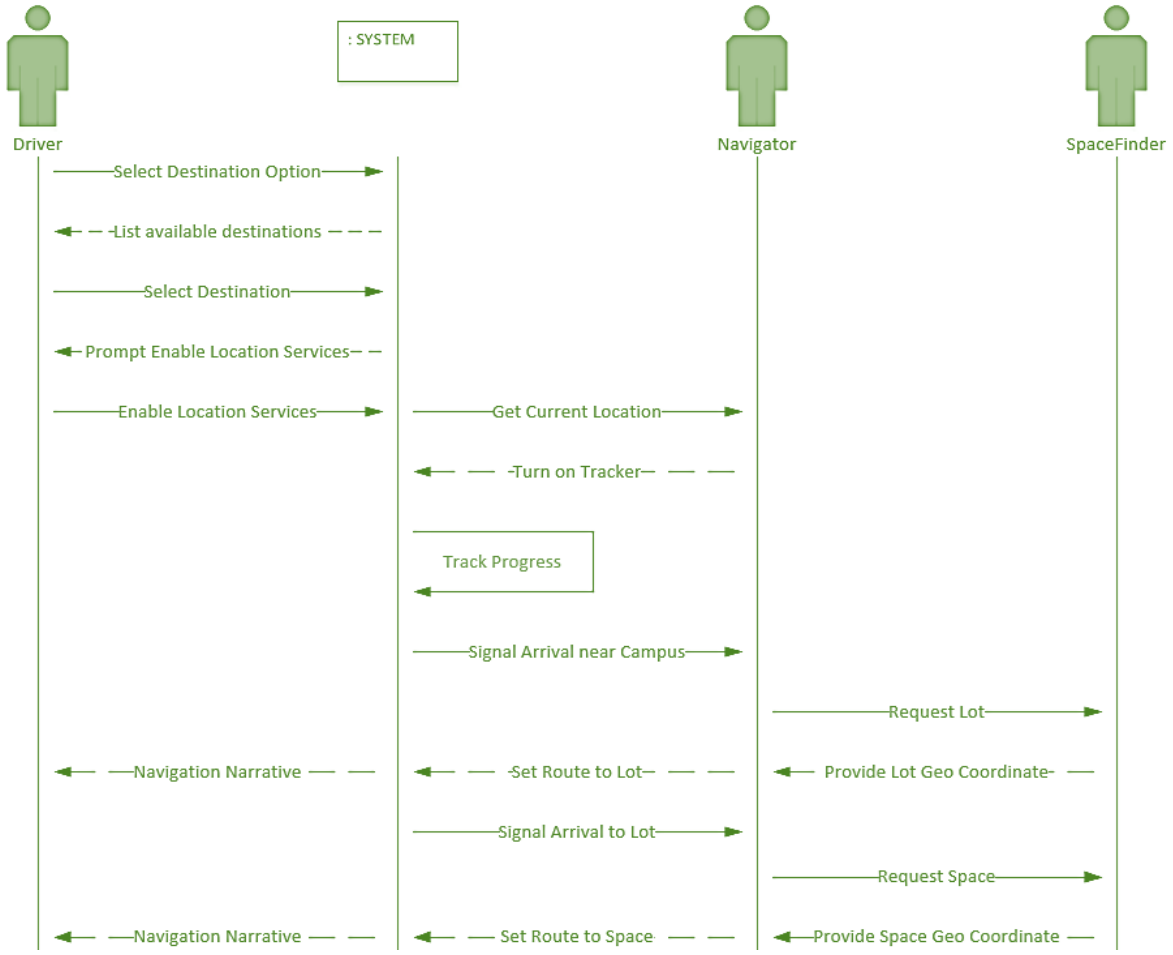
Postconditions	- A parking lot will be closed/open.

Interaction Diagram

ParkMe Valet



ParkMe Near

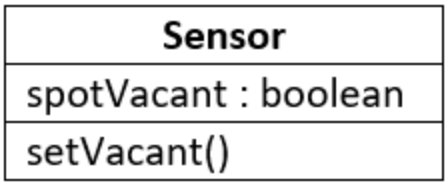


Class Diagram and Interface Specification

The diagrams below provide an overview of the key classes used in our system.

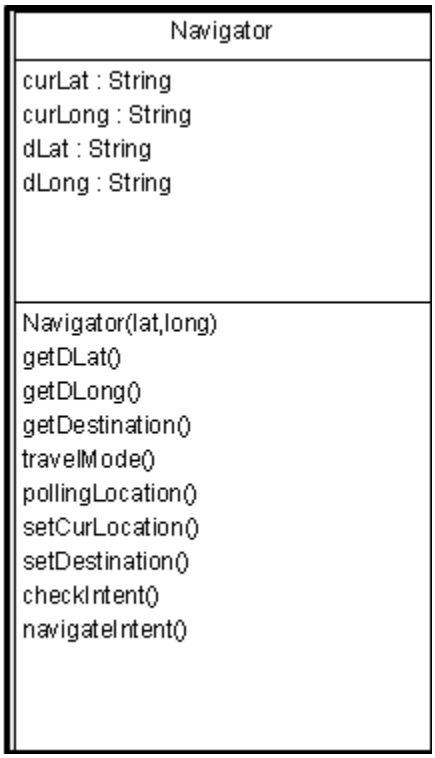
Sensor Class Diagrams

The sensor class utilizes a Raspberry Pi microcomputer in conjunction with a supersonic sensor which is used to detect the physical presence of a vehicle in a parking spot. When the supersonic sensor detects a vehicle, it sends a signal to the raspberry pi. From there, the pi is able to send information to the server on the status of the parking spot.



Navigator Class Diagrams

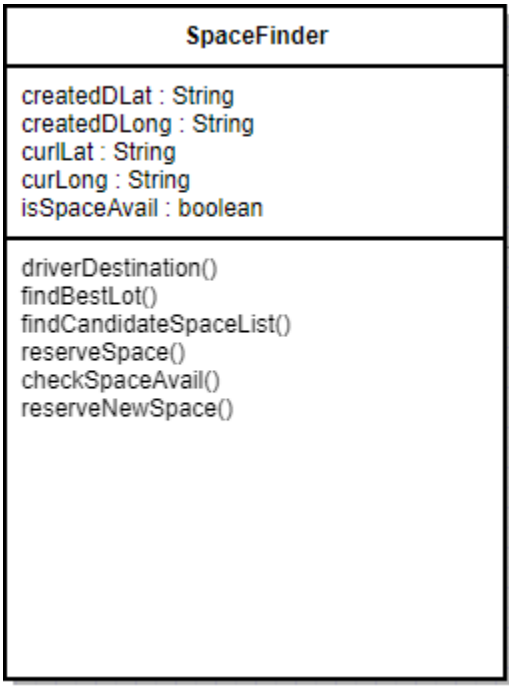
The Navigator class uses the users current Geo-location coordinates and the destination coordinates to setup a route. The destination Geo coordinates are set to the campus if the user is more than a mile away. Once entering a parking lot the polling function will invoke the SpaceFinder class to find new Geo coordinates and set up a route. The route is determined by the latitude and longitude values received back from SpaceFinder. Polling will continue to take place every 10 seconds and update routes. Once a route has been found the Google Maps Directions API will take over. Proceeding updates will be managed by the Google Maps Direction API.



SpaceFinder Class Diagrams

ParkMe Valet mode: The SpaceFinder class tries to find the best and closest parking spot from the current location of the user.
ParkMe Near mode: The SpaceFinder class let's the user choose where they want to park from a list of locations.

The SpaceFinder class uses the database to determine whether the parking space is vacant. If the parking space chosen by the SpaceFinder class is vacant, it will create a set of Geo location coordinates, using Google Maps API, of that parking space and hand them to the Navigator class. While the driver is on route, the SpaceFinder class will keep checking as to whether the parking spot is still vacant. If the parking space happens to fill up, the SpaceFinder class will find the closest parking space from the previous parking spot, create another set of Geo-location coordinates, and give them to the Navigator Class again.



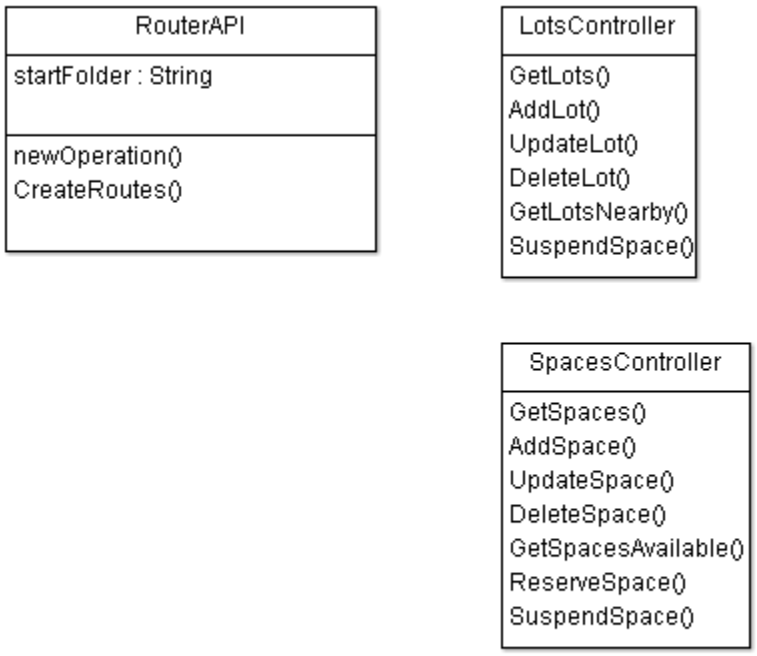
- driverDestination // Used in the ParkMe Near use case

Methods

- FindBestLot(pDriverDestination : destinationId = NULL)
- FindCandidateSpaceList(pLot : lotId)
- ReserveSpace(pSpace : spaceId) --->

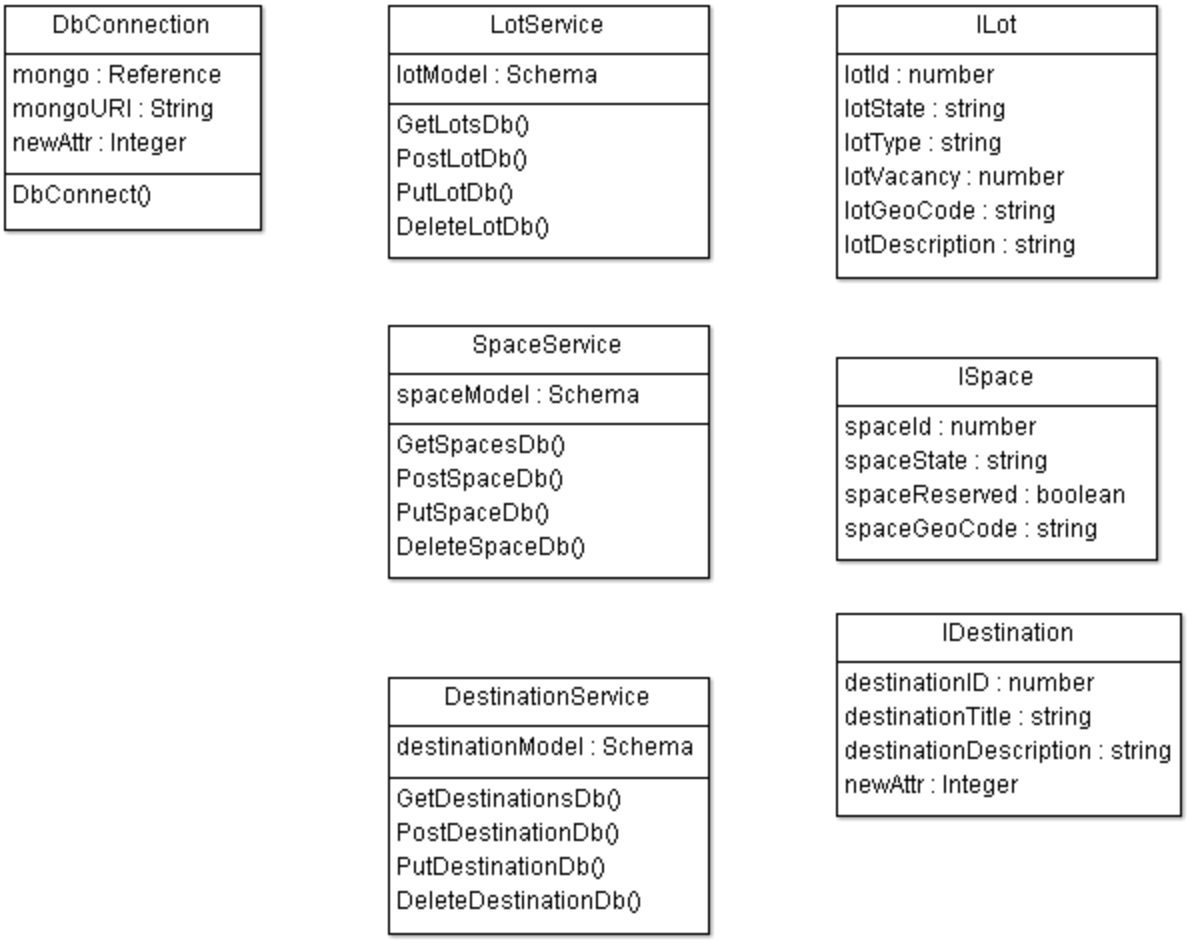
REST API Class Diagrams

The REST API relies on deep linking to build the necessary routes. The RouterAPI class dynamically builds new routes based on the current folder structure to simplify future expansion.



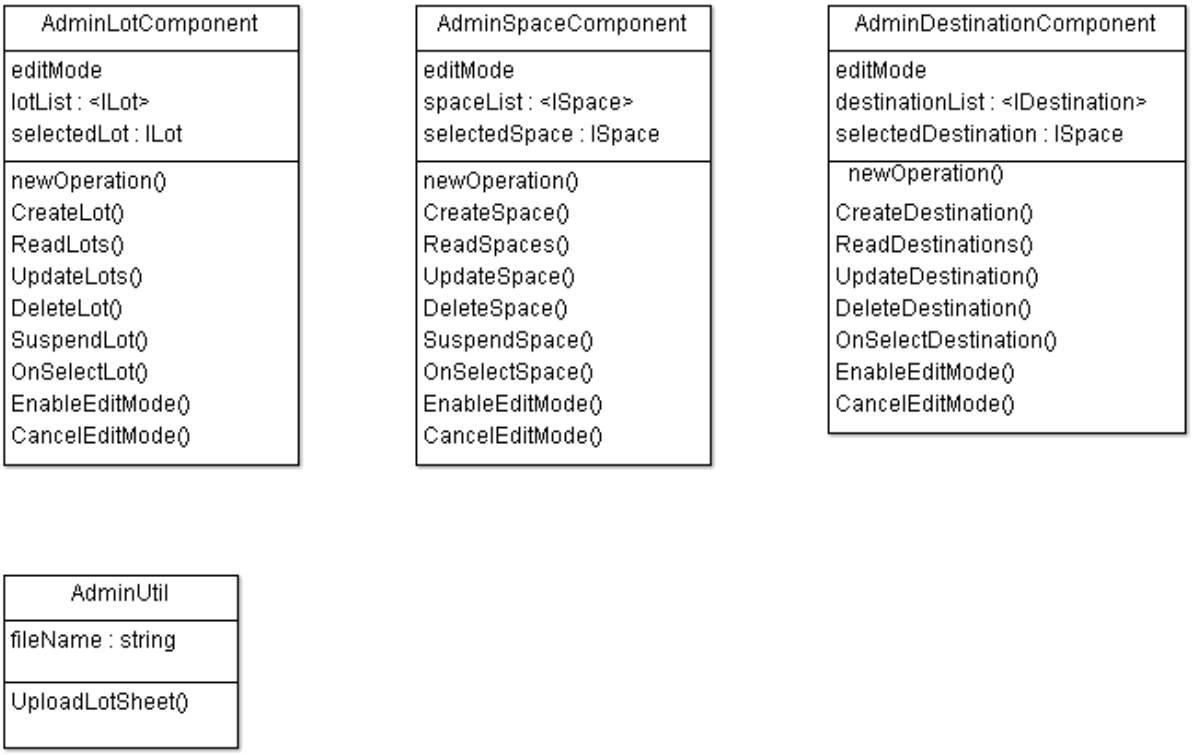
Db Access Class Diagrams

The DbAccess classes support basic CRUD operations for the parkdbcsus Cosmos DB database. In addition, some helper methods have been defined for key operations, such The Web and Android apps will be accessing the REST API through HTTP protocols since the database is hosted on the Azure Cloud.



Web Admin Class Diagrams

The WebAdmin classes are used by the system administrator to manage the parking lots with related parking spaces as well as the on campus destinations a driver can choose from in the ParkMe Near mode.



Web DriverPark Class Diagrams

The Web DriverPark classes are used by a driver to accomplish the following tasks: select the ParkMe mode (i.e., ParkMe Valet or ParkMe Near); optionally select a destination when the ParkMe Near mode is chosen; provide a visual map of the route.

Data Types and Operation Signatures

Navigator

Attributes:

curLat: String

Corresponds to the user's current latitude coordinate.

curLong: String

Corresponds to the user's current longitude coordinate.

dLat: String

Corresponds to the user's destination latitude coordinates. This value will be derived or set to relevant campus coordinate as default.

dLong: String

Corresponds to the user's destination longitude coordinates. These values will be derived or set to relevant campus coordinate as default.

Operations:

Navigator(String lat, String long)

Constructor method takes in the current Geo coordinates and stores the values.

GetDLat()

Method retrieves the users expected destination latitude coordinates.

getDLong()

Method retrieves the users expected destination longitude coordinates.

getDestination(SpaceFinder dCoordinate)

Method will use the information received from spaceFinder to set up destination coordinates (vacant space or campus) depending on the users current location.

travelMode()

Sets up the travel mode to vehicle to be used by the Google Maps Directions API when handed- off.

pollingLocation()

Method will continuously poll the SpaceFinder to update the routes to get to the vacant space.

setCurLocation()

Method sets up the Google Maps Directions API with current location.

setDestination()

Method sets up the Google Maps Directions API with desired destination Geo coordinates.

checkIntent()

Method will check to see if there is a valid intent can startActivity with an activity ready to receive the intent.

navigateIntent()

Calls Google Maps Directions API with valid new Intent and context.

SpaceFinder

Attributes:

createdDLat: String

Corresponds to the user's destination latitude coordinates.

createdDLong: String

Corresponds to the user's destination longitude coordinates.

curLat: String

Corresponds to the user's current latitude coordinate.

curLong: String

Corresponds to the user's current longitude coordinate.

isSpaceAvail : boolean

True if space is vacant false if not.

Operations:

driverDestination()

Used in ParkMe near, lets user choose from a list of buildings.

findBestLot()

Automatically finds best lot from the user's current location or from driver destination.

findCandidateSpaceList()

Automatically finds best space.

reserveSpace()

Reserves space for user and creates a set of Geo-coordinates, but does not guarantee space will not fill up.

checkSpaceAvail()

Continually checks with sensor if the Space is vacant or not.

reserveNewSpace()

If space is taken, will find a new Space and creates a new set of Geo-coordinates.

Design Patterns

Prototype Design Pattern

Intent

- Specifies the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype. This can be implemented when updating routes to vacant parking spaces.
- Co-opt one instance of a class that will be used while the application remains launched on device screen.

Object Constraint Language (OCL) Contracts

Driver

Invariant- Context driver inv: self.park

Precondition- Context driver:: (d integer) Pre:self.park=o

Postcondition- Context driver:: (d integer) Post:self.park=self.spotnum

Parking Enforcer

Invariant- Context enforcer inv: self.enforce

Precondition- Context enforcer:: (e integer) Pre:self.enforce=o

Postcondition- Context enforcer:: (e integer) Post:self.enforce=h*w

Lot

Invariant- Context lot inv: spots

Precondition- Context lot:: (l integer) Pre:self.spots=vacantspots

Postcondition- Context lot:: (e integer) Post:self.spots=self.spotnum

System Architecture and System Design

Architectural Styles

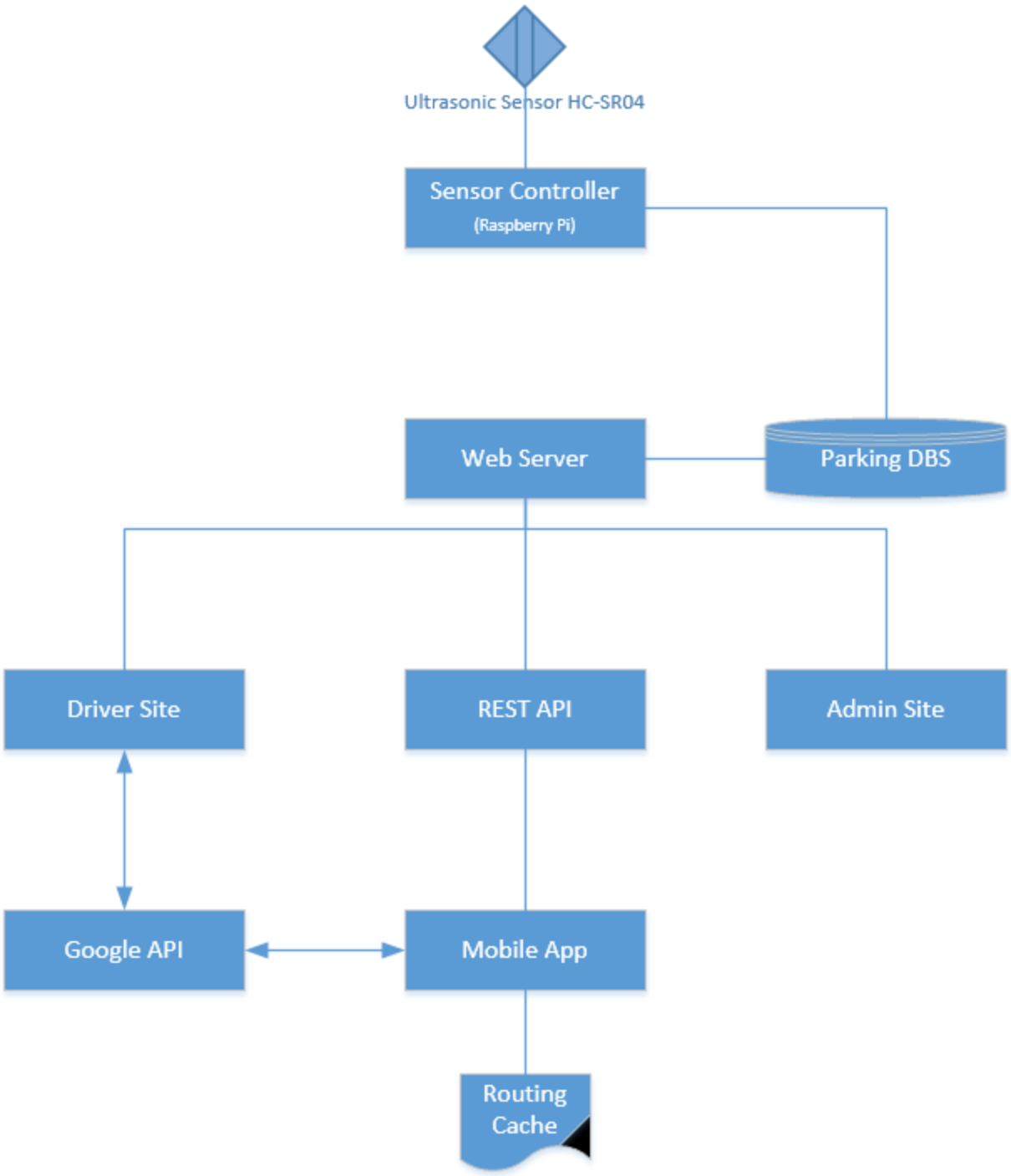
The ParkMe solution is built upon a client/server architecture and consists of three-tiers.

Clients include the following: Web-based Administration and Driver sites; Android mobile app; Sensor Controllers.

Client/Server

1. Cloud database
2. Web Server: URL based REST API
3. Web-based Administration and Driver sites; Android-based mobile Driver app

The image below provides a high level overview of the system architecture:

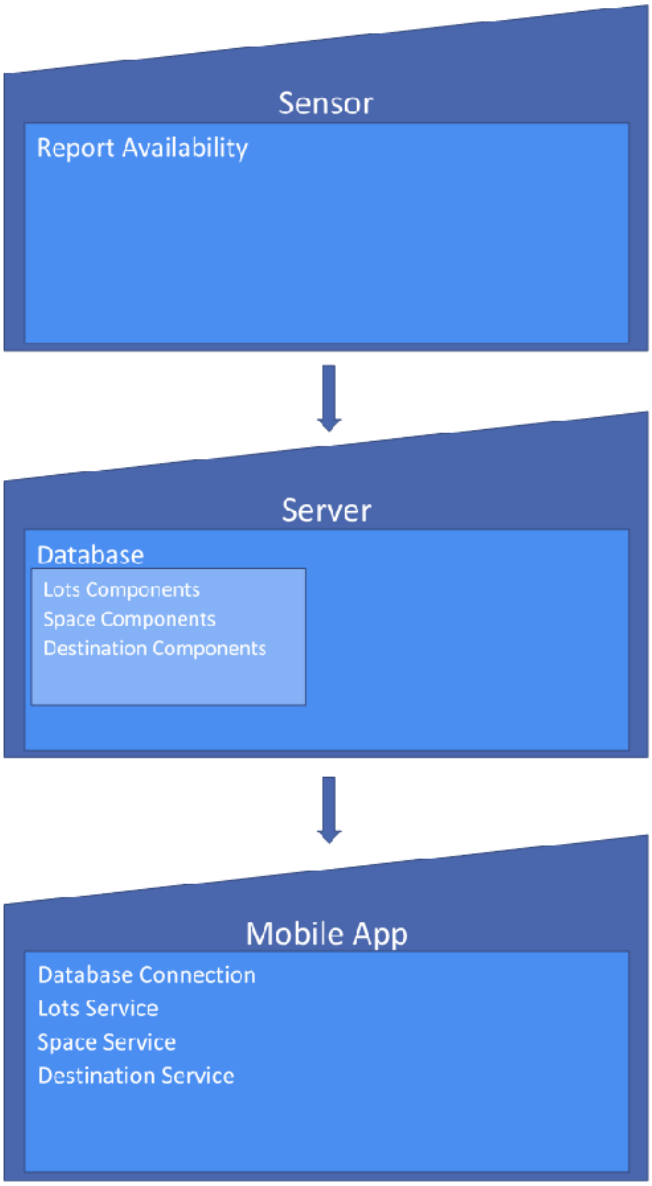


Identifying Subsystems

Everything will start on the sensor checking if a spot is available, then it will send the information to the database. The database comprises a subsystem that includes the

components of parking lots, spaces, and destination. The server then will communicate through the mobile app, which includes its subsystem on services such as get lots, spaces, and destination.

The image below identifies the subsystems:



Mapping Subsystem to Hardware

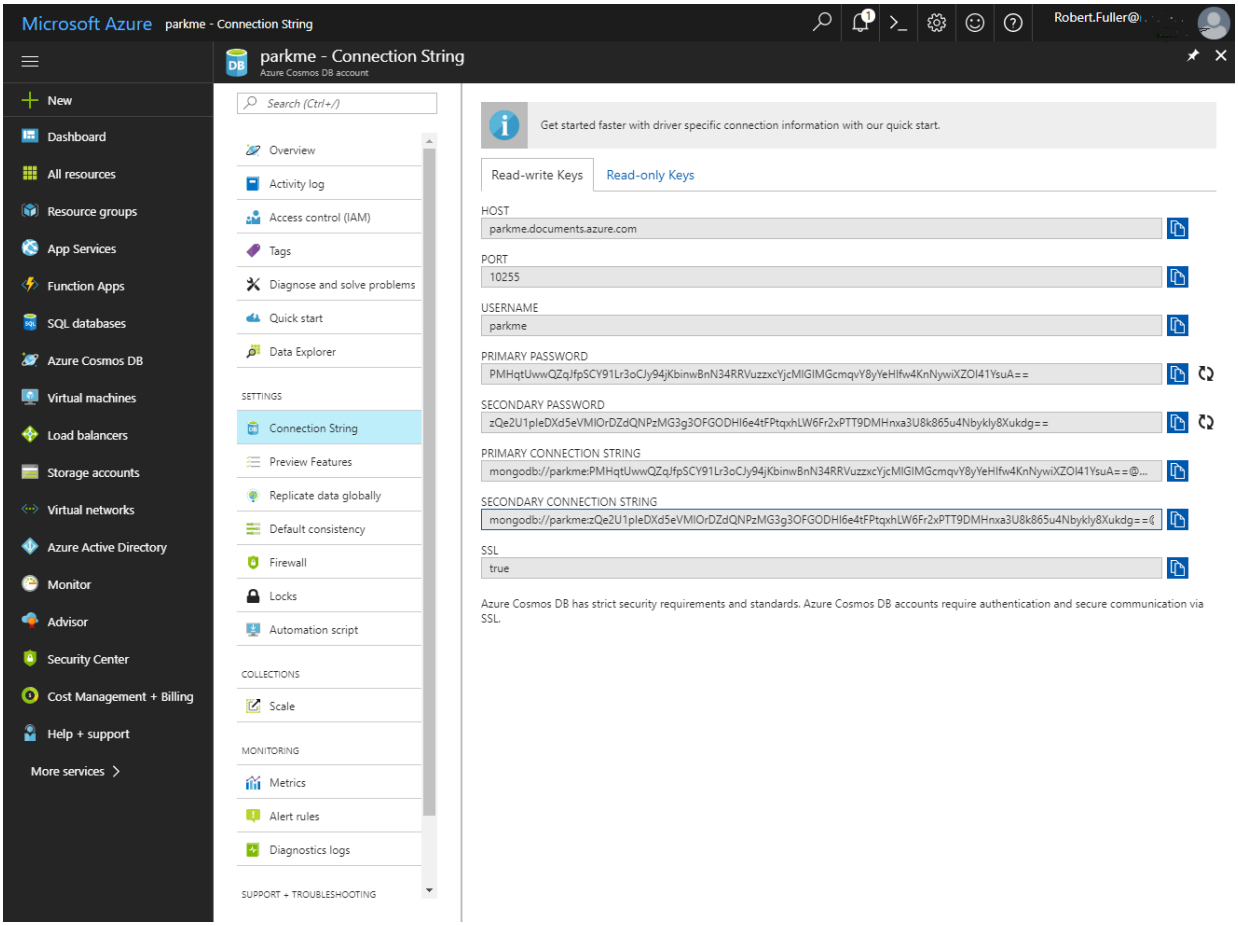
Outline of subsystems to hardware:

- Space Sensor: Ultrasonic Sensor HC-SR04
- Space Report: Raspberry Pi

- ParkMe Database: Azure Cosmos DB "dbName=parkdbcsus"
- REST API: Azure virtual machine
- Web Admin Site: Azure virtual machine
- Web Driver Site: Azure virtual machine
- Mobile Driver App: Android mobile device

Persistent Data Storage

A repository of the parking lots will be managed in an Azure Cosmos DB. The following data schema will be used to store JSON objects.



Parking Lot Schema

{ "lotId" : "number", "lotState" : "string", "lotType" : "string", "lotVacancy" : "number", "lotGeoCode" : "string", "lotDescription" : "string" }

Parking Space Schema

{ "spaceId" : "number", "spaceState" : "string", "spaceReserved" : "boolean", "spaceGeoCode" : "string" }

Destination Schema

{ "destinationId" : "number", "destinationTitle" : "string", "destinationDescription" : "string" }

Network Protocol

The following network protocols are used:

- ParkMe Database REST API: HTTP
- ParkMe Web App: HTTP
- Space Report: IoT

Global Control Flow

Our system will be based on Time-Dependency, the sensors will check the parking spot status on real-time and it will update the database every few seconds, and the results will be sent to the mobile app if it's ready to be routed.

Algorithms and Data Structures

Algorithms

Design Patterns

Observable

- Database calls via HTTP
- CandidateSpaceList collection to be notified when space availability changes

Data Structures

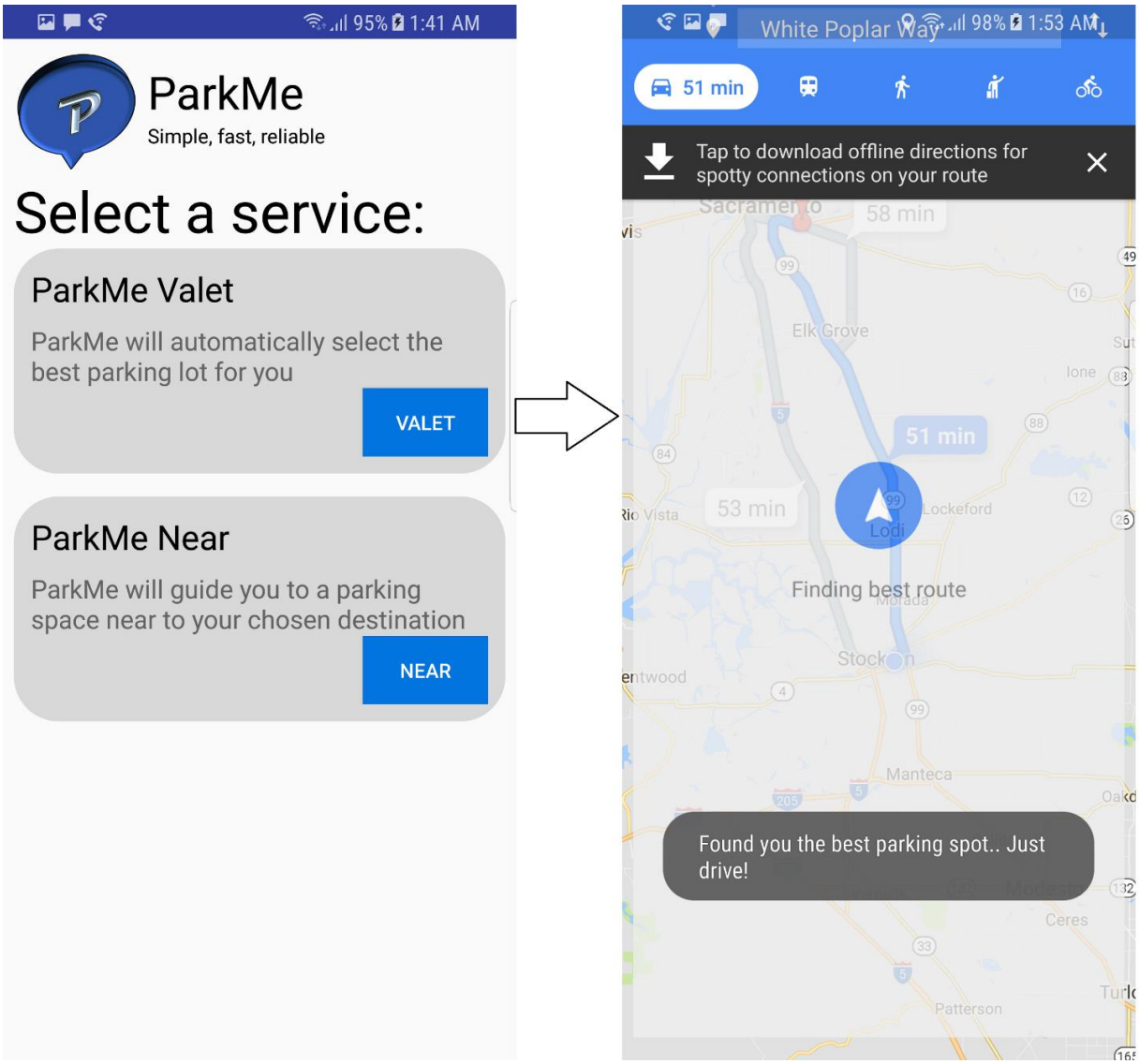
Lists

Lists will be the main data structure used to store information such as vacant spaces, each driver's set of space options, etc. The Web Admin classes will be the ones that predominantly use this data structure to store the information.

User Interface Design and Implementation

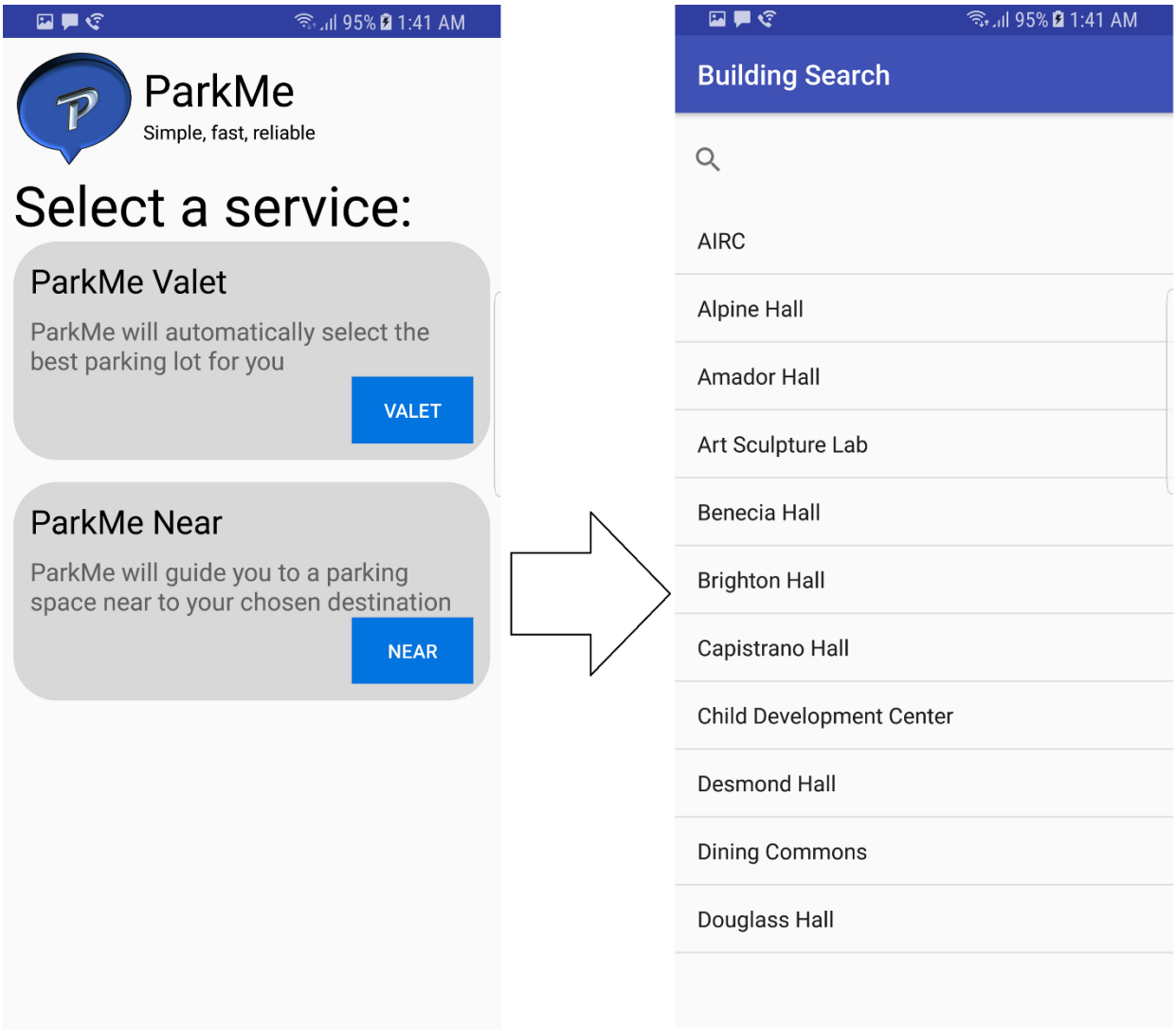
Simplicity is what distinguishes the ParkMe app for other solutions. Thus, the user interface is minimal since our objective is a hands-free experience for the driver once they are enroute and unless the administrator needs to upload a new park definition or suspend access to a parking space the Sensor is self-reporting so maintenance is minimal. However, we do have some simple user interfaces that can be accessed as shown below:

Valet: *User Selects ParkMe Valet. The app automatically finds the best available parking space for the user.*

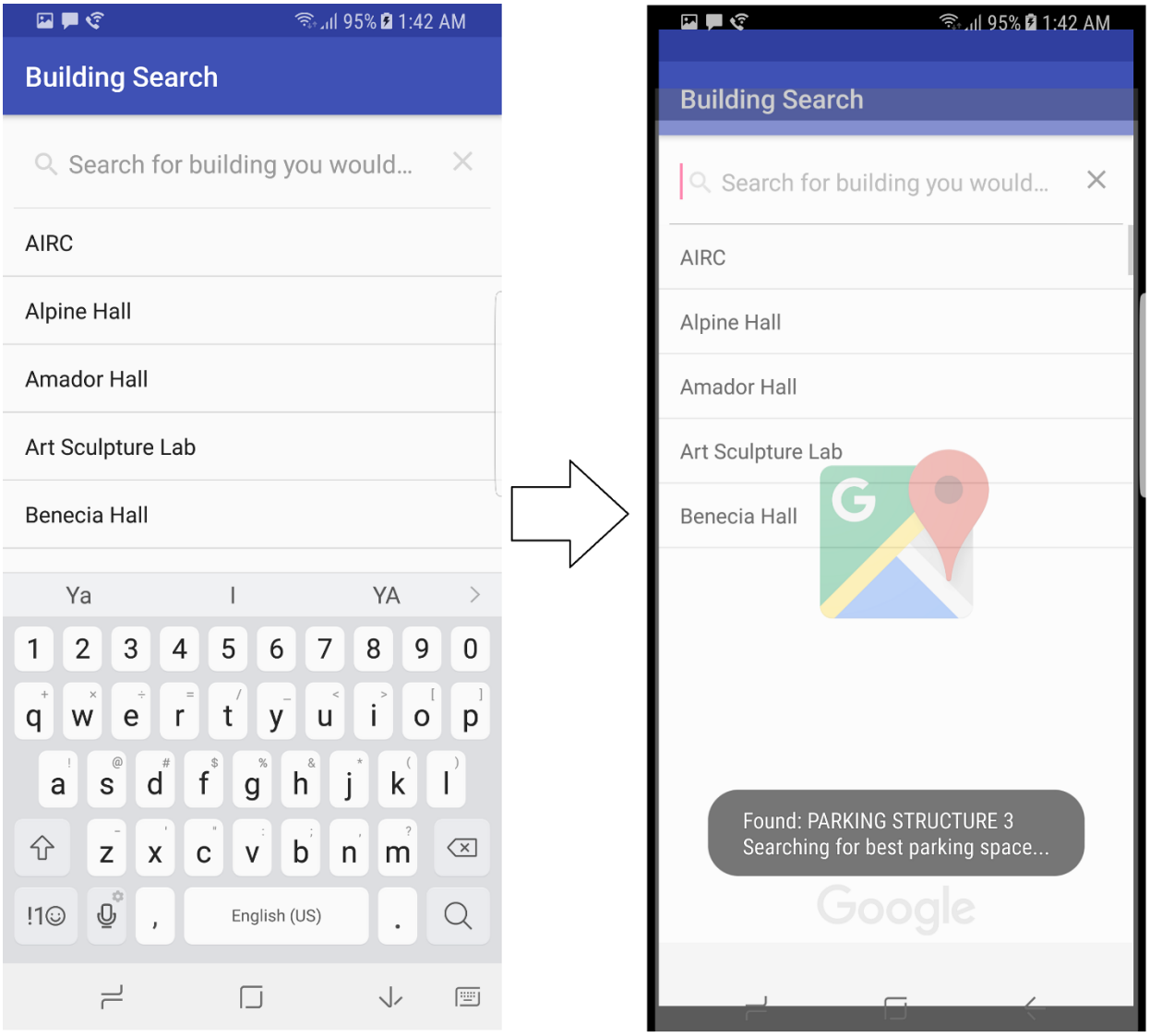


Near:

User selects ParkMe Near. They can scroll/search through a list of buildings.

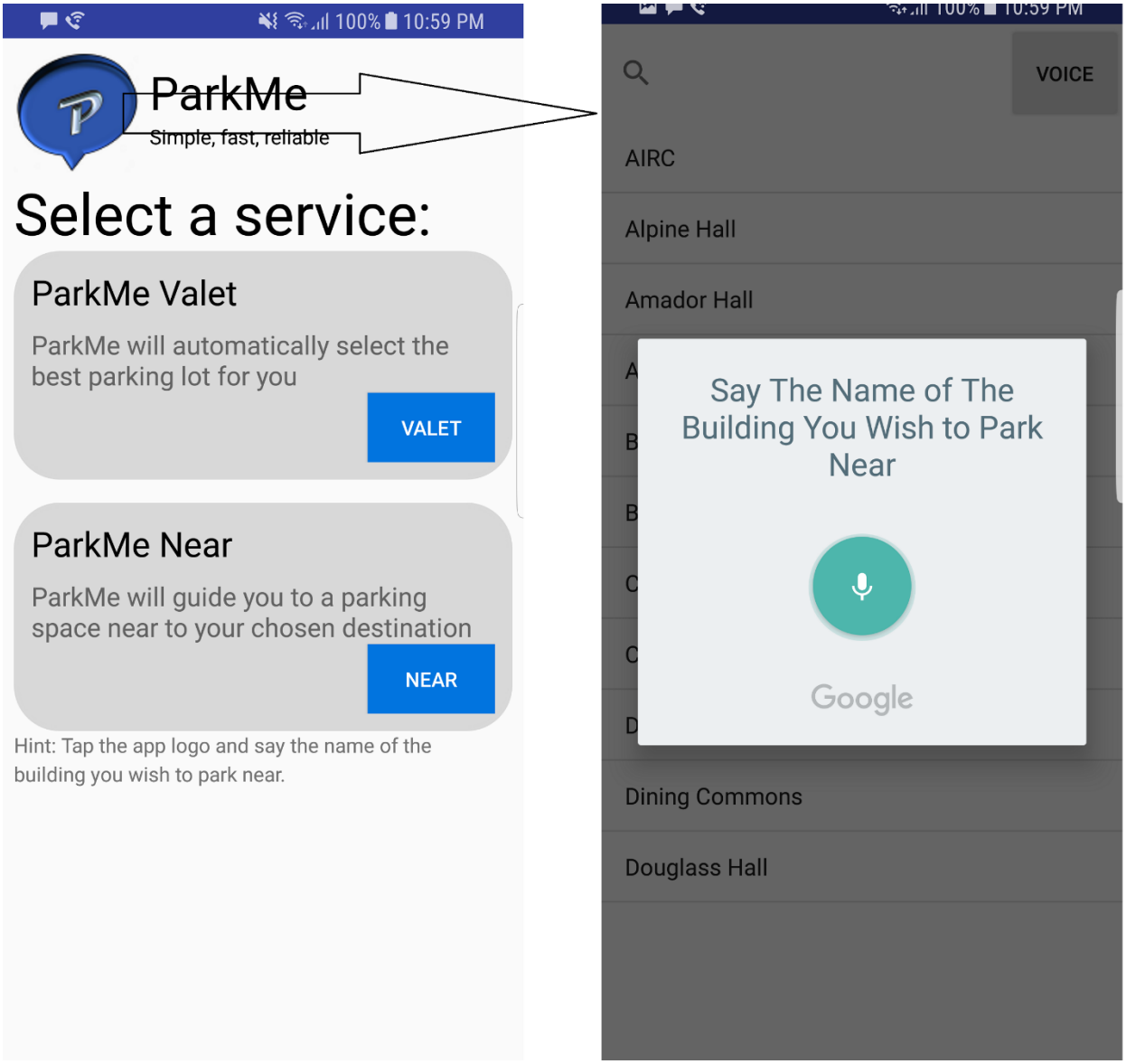


Once the building is selected, the app finds the user the best lot and then the best parking spot inisde that parking lot.

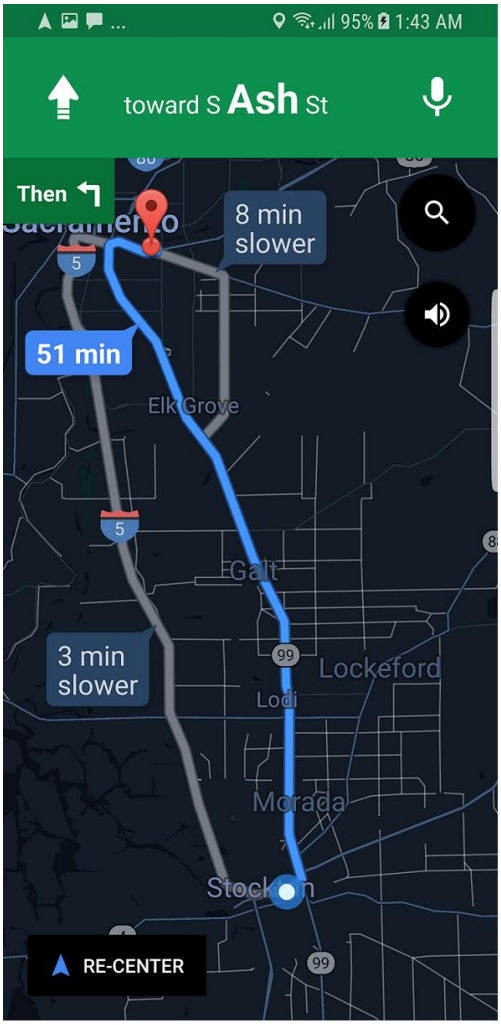


Voice-Functionality(Using the Near Case):

The user can also, when at the main front page, tap the app logo to speak to the app and say the building where they wish to park nearby so that the app can find them a parking spot.



Using the google maps app, it then gives the user step-by-by directions to the parking space all the way up until the user is parked at the parking spot. This is for both the valet and near scenarios.



Test Cases

A series of unit tests are implemented to test the Android API. The test are implemented by integrating the Espresso Framework to the system.

1. Verify the expected application layout upon application launch.
2. Verify the Valet functionality of the application. Tapping on the Valet button will take user to Google Maps API.
3. Verify that the Near button appears on screen upon launch is clickable by the user.
4. Verify clicking on the Near button will navigate user to parking lot list, and proper list appears on screen.

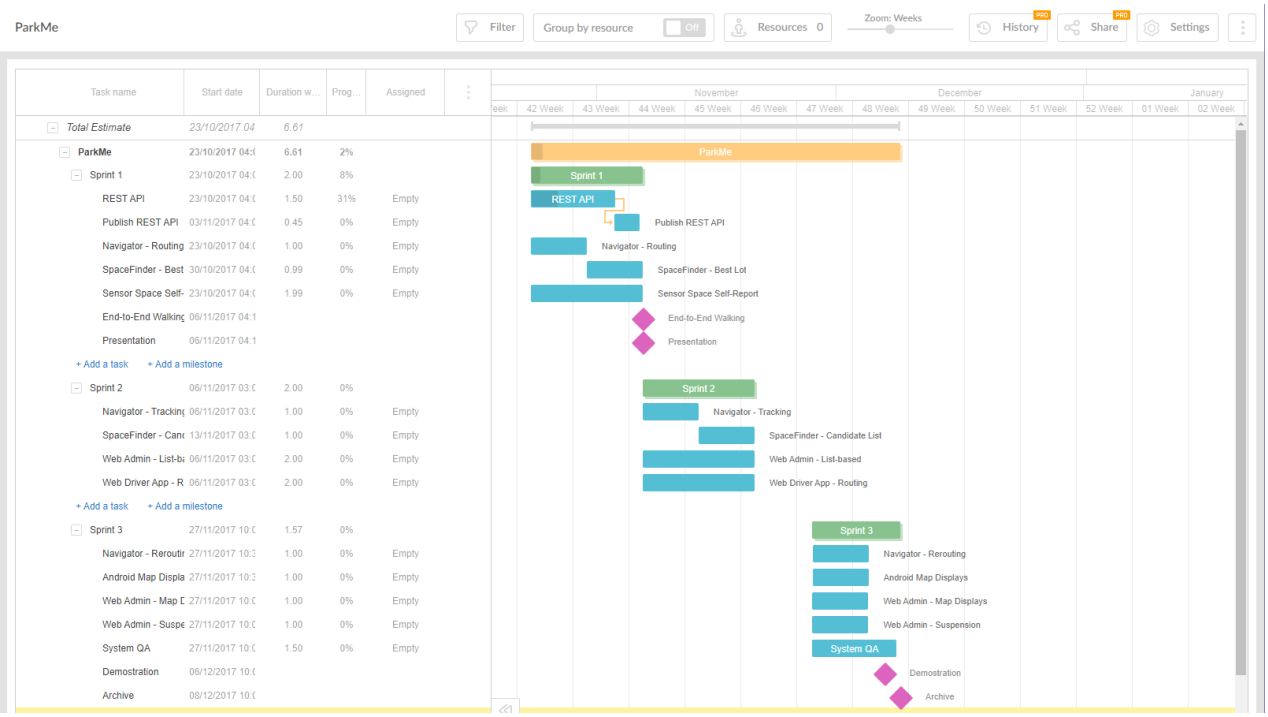
History of Work & Current Status of Implementation

During the course of the semester, the Pink Unicorns team has continued to refine its ParkMe solution. Significantly, we have identified the primary product differentiation, namely, that ParkMe minimizes interactions required from both the administrative and usage perspectives. In other words, ParkMe provides an automated parking solution.

Our goal for our implementation is to provide a hands free option for finding a parking space, from driver initiation to parking on campus. We will leverage location tracking and routing utilizing mobile device based technologies for audio interactions and available mapping technologies. This type of solution is in sharp contrast to other solutions, such as the one made available by Sac State, which require to user to scan a map to find out where to go. From the administrator perspective, we offer a self-reporting solution that can be placed in each parking space and can provide near real-time updates about the availability of each parking space in campus. This will enable us to both direct a driver to the best candidate parking lot at any time during the day and then route them to a vacant parking space. By comparison, the current Sac State map solution is based on the best guess of the administrator which is periodically submitted during the day.

The refinement of our solution has allowed us to focus on the key use cases which differentiate our product, in that It makes it extremely easy for the user to use. There should never be any type of hassle when using our app.

Work Done



Conclusions and Future Work

The Pink Unicorns have made significant progress during the course of the semester. Starting with the basic idea to develop a solution to streamline parking on university campuses, we have iterated through the ideation process to develop an approach which leverages technology that can minimize interactions and is largely self-supporting. This is significant since our primary users will be driving while using our application and campus administrators are faced with the daunting challenge of providing real time information about the availability of parking in large lots scattered across sprawling campuses.

During our due diligence, we contacted the Sac State parking enforcement and discovered that the current approach for providing parking availability is based on a best guess algorithm where an administrator estimates the percentage of parking spaces available for each lot and periodically posts these guesses. Drivers can then access these best guesses, which may not even be timely, but they are only provided with a high level map that is poorly labelled and requires scrolling to see different sections of the campus. Imagine trying to use this interface while driving.

Two fundamental challenges were faced by our team during the design process:

1. How can we get a real-time report for the status of each parking space on campus (literally thousands of spaces)?
2. How can we help our user find the best available parking space at any giving time during the day?

Fortunately, with a bit of inspiration and the aid of technology we were able to develop a solution architecture that addressed both of these challenges. To track real-time parking availability across lots, a self-reporting sensor can be placed in each parking space and coupled with strategically located Raspberry Pi computers to report the availability of each space throughout the course of the day to a cloud-based database. This information can then be accessed through mobile devices where we tap into location services to track progress of the driver as s/he heads to campus and rely on Google Maps to narrate the route to the best parking space.

The future plans of work the Pink Unicorns will consider are:

1. Can we initiate by voice using Google on Android (e.g., "Google, start ParkMe Valet") and make the app fully voice controlled?
2. Add a feature to check statistics of all parking spaces that shows the density of the parking structures/lots overtime, and shows the peak hours of parking. The feature also shows which lots has are the easiest to find a spot for most hours.

References

Sac State Parking Map

http://www.csus.edu/nsm/about%20nsm/resources%20docs/color_map.pdf

Google Maps API

https://developers.google.com/maps

Android Studio

https://developer.android.com/studio/index.html

"Software Engineering book", Ivan Marsic

http://www.ece.rutgers.edu/~marsic/books/SE/book-SE_marsic.pdf

Azure Cosmos Database

https://docs.microsoft.com/en-us/azure/cosmos-db/documentdb-java-application