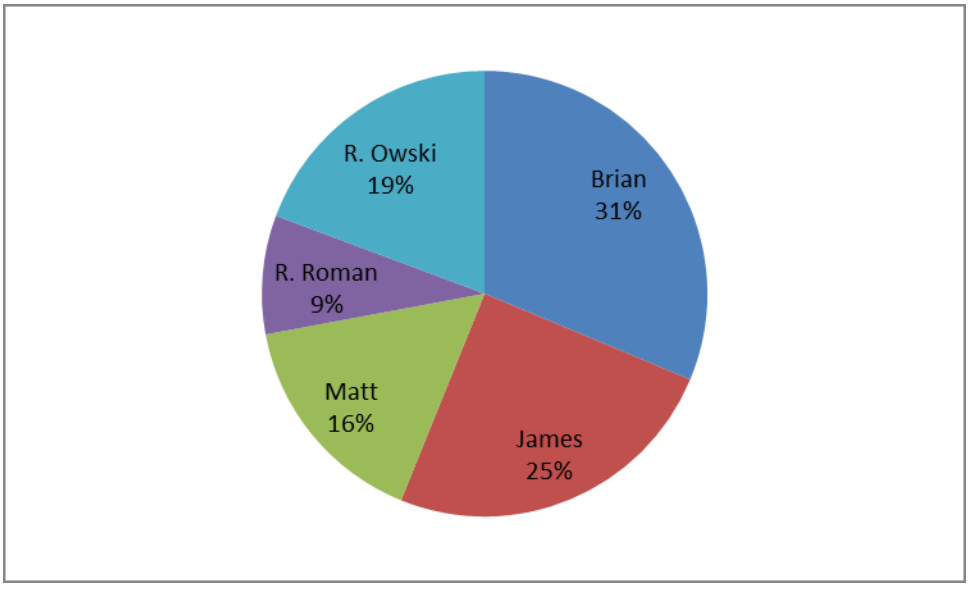
Report #3

Brian Goodacre, James Jacob, Richard Roman, Richard Romanowski, and Matthew Rodriguez

Table of Contents

Effort Breakdown

#	Section	EBP	Brian	James	Matt	R. Roman	R. Owski
1	Cover Page and Breakdown	0					
2	Table of Contents	0					
3	Summary of Changes	5	100.00%				
4	Customer Statement of Requirements	1				100.00%	
5	Glossary of Terms	1				100.00%	
6	Functional Requirements Specification - R Roman	1				100.00%	
6b	Use Case Diagram	0					
6c	Fully Dressed Use Case	0					
7	Nonfunctional Requirements	1				100.00%	
8	Effort Estimation using Use Case Points	6	20.00%	80%			
9	Domain Analysis	9		100.00%			
10	Interaction Diagrams	10		100.00%			
11 a	Class Diagram	11	100.00%				100.00%
11 b	Interface Specification	11	50.00%				50.00%
11 c	Design Patterns - R Roman	6				100.00%	
11 d	Object Constraint Language Contracts	5		100.00%			
12	System Architecture and System Design -owski	6	10.00%				90.00%
	UML Package Diagram	5	100.00%				
13	Algorithms and Data Structures	5	100.00%				
14	User Interface Design and Implementation	5			90.00%		10.00%
15	Conclusions and Future Work	7			100.00%		
16	History of Work and Current Status of Implementation	7			100.00%		
17	References	0					
18	Compiling Report	3	100.00%				



Summary of Changes

- Project Objectives

  - Our project objectives did change. Our first goal was to make sure that the parking garage could accurately store customers, cars, reservations, and money transactions. The details of managing these multiple fields because very tedious but worthwhile for the ease to the non-database related functions.
  - Our test bench was successfully multithreaded
- Use Case Descriptions
  - The Use Case Descriptions did not change that much. Effectively, the use cases are the same except that more now happens behind the scenes for each Use Case. For example, make a reservation requires searching for an open spot and booking that spot, an operation unseen in by the use cases.
- System Design
  - Class Diagram
    - The Class Diagram was redone to reflect the actual code of the working project. It included all the class variables that were used along with all the functions. It is quite large but pretty.
    - New to the class diagram are the reservation, customer, and car classes that relate on the database. Additionally, the organization of the testbench and GUI classes are better organized.
    - The older class diagram includes items that we were not able to get to, such as the prediction and administrator. There are revisions to how the Hardware Controller interacts with the various pieces of hardware. Additionally, the Parking Database has been removed and replaced with a more functioning parking space table.
  - Package Diagram
    - The packages that we used changed as we began programming.
    - A layer was added on top of the database so that outside functions could only call and create objects. They no longer had direct access to the database.
    - In the previous package diagram, there were more packages that were spread across many areas of expertise. Organizing the packages into easily defined groups was difficult. The ways the packages accessed each other was poor and not sustainable.

        - Now, in the current package diagram, the design is much more modular, such that only a few packages access the database. These packages provide all the information required for the other packages.
        - Furthermore, the packages for the testbench and interface are much more related and all run through the common Main package. Once again, this is more modular. The various packages need to import the Main Class so that it has access to the variables in the main class.
- Database
    - Increased the tables in the database from two to five
        - Customer, Car, Reservation, Parking Spaces, Accountant
        - This is very important because it allows for the non-database accessing functions to use simple objects, which in turn edit the database.
        - By increasing the number of tables, more information could be stored in a much easier and faster manner. For example, previously to determine with customer had a car with license plate "XXXAA," one had to search through all customers and all their plates. Now, one can do a MySQL query and receive the car ID of the car with that plate, along with the customers associated with that car.
    - New to our design were functions in the Database_Constructors class that could return a customer, car, or reservation based on an ID number of upload a customer, car, or reservation to the database. This made for less MySQL queries and an overall easier programming environment.
- Interface
    - Website
        - An additional interface was added through TomCat: a website. This website is a simple website that will be improved, but the functionality as previously discussed improved but not to a very useable standard. One can manage their account and make a reservation through the website. Previously, the website design allowed for logging in, so in that aspect, we have gone much further. In the aspect of the requirements of the website, we did not completely meet all of our requirements but built our program in a way that future modifications can be added easily.
    - GUI

        - This aspect of the program improved greatly. Enter, Exit, Mobile, and Administrator interfaces were developed, more than previously designed for. We worked hard to improve the functionality of the GUI's by having additional screens for the user, such as confirming a license plate or making a reservation at drive up. This improved from the previous design because there is much more happening now.
- Test bench
    - Multithreaded
        - The previous designs called for a multithreaded database, and it happened. Cars can enter and leave in this demonstration at the same time. Also, the elevator can be moving and then the exit gate be called while the elevator is on the move.
    - More Hardware
        - All hardware requested was implemented. It was difficult but an authentic representation of a parking garage can now occur because of the multithreading and hardware that was created.

Customer Statement of Requirements

Essay Version

This project is designed to fully automate a parking garage. Another goal is to increase profit by using methods to achieve optimum occupancy. At the moment, the parking garage is being run with no form of computerized system. Additionally, the system is being organized by employees entering the information manually into excel type programs. Additionally people either pay at machines, or at tollbooths at the exit. Currently, people are allowed to park as long as they want, and pay an hourly rate. Its quite apparent that a system like this would be likely to have many issues in terms of accuracy as well as overall efficiency and speed. Economically it is also impractical to have people in charge of checking parking spots as well as writing down all of the customer information manually when a computer and a camera could achieve the same thing faster, cleaner, and be cheaper. The new system will encourage customers to make reservations through the website, or mobile phone app. This reservation will include time and duration of stay, and will allow the customer to choose certain parking preferences such as distance from the elevator or floor level. Because one car can also be registered to two different accounts, at the time of reservation, the user will have the option of inputting the license plate number of the car that he/she will be using.

The garage is currently being remodeled so that the parking decks above the ground level will only be accessible via an elevator that will lift the cars to different decks. Additionally there will be a door in the elevator that will open to allow for ground floor access. There will be a ramp that the cars can use to exit the garage so there is no two-way traffic at any time to prevent collisions and drastically decrease the chances of an accident. The garage will rely on cameras rather than electronic tagging for vehicle recognition. Customers will be required to register at the company website in advance to making reservations at the parking garage. The customer will not be tied to his license plate number since many drivers have access to multiple cars but it will be possible to input a license plate number to associate with the customer profile.

The system will rely on cameras, sensors, and a database to manage the garage which will seemingly put a little bit of pressure in terms of data accuracy on the customer. For example if the license plate recognition system doesn't recognize the registration number, the elevator will not move causing the customer to have to input their membership information such as a unique membership ID number before they are allowed to progress to their spot. This also happens if the car has no front license plate. If the license plate is not registered to an account, a new account can be created at the elevator interface.

Customers who are registered may be allowed to take a walk-in parking spot without a reservation if there are available spots at the time. If the vehicle registration number is recognized, and the system cannot find an existing reservation for that customer or vehicle, then the customer will be able to specify the expected duration and time of departure using the keypad in the elevator. They are referred to as registered walk-ins. For every reservation there will be a half hour grace period, in which the spot will be held for the patron who made the reservation. After the expiration of the grace period, the parking spot will be marked unreserved and added to the list of open parking spots. A customer who does not show up without notifying the garage that they will not be arriving will be billed for the entire duration of the interval they initially signed up for. A customer who does show up, but after the grace period will be allowed to extend their reservation, assuming there are spots available to accommodate their needs, otherwise they will be out of luck. The customer will be billed from the start time detailed in the reservation, until the end time, or if time was extended, the new end time.

A customer also has some flexibility if they want to stay longer than they were expecting to stay. A customer can extend their session up to a half-hour before the scheduled expiration granted there are not any conflicting reservations with spots. They can extend an unlimited amount of times as long as

there are still unreserved spots available. In contrast, the customer will be billed at a higher rate for every minute he or she spends in their spot after their expected and set duration time.

As with any business that depends on predetermined duration, people will leave earlier than expected. In this case, the customer will still be billed for the full duration. The spot will then be marked vacant and will be added to the list of open spots.

The customer is allowed to have three standing reservations, as long as each reservation is at least one hour apart from each other. This is meant to encourage people to merge multiple reservations into a single reservation. One problem that may arise frequently is when someone has a reservation that relies on someone leaving at a very close time to his/her arrival. If said person in the spot doesn't leave at the expected time someone might have to wait for the spot. If a spot is open at the time than the waiting customer will be redirected to that spot, otherwise it will be a problem since there will be a waiting customer with no spot to park in. There be rain checks for when this occurs and for the times when the garage is full. We assume that people will leave the premises when they are asked to and will not take an open spot that is most likely reserved for someone else.

The payment system for customers will be entirely electronic. Customers will have an account tab that is recalculated every time they make a reservation or park as a walk-in. This account will have to be paid off by the customer. Walk-in customers will still pay at the exit with a credit card.

While much of the success of the system relies on the customers being able to easily navigate the user interface of the system, it also requires that the electronic devices being used work effectively when integrated into the system. One of the most important devices is our license plate reading camera. There will be one in the elevator at the entrance and one at the exit. The first camera will attempt to match the license plate number with an account and any standing reservations that the account has. The second will read the license plate numbers of cars exiting to report when they exit and how they correspond to the reservation, or specified duration (walk-ins). It will be able to report if a car has overstayed, left early, or left on time. For this project we assume that the initial camera always can read the license plate and only does not recognize a number if it is not registered. We assume that the exit license plate always reads the license plates.

Another device that will be necessary is the sensor on each parking spot. This sensor will only be able to report when a spot is vacant or occupied. We assume that the sensor will always work, and will only report an occupied status when a car is in the spot, and not for instance when the garage is dark. We

also assume that each car parks in the assigned spot. In order to have sensors that check the car parked, the expenses for the project would be greatly increased.

There will need to be a digital display in the elevator that displays messages to the driver. These messages will include prompts for a keypad, and also what spot the car has been assigned. The keypad will serve the function of data input. Drivers will be able to enter their account / reservation number, license plate number, and if they are walk-ins or have a reservation. It will work with the license plate reader to make sure the registration numbers match the license plates so people can't come in and take other people's spots. We assume that the elevator will always stop at the correct floor.

As mentioned before, we must make some assumptions in this project in order to simplify it. If we did not this project would be much more complicated, and we would run the risk of not completing it in the time allotted. We assume that if a vehicle is registered to one or more accounts, the driver both knows and has permission to use one of the account numbers. We also assume that the customer has an email address, a credit/debit card, and a mobile phone with SMS text capabilities.

All storage of information will be in an on-site database. We will be using an on-site database, instead of a remote database because it will not require extra calls to a remote site. The database will store the information in the registered accounts, the occupancy status of each spot, and current parking reservations. The garage operator will be able to use the database to view the user accounts and view past customer activity. Customers will be able to access database information via the website or a mobile app to check parking space availability, view reservation number, and extend reservation.

The system will be tested using a simulator. A website interface will be designed for the garage. This will allow the basic functions mentioned previously. The simulator will consist of a program that will simulate actual parking. It will include arrival/departure buttons. The program will ask for entry of license plate number. If the plate number is not recognized, the account number will be requested. A walk-in option will also be available on the keypad. The program will inform the user of his/her parking spot, and the user will confirm that the car is parked correctly. Another part of the testing program will inform the database which spots are occupied and which spots are taken. This part of the program determines if there are no shows, late arrivals, extended reservations, or failures to depart on time, and determines when the spot is released. The parking operator will be able to configure the simulator with parameters such as the total capacity of the parking garage (number of floors, spots per floor, etc.).

List Version

1. The customer shall be able to register for an account online, and be able to make reservations for parking.

2. The customer shall be able to make reservations and view reservation information for parking on the mobile smart-phone app.

3. Upon entry into the elevator, the system should be able to identify if the customer has a reservation, through either the camera recognition system or from the keypad, and transport the user to the appropriate floor and provide the customer with his or her assigned parking spot.

4. If the customer has no reservation he or she should be able to specify walk-in status and input duration of stay. The system will then assign them an unreserved spot and transport them to the appropriate floor.

5. The system shall be able to report which parking spots are open and which spots are occupied using the parking spot sensors.

6. The system shall be able to report when a car is leaving and determine if it has left on time or has overstayed its specified time slot.

7. The system should allow the garage manager to access database information.

8. The system shall be able to assign parking spots to reservations and walk-ins taking into account the schedule of all of the reservations and walk-in durations.

9. The system shall be able to adjust these assignments if parking spots are not vacated when they are expected to be, or if they are vacated sooner than expected.

10. The system shall be able to charge accounts after every reservation/walk-in parking session.

11. The system shall be able to change (extend) reservation times easily, and rearrange the parking spot assignments accordingly.

12. The system shall be able to recognize invalid reservations (inaccurate information, or contiguous reservations).

13. The system shall allow an employee to see how many spots are taken/reserved and how many spots are open/unreserved

14. The system shall open the elevator door when it reaches the correct floor, and shall open the back door when it is ready for another car.

15. The system shall close the elevator back door when it is transporting the car to the proper floor.

16. The sensors shall identify occupied parking spots to the system.

Glossary of Terms

Camera – device that reads in the license plates of cars and provides this information to the system

Customer – anyone who parks in the parking garage (garage)

Database – the entity that will be storing all of the information for the garage such as user accounts, rates, and past garage data.

Device – object that will be installed in the garage, including license plate readers and occupancy sensors

Extensions – the act of redefining the end time of the reservation. Extensions can be made up to a half hour before the end of the reservation, provided there are unreserved parking spots.

Elevator display – displays the output for the customer on the elevator

Elevator interface – includes the elevator keypad (keypad) and elevator display (display) that allows customer input and output on the elevator.

Elevator keypad – allows the customer to input information on the elevator

Grace period – an allotted amount of time of 30 minutes that starts at the beginning of a reserved interval. During this holding time, the spot will not be given away even if the customer hasn't arrived to allow for lateness. This grace period can be extended to longer amounts of time for an additional fee. The customer will still be billed for this time period.

Guaranteed reservation – a type of reservation; a monthly contract that allows the customers to make a contract with the parking garage for a parking spot for a predetermined period outlining specific hours.

License plate reader – a digital camera combined with a recognition system to send license plate number of vehicle to system interface

Mobile App – this is a user interface designed for smart phones. This will provide the more basic capabilities to customers on the go. Customers will specifically be able to make reservations and view information about their reservations.

No-Show – the act of missing a reservation. The customer will still be billed for the reservation

Occupancy sensor – sensors installed in parking spots to determine if the parking spot is occupied or vacant

Overbooking – the act of accepting more reservations than available parking spots. This is a strategy used to maximize profit, and takes into account any reservations that are not fulfilled. This will be determined with an equation for a Poisson variable.

Overstay – the event in which a customer does not leave his or her spot until after the reservation has terminated. The customer will be charged an increased rate, and will be notified via email.

Rain Check – a credit that goes to the account of any customer that is unable to use reservation because of lack of available parking spots. The customer is able to make another reservation of equal or lesser duration for free in the future.

Register¬ – to visit the company website and complete a reservation and become a registered customer. Any customer who does not register is an unregistered customer, and may only obtain reservations via a walk-in.

Registered customer – any customer who has previously registered on the company website and provided all necessary information at registration time, and who has a registration identification number associated with them.

Registration number – a number that is associated with each registered customer that uniquely identifies them

Registration time – when the customer registers with the company. The customer will provide demographic information, a valid email, and a valid credit card number. The customer may also provide license plates of his or her vehicles.

Reservation – an agreement between the parking garage company (company) and the customer to hold a parking spot in advance

Reservation confirmation number – a uniquely given number given to a customer for each reservation he or she holds. This number is used as another option to identify the customer if the license plate recognition system fails, and/or the customer is driving in a vehicle that is not registered to them.

Vehicle – any car that can be parked in the parking garage. Note that large vehicles such as large trucks or busses will be unable to use this parking garage.

Walk-in – a reservation made at time of arrival. May be made by an unregistered or registered customer

Website – the interface online that will allow customers to register accounts, and vehicles. Customers will be able to make reservations, pay bills, and view account history.

Functional Requirements Specification

Stakeholders

- Commuters
- Parking garage owners
- Local businesses
- Parking garage employees
- Tourists & shoppers

Actors and Goals

- Customer (Initiating)
  - To make a reservation for a parking space
  - To find his/her parking space within the garage
- Employee (Initiating)
  - To check the availability of open parking space within the garage
- CameraIn (Initiating and Participating)
  - To identify the license plate number of a vehicle entering the garage and send the data to the system in order to use the elevator
- CameraOut (Participating)

- To identify the license plate number of a vehicle exiting the garage and send the data to the system in order to open the gate

- Sensor (Participating)
  - To identify whether or not a parking space is occupied and send the data to the system
  - To identify whether or not a parking space is vacant and send the data to the system
- Keypad (Participating)
  - To acquire the customer reservation number and send the data to the system in order to use the elevator
- DebitMachine (Participating)
  - To charge the customer for his/her stay within the parking garage if not registered
- VehicleIn (Initiating)
  - To navigate to the appropriate parking space within the parking garage
- VehicleOut (Initiating)
  - To navigate to the Parking Garage Exit and pay the parking machine

Use Cases

Casual Description

- updateReservations
  - Allows the customer to check his/her reservations with the parking garage and add to, change, or remove from them as necessary
- makeReservation
  - Allows the customer to make a new reservation with the parking garage
- useElevator
  - Allows the elevator camera to scan the license plate number of the incoming vehicle in order to use the elevator and open/close the elevator doors
- openDoors
  - Opens the elevator doors on the appropriate level of the parking garage for incoming vehicles to navigate to their parking space
- closeDoors
  - Closes the elevator doors when the elevator is ascending or descending
- spaceOccupied
  - Identifies that the parking space is occupied and sends this information to the system

- spaceVacated
  - Identifies that the parking space is vacant and sends this information to the system
- updateDatabase
- Updates the database whenever any information changes, including account information, or account history
- openGate
  - Opens the exit gate when either the camera has identified the vehicle license plate number and debited the corresponding account or the customer has manually paid the fee
- payFee
  - Allows an unregistered customer to manually pay their parking fee via cash or credit/debit card
- chargeCar
  - For a registered user, the duration that car has spent in the parking space has been calculated and the user charged

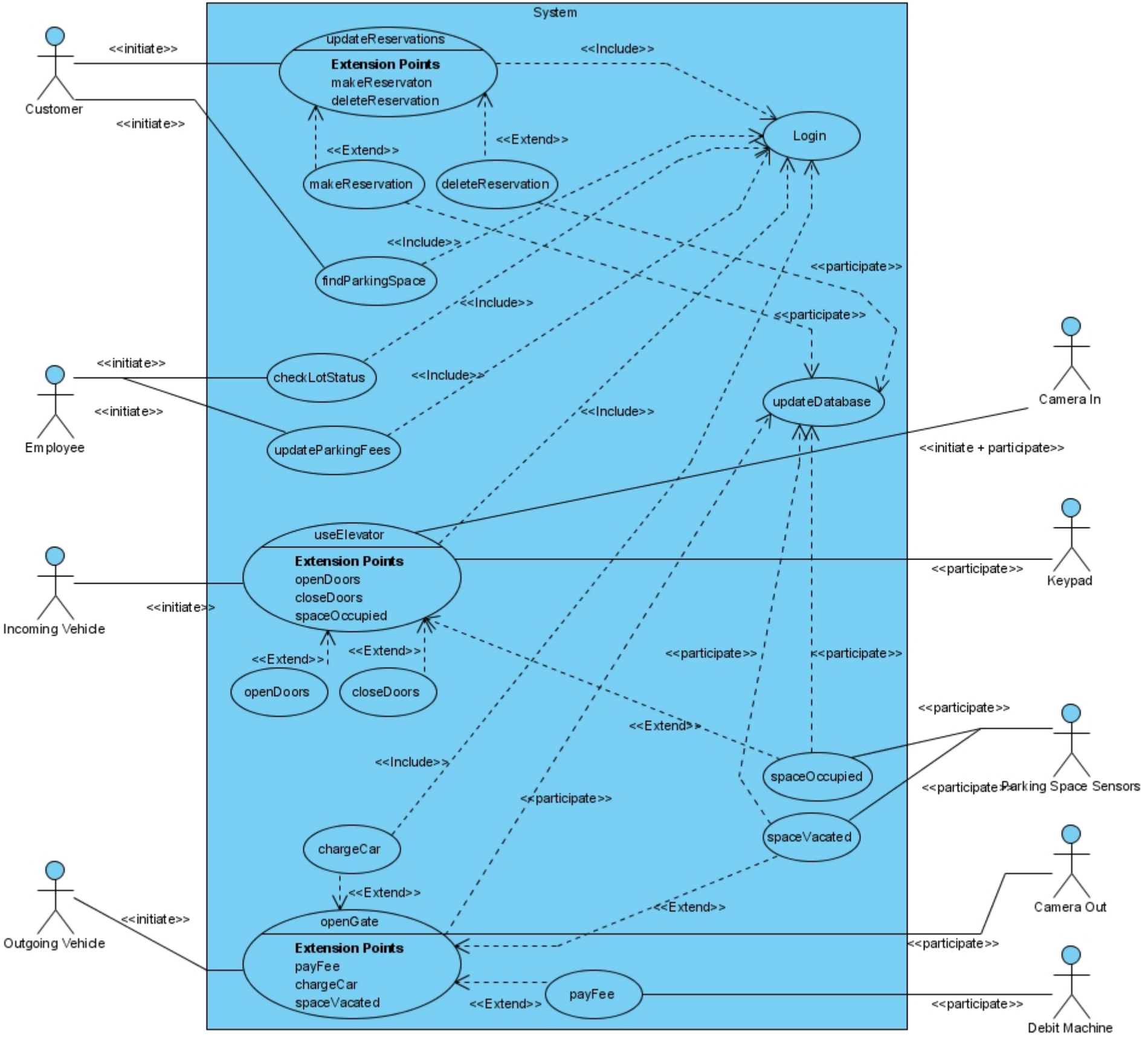
Fully Dressed Description

- Use Case UC-1: updateReservation
  - Related Requirements: req1, req2, req15
  - Initiating Actor: Customer
  - Actor's Goal: To make changes to his/her parking garage reservations
  - Preconditions:
    - Customer has registered with the parking garage reservation system
  - Postconditions:
    - Parking garage database is updated with new reservations
  - Flow of Events:
    - →: Customer loads parking garage service website and logs in
    - →: Customer enters updated reservation information
      - Time period, preferences, vehicle
    - Service website updates system database with new reservation
    - ←: Customer received reservation confirmation
    - →: Customer logs out of service website

- Use Case UC-2: useElevator
    - Related Requirements: req3, req4
    - Initiating Actor: VehicleIn
    - Actor's Goal: To enter the parking garage and find a parking space
    - Preconditions:
        - Elevator car is located on ground floor
        - Front door of elevator is open
        - Vehicles pulls into elevator car
    - Postconditions:
        - Elevator car returns to ground floor
    - Flow of Events:
        - →: Vehicle pulls into elevator car
        - ←: Camera scans vehicle license plate number and sends data to database
        - ←: Elevator doors close, elevator rises to appropriate floor, elevator doors open
        - →: Vehicles exits elevator car
- Use Case UC-3: openGate
    - Related Requirements: req7
    - Initiating Actor: VehicleOut
    - Actor's Goal: To pay parking fee and exit the parking garage
    - Preconditions:
        - Exit gate is lowered
    - Postconditions:
        - Exit gate is lowered again
    - Flow of Events:
        - →: Vehicle pull up to exit gate
        - ←: Camera scans vehicle license plate number and sends data to database
        - ←: Exit gate raises
        - →: Vehicle exits parking garage
- Use Case UC-4: payFee
    - Related Requirements: req8, req10, req13
    - Initiating Actor: VehicleOut
    - Preconditions:

        - Exit gate is lowered
        - Vehicle license plate number is not registered with system
    - Postconditions:
        - Exit gate is lowered again
    - Flow of Events:
        - ←: DebitMachine prompts customer with payment data
        - →: Customer feeds money into debitMachine
        - ←: Exit gate raises
        - →: Vehicle exits parking garage
- Use Case UC-5: spaceOccupied
    - Related Requirements: req5, req6, req8
    - Initiating Actor: VehicleIn
    - Preconditions:
        - Parking space is currently empty
    - Postconditions:
    - Flow of Events:
        - →: Vehicle pull into parking space
        - ←: Sensor identifies that parking space occupied and updates the system

Use Case Diagram


System
Customer
Employee
Incoming Vehicle
Outgoing Vehicle
Camera In
Keypad
Parking Space Sensors
Camera Out
Debit Machine
<<initiate>>
updateReservations
Extension Points
makeReservaton
deleteReservation
<<Include>>
Login
<<Extend>>
makeReservation
deleteReservation
findParkingSpace
<<participate>>
checkLotStatus
updateParkingFees
updateDatabase
<<initiate + participate>>
useElevator
Extension Points
openDoors
closeDoors
spaceOccupied
openDoors
closeDoors
spaceOccupied
spaceVacated
chargeCar
openGate
Extension Points
payFee
chargeCar
spaceVacated
payFee

System Requirements – Use Case Traceability Matrix

Use Cases	updateReservations	makeReservation	useElevator	openDoors	closeDoors	spaceOccupied	spaceVacated	updateDatabase	openGate	payFee	chargeCar
Requirements											
Register Online		<-						<-			
Make Reservation Online		<-						<-			
Make Reservation on App		<-						<-			
Identify Reservation (at entrance)			<-								<-
Identify Walk-In (at entrance)			<-							<-	
Specify Duration of Walk-In								<-			
Report Overstay										<-	<-
Report Vacant Spot							<-				
Identify exiting Vehicle							<-	<-	<-	<-	<-
Assign Parking Spot		<-						<-			
Adjust Parking Spot Assignment	<-							<-			
Adjust Reservation	<-							<-			
Recognize Invalid Reservation/ Parking Duration			<-								
Open Elevator Door			<-	<-							
Close Elevator Door			<-		<-						
Identify Occupied Parking Spot						<-					
Pay Bill Online											<-

Nonfunctional Requirements

- Customers reservations are always satisfied
- Any customers that make reservations are guaranteed to have a parking space during the time period that they requested. This can be accomplished by maintaining a balance as to how many cars are inside the parking garage at all times
- Transactions within the elevator and at the exit gate are quick and easy.

  In order to keep the flow of cars in and out of the garage moving, the actions taken within the elevator and at the exit gate need to be quick and easy. The camera should be able to scan the plate quickly and the user must be able to enter their reservation information quickly on the keypad. At the exit gate, the customer must be able to see, understand, and pay their bill in a timely manner with little difficulty.

- Because the website is run on a local-host at the moment, it is very fast and easy to use. Loading times are cut down an incredible amount because all of the pages and information are stored on the machine.
- Included in the program is a testbench that allows the user to test how the system works in order to better understand it.

Effort Estimation using Use Case Points

Unadjusted Actor Weight:

Controller: 2

Elevator: 1

Door: 1

Accountant: 2

ExitInteface: 2

EnterInterface: 2

Database: 3

Gate: 1

Sensors: 1

Website: 2

Camera: 1

UAW = 18

Unadjusted Use Case Weight

updateReservation: 15

useElevator: 10

openGate: 5

payFee: 10

checkSpaceAvailability: 10

accessAdminView: 10

UUCW = 60

Unadjusted Use Case Points

UCCP = 78

Technical Complexity Factors

T1: 3*2 = 6

T2: 1*1 = 1

T3: 1*1 = 1

T4: 3*1 = 3

T5: 2*1 = 2

T6: 1*0.5 = 0.5

T7: 1*0.5 = 0.5

T8: 2*2 = 4

T9: 1*1 = 1

T10: 2*1 = 2

T11: 1*1 = 1

T12: 2*1 = 2

T13: 1*1 = 1

TCF = 0.6 + 0.01 * 25 = 0.85

Environment Complexity Factors

E1: 5*1.5 = 7.5

E2: 4*0.5 = 2

E3: 2*1 = 2

E4: 2*0.5 = 1

E5: 3*1 = 3

E6: 3*2 = 6

E7: 1*-1 = -1

E8: 4&-1 = -4

ECF = 1.4 -0.03*16.5 = 0.905

Use Case Points

UCP = 78*0.85*0.905 = **60**

Domain Analysis

Concept Definitions

- Controller (doing)
    - Coordinate actions of all concepts and communication between concepts
    - Delegate work to other concepts
- EnterInterface (doing)
    - Display reservation information for each customer entering garage
    - Confirm with customer that scanned license plate number is correct
    - Retrieves reservation information from database
- License Plate (knowing)
    - Container for vehicle license plate number
- Database (knowing)
    - Container for all parking garage information including reservations, customers, vehicles, and parking spaces
- ElevatorOperator (doing)
    - Operate elevator car to move it between floors
- FrontDoorOperator (doing)
    - Operate the front door of elevator to open/close it
- BackDoorOperator (doing)
    - Operate the back door of elevator to open/close it
- ExitInterface (doing)
    - Display payment information for each customer exiting garage
    - Retrieves reservation information from database
- Accountant (doing)
    - Calculate customer payment for each garage visit
- GateOperator (doing)
    - Operate the exit gate to raise/lower it
- Reservation (doing)
    - Creates reservation based on data customer inputs on website

Association Definitions

- Controller – LicensePlate: obtain

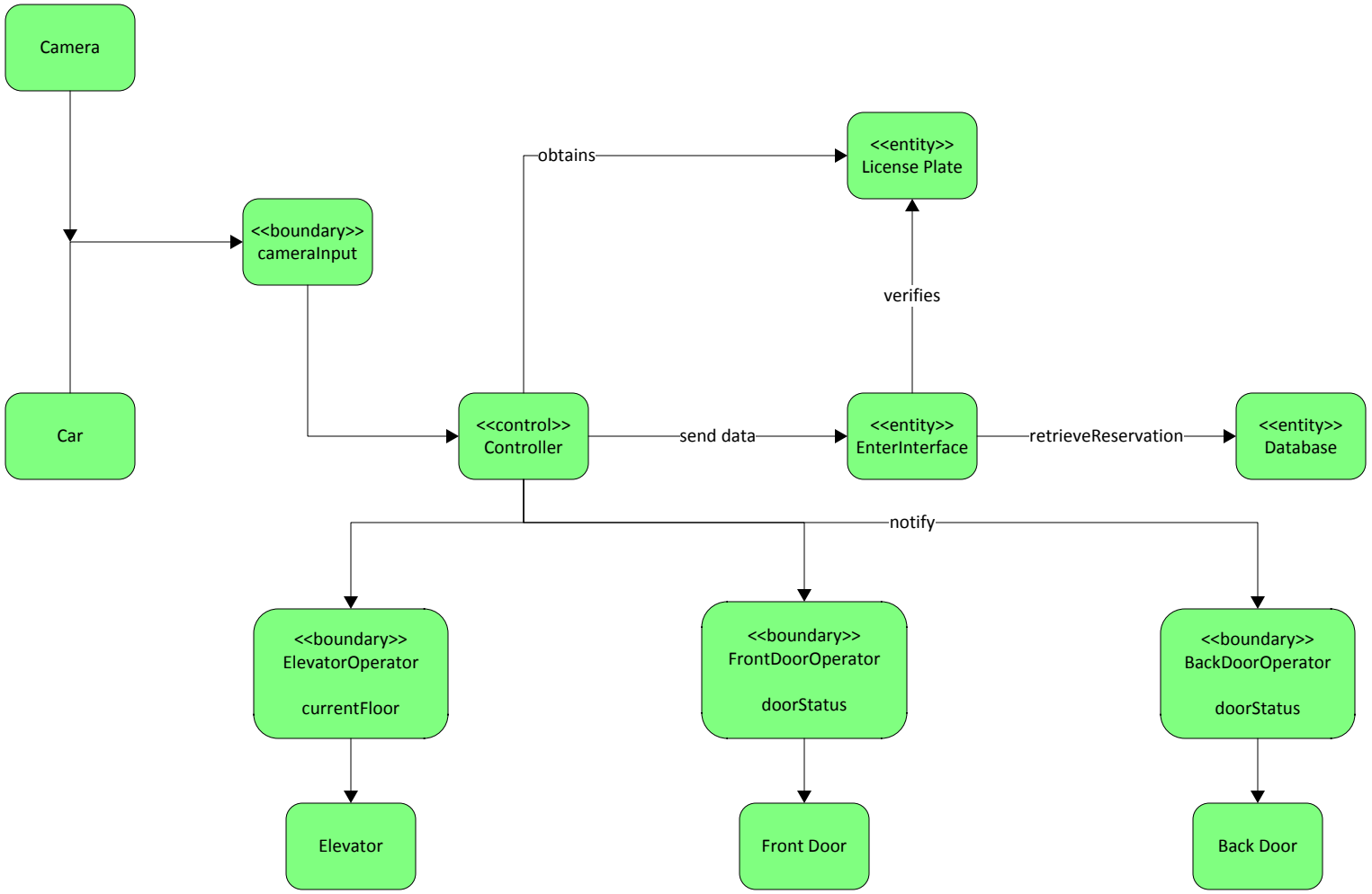
    - Controller obtains LicensePlate from CameraInput
- Controller – EnterInterface: send data
    - Controller sends LicensePlate to EnterInterface
- EnterInterface – LicensePlate: verifies
    - EnterInterface verifies LicensePlate with Database
- EnterInterface – Database: retrieveReservation
    - EnterInterface retrieves Reservation from Database
- Controller – ElevatorOperator: notify
    - Controller notifies ElevatorOperator to send elevator to appropriate floor
- Controller – FrontDoorOperator: notify
    - Controller notifies FrontDoorOperator to open/close front door of elevator
- Controller – BackDoorOperator: notify
    - Controller notifies BackDoorOperator to open/close back door of elevator
- Controller – ExitInterface: send data
    - Controller seds LicensePlate to ExitInterface
- ExitInterface – LicensePlate: verifies
    - ExitInterface verifies LicensePlate with Database
- ExitInterface – Database: retrieveReservation
    - ExitInterface retrieves Reservation from Database
- ExitInterface – Accountant: send data
    - ExitInterface sends reservation data to Accountant and prompts calculation
- Controller – GateOperator: notify
    - Controller notifies GateOperator to raise/lower the exit gate
- Controller – Database: update
    - Controller updates Database with parking space occupancy or reservation info

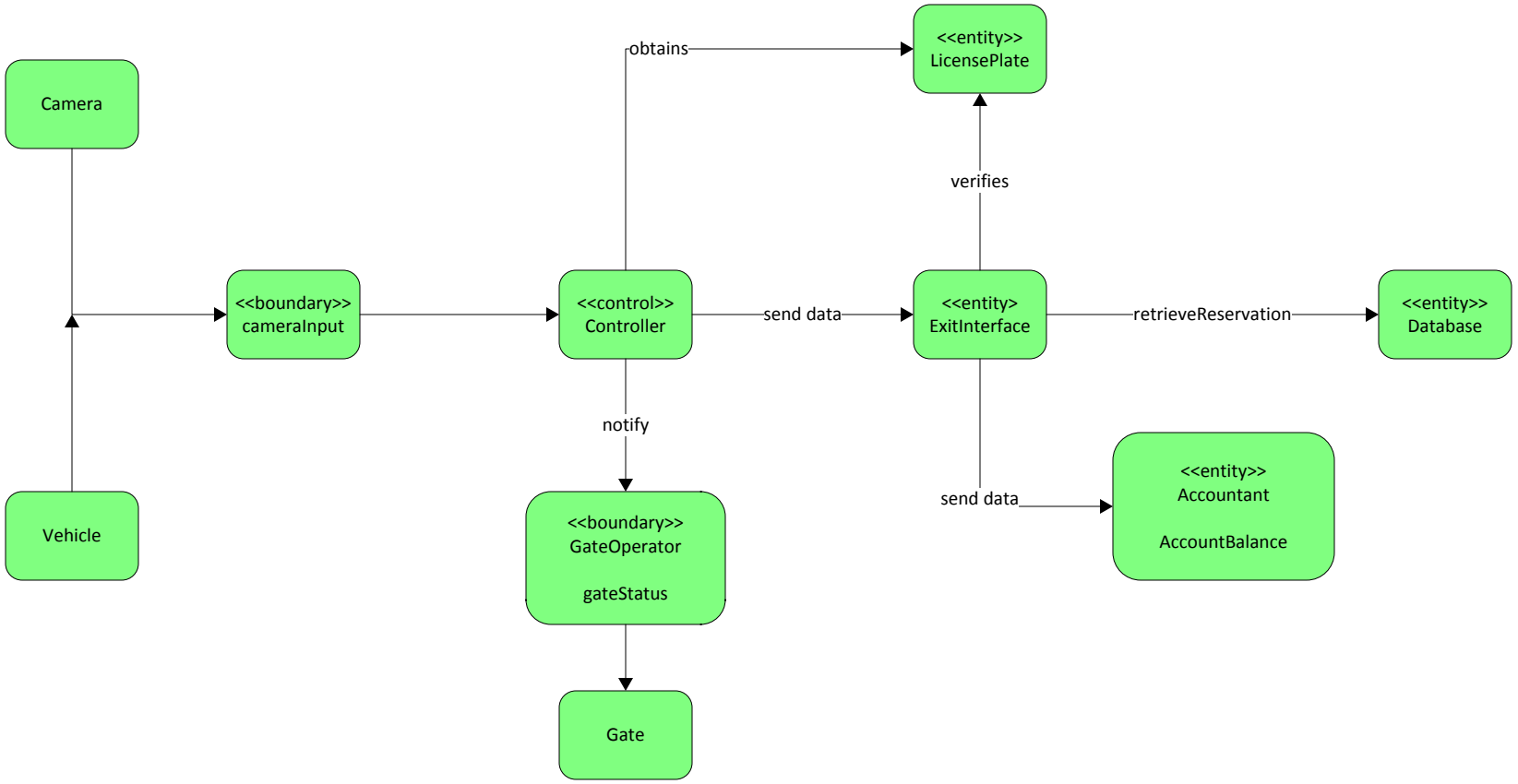
Attribute Definitions

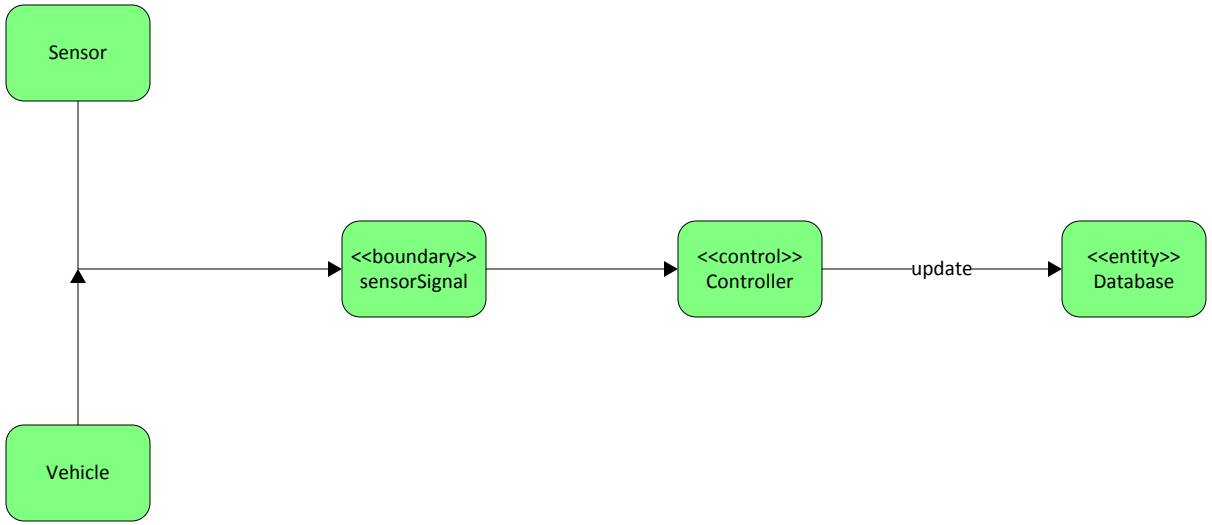
- LicensePlate:
    - lp: String of license number characters from camera input
- FrontDoorOperator:
    - status: String that states if front elevator door is currently open or closed
- BackDoorOperator:

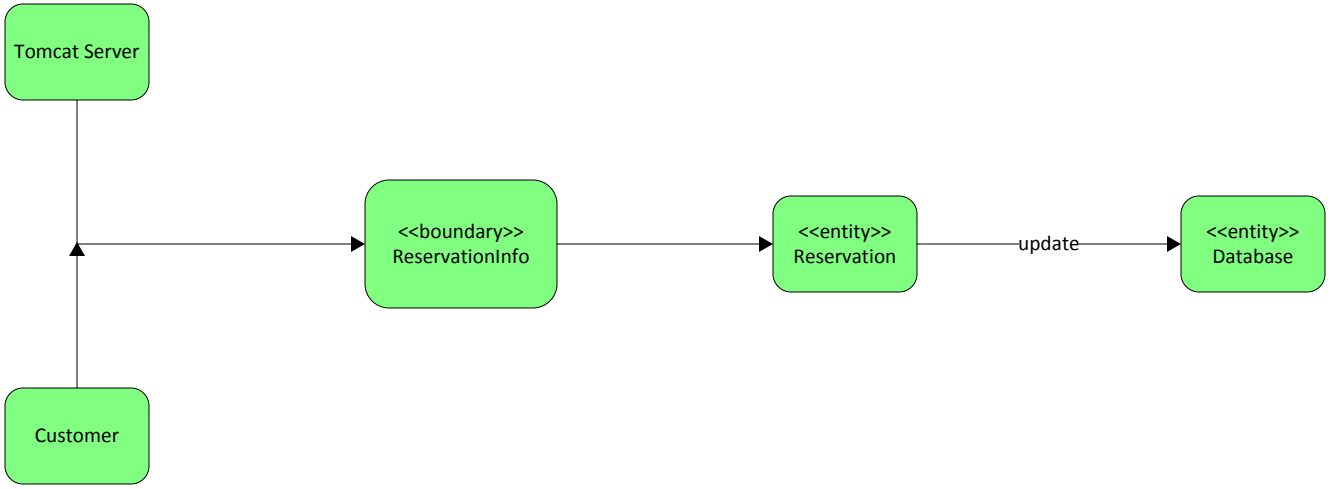
    - status: String that states if back elevator door is currently open or closed
- ElevatorOperator:
    - floor: integer value of floor elevator currently is on
    - occupied: boolean value of whether elevator is occupied or not
- Accountant:
    - accountBalance: double value of how much the customer currently owes
- GateOperator:
    - status: String that states if gate is currently raised or lowered
- Reservation:
    - reservationTimeIn: calendar object of start time of reservation
    - reservationTimeOut: calendar object of end time of reservation

Domain Model UML Diagrams


Camera
<<boundary>>
cameraInput
Car
obtains
<<entity>>
License Plate
verifies
<<control>>
Controller
send data
<<entity>>
EnterInterface
retrieveReservation
<<entity>>
Database
notify
<<boundary>>
ElevatorOperator
currentFloor
<<boundary>>
FrontDoorOperator
doorStatus
<<boundary>>
BackDoorOperator
doorStatus
Elevator
Front Door
Back Door


Camera
Vehicle
<<boundary>>
cameraInput
<<control>>
Controller
obtains
<<entity>>
LicensePlate
verifies
send data
<<entity>
ExitInterface
retrieveReservation
<<entity>>
Database
notify
<<boundary>>
GateOperator
gateStatus
Gate
send data
<<entity>>
Accountant
AccountBalance


Sensor
<<boundary>>
sensorSignal
<<control>>
Controller
update
<<entity>>
Database
Vehicle


Tomcat Server
<<boundary>>
ReservationInfo
<<entity>>
Reservation
update
<<entity>>
Database
Customer

System Operation Contracts:

- updateReservation
    - Preconditions
        - All parking not occupied between reservationTimeIn and reservationTimeOut
        - Customer is registered with database
    - Postconditions
        - None
- useElevator
    - Preconditions
        - frontDoor.status = open
        - backDoor.status = closed
        - floor = 1
        - occupied = false
    - Postconditions
        - frontDoor.status = open
        - backDoor.status = closed
        - floor = 1
        - occupied = false
- openGate
    - Preconditions
        - gate.status = closed
    - Postconditions
        - gate.status = closed
- payFee
    - Preconditions
        - accountBalance ≥ 0
        - gate.status = closed
    - Postconditions
        - accountBalance > 0
        - gate.status = open
- spaceOccupied
    - Preconditions

- None
- Postconditions
  - None

Mathematical Model

In order to facilitate the practice of overbooking, the system should have an occupancy management formula which is used in order to achieve 100% occupancy within the parking garage. Thus, the parking garage can take more reservations than parking spaces ahead of schedule with the knowledge that a certain number of reservations will be unfulfilled and that a certain number of customers will approach the garage with walk-in parking. The formula accounts for reservations of all types, unfulfilled reservations, overtime parking, undertime parking, and walk-in customers. It is:

Total number of open parking spaces

- Current reservations × Percentage unfulfilled
- Predicted overtime parking
- Predicted walk-ins
+ Predicted undertime parking
= Number of parking spaces available

The predictions for unfulfilled reservations, overtime parking, walk-ins, and undertime parking will be made based on prior experience collected from the parking garage database.

Interaction Diagrams

useElevator

The useElevator use case employs the Command pattern quite heavily throughout its implementation. Almost all communication and transactions that occur within this use case occur involve the controller object in some way or form. The controller is constantly checking to see if the camera object has read a license plate. When a license plate is read, the controller immediately retrieves the value and sends it to the user interface object. Instead of the camera directly forwarding the information to the user interface, the controller object is appointed to the task. Furthermore, when the user interface gets confirmation from the customer, it is the controller which prompts the elevator and its doors to operate. Again, the controller acts as a mediator between two sets of objects. The usage of the controller class within this use case reduces the workload on sensitive objects such as the camera and the user interface. Both of these objects are constantly searching for external input so tasking them with additional operations could easily overload them. The controller object takes care of any communication that should occur between objects such as these.

In addition, this use case implements the Proxy pattern with its database. All communication between the Java objects and the SQL database occur through the database class. This class contains the functions required for information to flow to and from the database. Without the database class, the Java classes and the database would not be able to communicate with each other and information could not be saved or accessed.

openGate

The openGate use case utilizes both the Command and Proxy patterns much like the useElevator use case. Most of the activity within the use case occurs via the controller object, who coordinates communication between the camera and the user interface and database. Furthermore, the controller prompts the gate object to open and close when necessary. The Proxy pattern is used within the database class. In order to access the SQL database, the database class is used as a proxy between the other classes and the database. Without this component, communication with the database would not be possible.

payFee

The payFee use case employs the Publisher-Subscriber pattern. When the exit interface needs to inform objects of updates to the payment information, it does so by publishing the necessary data. The

subscribers, such as the accountant and database objects, simply pick up the published data and format it to meet their specifications. This pattern allows the interface to output all necessary information to all parties with a single function call instead of calling each object individually. This method allows for greater efficiency within this use case.

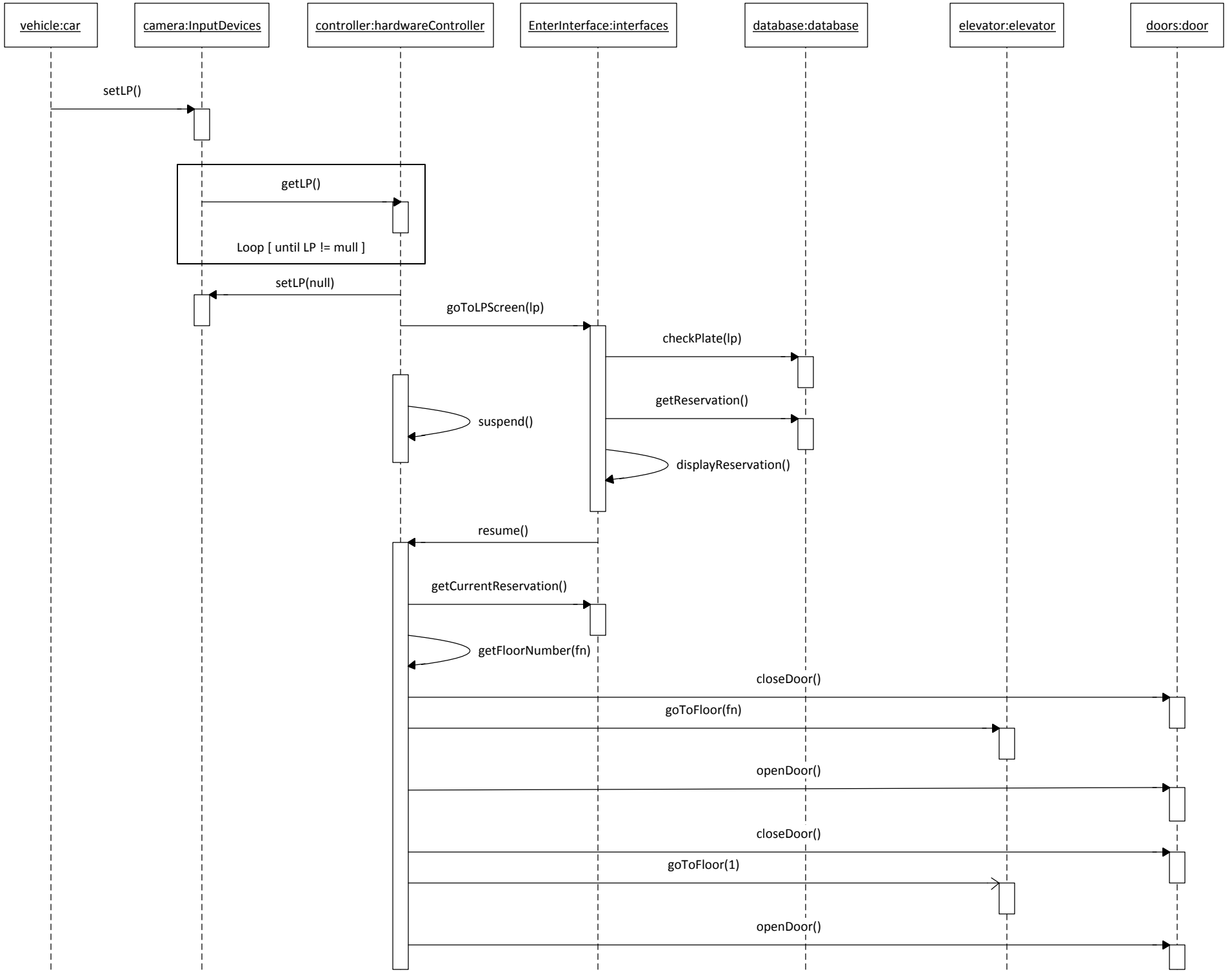
checkSpaceAvailability

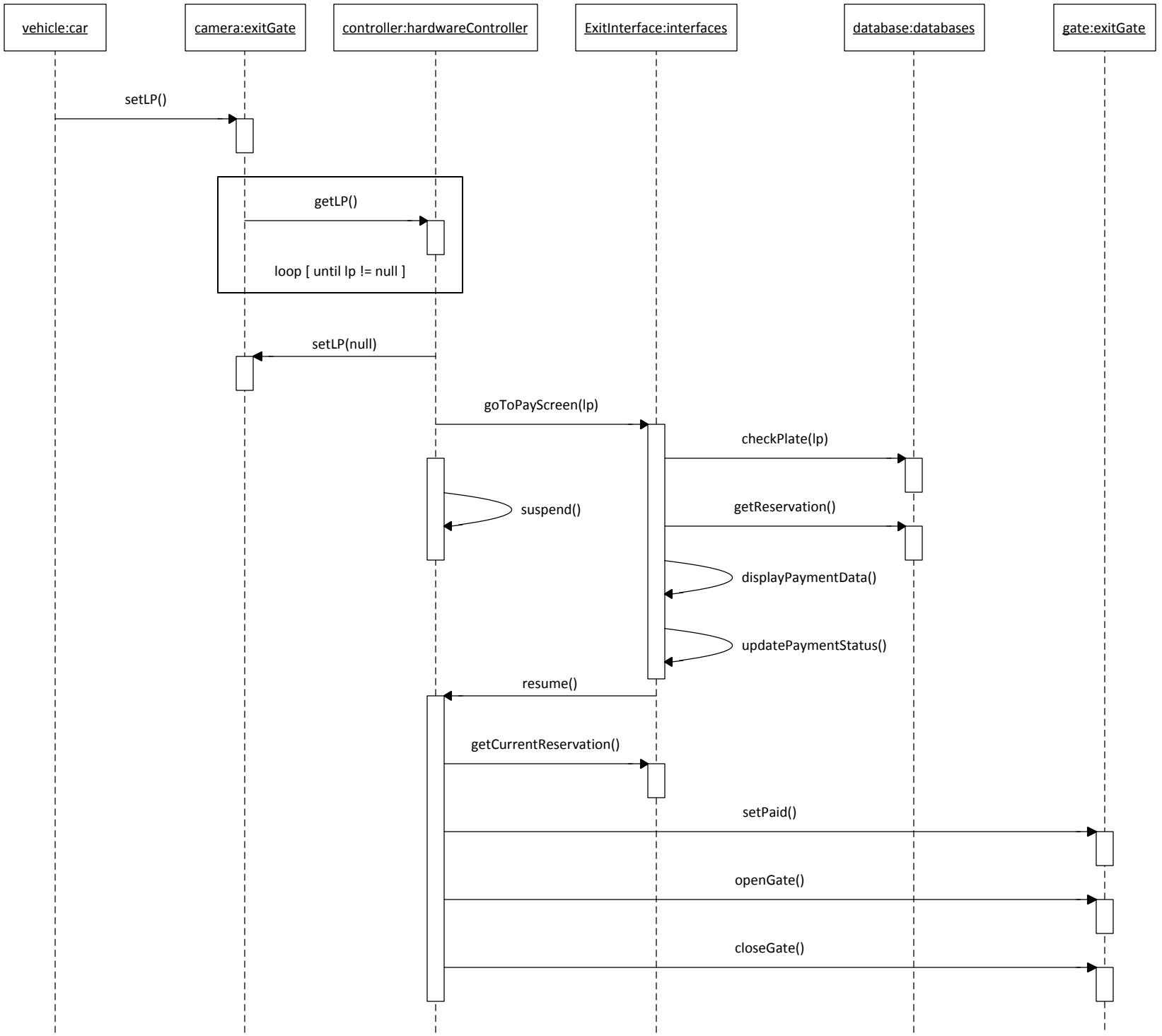
The checkSpaceAvailability use case is a fairly straightforward case. It utilizes the Command pattern in that the sensor object routes information through the controller in order to send it to the database. Using a controller object as a middle-man allows the sensor object to focus on its task instead of interfacing with the database as well.

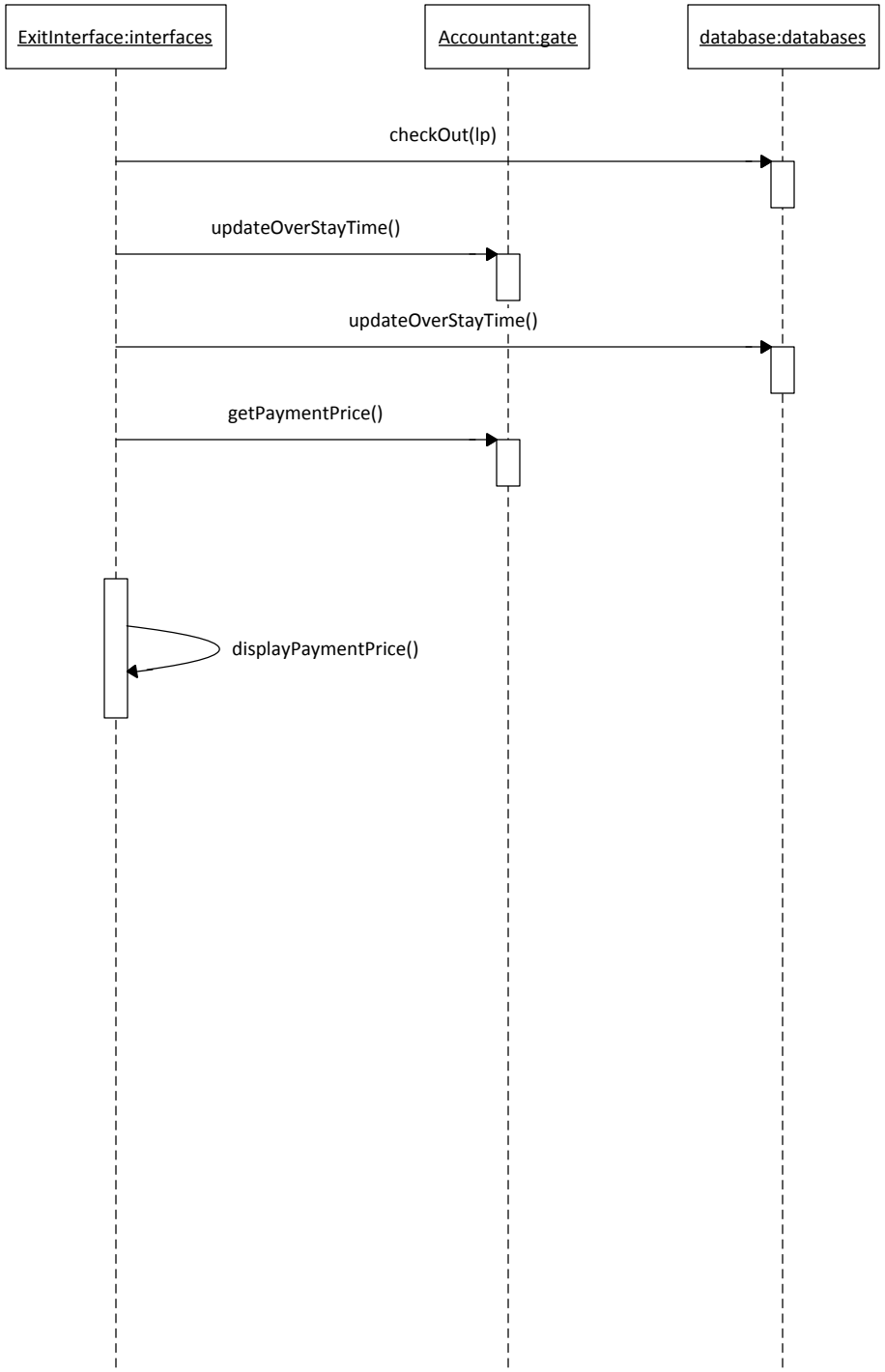
updateReservations

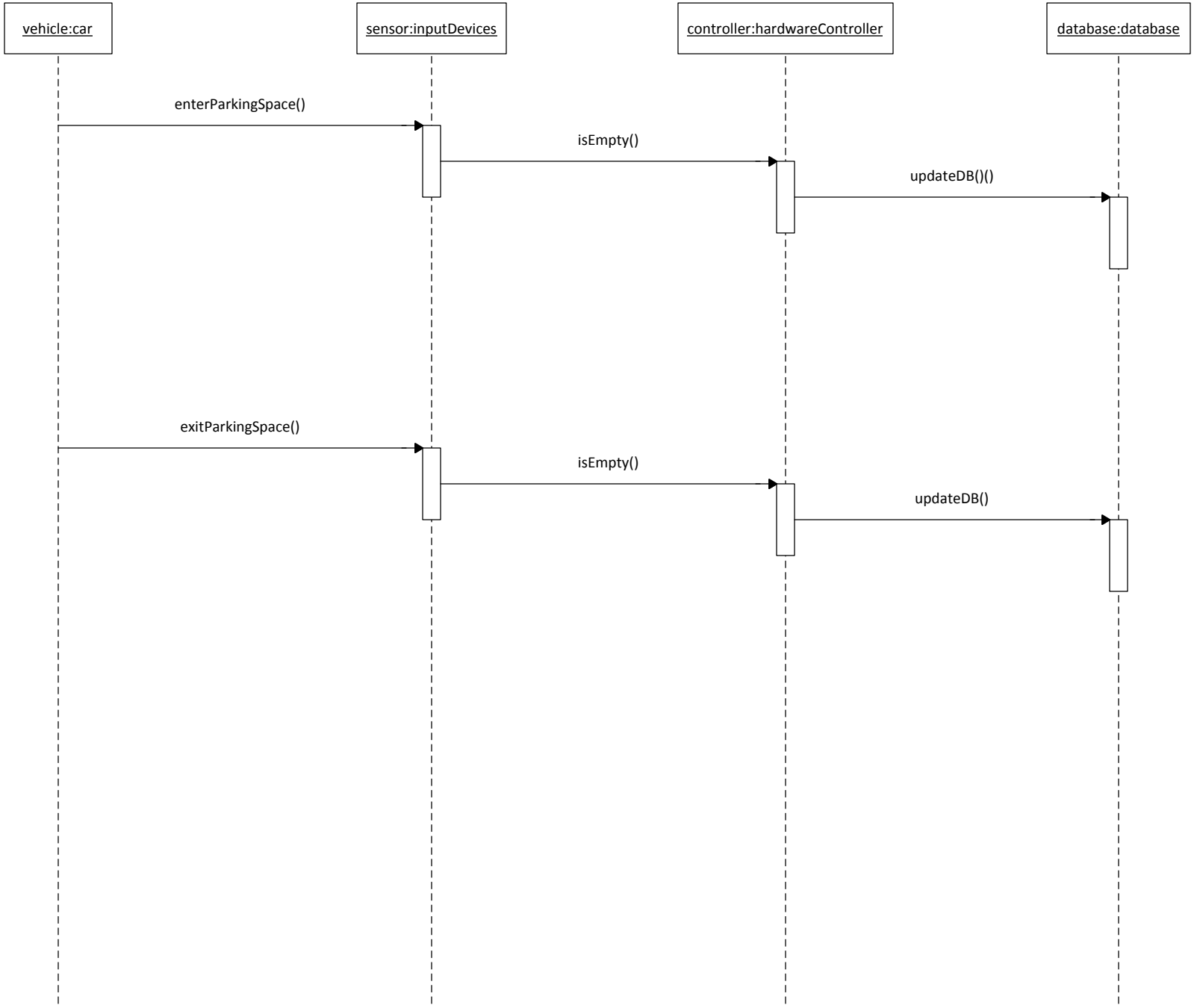
The updateReservations use case employs the Proxy pattern in order to interface with the Tomcat server on which the website runs. The Reservations class serves as the proxy between the other Java objects and the Tomcat server since the Java objects do not have the functionality to access the website themselves.

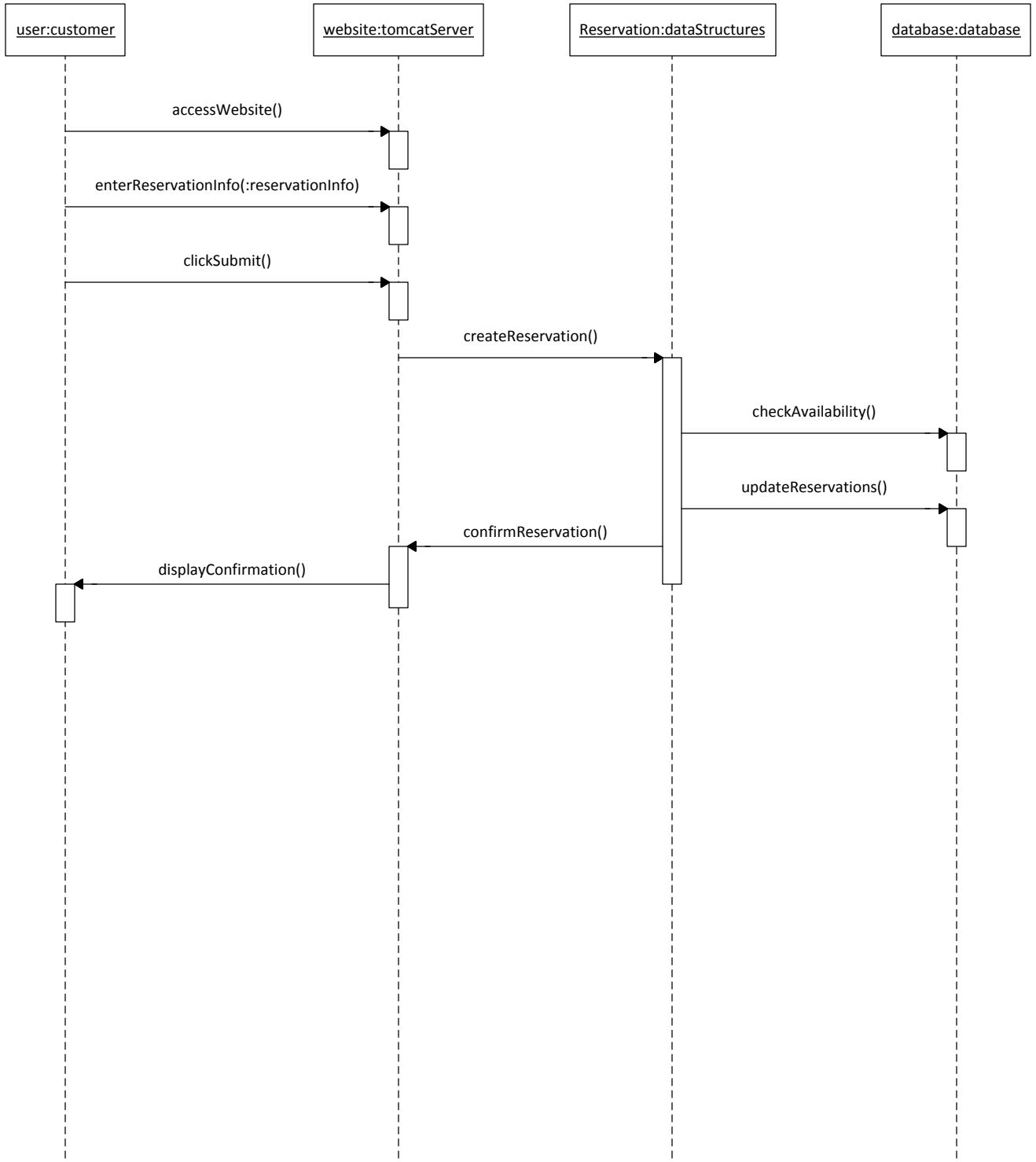
UML Interaction Diagrams


vehicle:car
camera:InputDevices
controller:hardwareController
EnterInterface:interfaces
database:database
elevator:elevator
doors:door
setLP()
getLP()
Loop [until LP != mull]
setLP(null)
goToLPScreen(lp)
checkPlate(lp)
getReservation()
suspend()
displayReservation()
resume()
getCurrentReservation()
getFloorNumber(fn)
closeDoor()
goToFloor(fn)
openDoor()
closeDoor()
goToFloor(1)
openDoor()

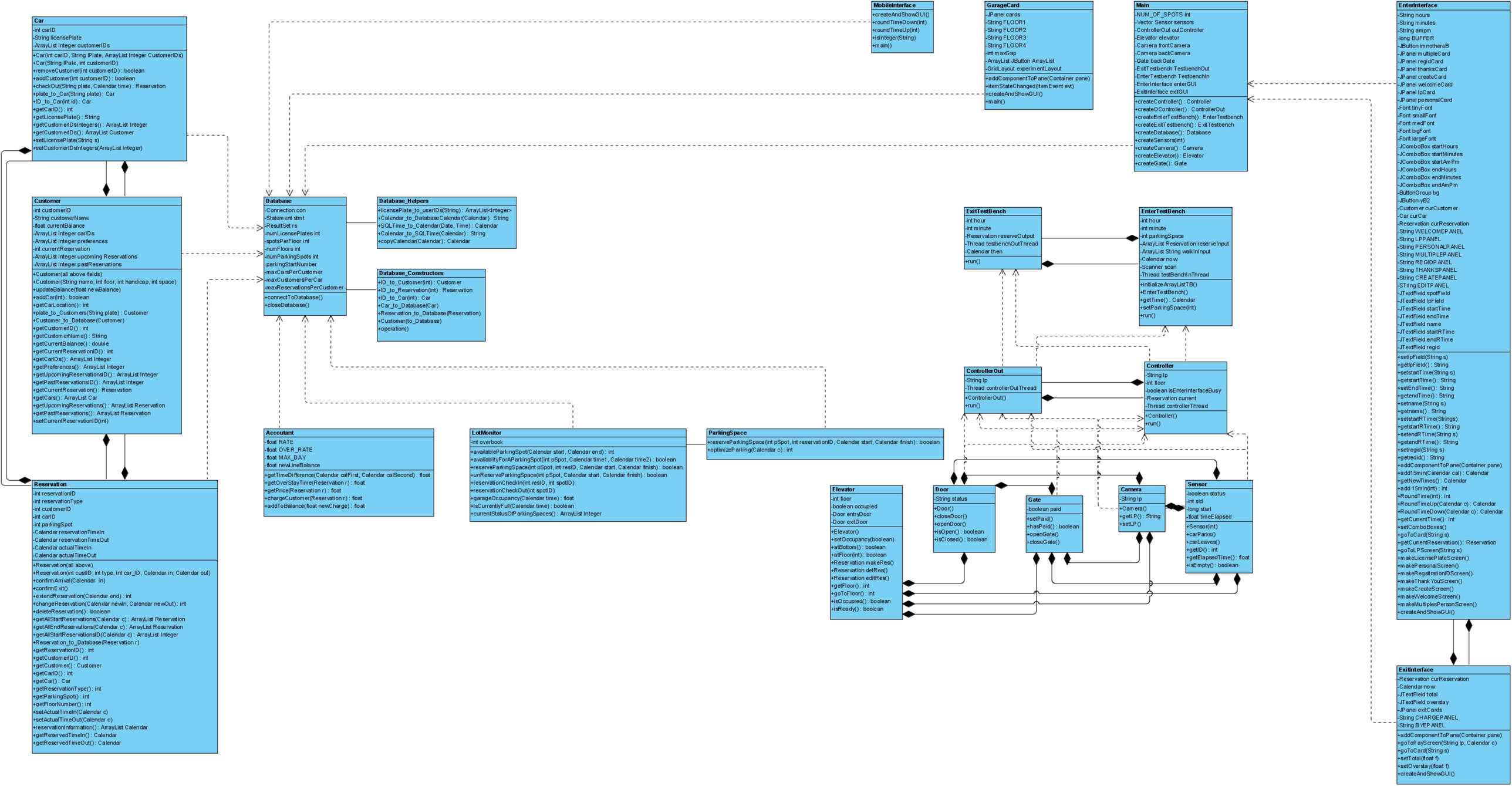

vehicle:car
camera:exitGate
controller:hardwareController
ExitInterface:interfaces
database:databases
gate:exitGate
setLP()
getLP()
loop [until lp != null]
setLP(null)
goToPayScreen(lp)
checkPlate(lp)
suspend()
getReservation()
displayPaymentData()
updatePaymentStatus()
resume()
getCurrentReservation()
setPaid()
openGate()
closeGate()


ExitInterface:interfaces
Accountant:gate
database:databases
checkOut(lp)
updateOverStayTime()
updateOverStayTime()
getPaymentPrice()
displayPaymentPrice()


vehicle:car
sensor:inputDevices
controller:hardwareController
database:database
enterParkingSpace()
isEmpty()
updateDB()()
exitParkingSpace()
isEmpty()
updateDB()


user:customer
website:tomcatServer
Reservation:dataStructures
database:database
accessWebsite()
enterReservationInfo(:reservationInfo)
clickSubmit()
createReservation()
checkAvailability()
updateReservations()
confirmReservation()
displayConfirmation()

Class Diagram


Car
-int carID
-String licensePlate
-ArrayList Integer customerIDs
+Car(int carID, String lPlate, ArrayList Integer CustomerIDs)
+Car(String lPate, int customerID)
+removeCustomer(int customerID) : boolean
+addCustomer(int customerID) : boolean
+checkOut(String plate, Calendar time) : Reservation
+plate_to_Car(String plate) : Car
+ID_to_Car(int id) : Car
+getCarID() : int
+getLicensePlate() : String
+getCustomerIDsIntegers() : ArrayList Integer
+getCustomerIDs() : ArrayList Customer
+setLicensePlate(String s)
+setCustomerIDsIntegers(ArrayList Integer)
MobileInterface
+createAndShowGUI()
+roundTimeDown(int)
+roundTimeUp(int)
+isInteger(String)
+main()
GarageCard
-JPanel cards
-String FLOOR1
-String FLOOR2
-String FLOOR3
-String FLOOR4
-int maxGap
-ArrayList JButton ArrayList
-GridLayout experimentLayout
+addComponentToPane(Container pane)
+itemStateChanged(ItemEvent evt)
+createAndShowGUI()
+main()
Main
-NUM_OF_SPOTS int
-Vector Sensor sensors
-ControllerOut outController
-Elevator elevator
-Camera frontCamera
-Camera backCamera
-Gate backGate
-ExitTestbench TestbenchOut
-EnterTestbench TestbenchIn
-EnterInterface enterGUI
-ExitInterface exitGUI
+createController() : Controller
+createOController() : ControllerOut
+createEnterTestBench() : EnterTestbench
+createExitTestbench() : ExitTestbench
+createDatabase() : Database
+createSensors(int)
+createCamera() : Camera
+createElevator() : Elevator
+createGate() : Gate
EnterInterface
-String hours
-String minutes
-String ampm
-long BUFFER
-JButton imnothereB
-JPanel multipleCard
-JPanel regidCard
-JPanel thanksCard
-JPanel createCard
-JPanel welcomeCard
-JPanel lpCard
-JPanel personalCard
-Font tinyFont
-Font smallFont
-Font medFont
-Font bigFont
-Font largeFont
-JComboBox startHours
-JComboBox startMinutes
-JComboBox startAmPm
-JComboBox endHours
-JComboBox endMinutes
-JComboBox endAmPm
-ButtonGroup bg
-JButton yB2
-Customer curCustomer
-Car curCar
-Reservation curReservation
-String WELCOMEPANEL
-String LPPANEL
-String PERSONALPANEL
-String MULTIPLEPANEL
-String REGIDPANEL
-String THANKSPANEL
-String CREATEPANEL
-STring EDITPANEL
-JTextField spotField
-JTextField lpField
-JTextField startTime
-JTextField endTime
-JTextField name
-JTextField startRTime
-JTextField endRTime
-JTextField regid
+setlpField(String s)
+getlpField() : String
+setstartTime(String s)
+getstartTime() : String
+setEndTime() : String
+getendTime() : String
+setname(String s)
+getname() : String
+setstartRTime(String s)
+getstartRTime() : String
+setendRTime(String s)
+getendRTime() : String
+setregid(String s)
+getregid() : String
+addComponentToPane(Container pane)
+add15min(Calendar cal) : Calendar
+getNewTimes() : Calendar
+add 15min(int) : int
+RoundTime(int) : int
+RoundTimeUp(Calendar c) : Calendar
+RoundTimeDown(Calendar c) : Calendar
+getCurrentTime() : int
+setComboBoxes()
+goToCard(String s)
+getCurrentReservation() : Reservation
+goToLPScreen(String s)
+makeLicensePlateScreen()
+makePersonalScreen()
+makeRegistrationIDScreen()
+makeThankYouScreen()
+makeCreateScreen()
+makeWelcomeScreen()
+makeMultiplesPersonScreen()
+createAndShowGUI()
Customer
-int customerID
-String customerName
-float currentBalance
-ArrayList Integer carIDs
-ArrayList Integer preferences
-int currentReservation
-ArrayList Integer upcoming Reservations
-ArrayList Integer pastReservations
+Customer(all above fields)
+Customer(String name, int floor, int handicap, int space)
+updateBalance(float newBalance)
+addCar(int) : boolean
+getCarLocation() : int
+plate_to_Customers(String plate) : Customer
+Customer_to_Database(Customer)
+getCustomerID() : int
+getCustomerName() : String
+getCurrentBalance() : double
+getCurrentReservationID() : int
+getCarIDs() : ArrayList Integer
+getPreferences() : ArrayList Integer
+getUpcomingReservationsID() : ArrayList Integer
+getPastReservationsID() : ArrayList Integer
+getCurrentReservation() : Reservation
+getCars() : ArrayList Car
+getUpcomingReservations() : ArrayList Reservation
+getPastReservations() : ArrayList Reservation
+setCurrentReservationID(int)
Database
-Connection con
-Statement stmt
-ResultSet rs
-numLicensePlates int
-spotsPerFloor int
-numFloors int
-numParkingSpots int
-parkingStartNumber
-maxCarsPerCustomer
-maxCustomersPerCar
-maxReservationsPerCustomer
+connectToDatabase()
+closeDatabase()
Database_Helpers
+licensePlate_to_userIDs(String) : ArrayList<Integer>
+Calendar_to_DatabaseCalendar(Calendar) : String
+SQLTime_to_Calendar(Date, Time) : Calendar
+Calendar_to_SQLTime(Calendar) : String
+copyCalendar(Calendar) : Calendar
Database_Constructors
+ID_to_Customer(int) : Customer
+ID_to_Reservation(int) : Reservation
+ID_to_Car(int) : Car
+Car_to_Database(Car)
+Reservation_to_Database(Reservation)
+Customer(to_Database)
+operation()
ExitTestBench
-int hour
-int minute
-Reservation reserveOutput
-Thread testbenchOutThread
-Calendar then
+run()
EnterTestBench
-int hour
-int minute
-int parkingSpace
-ArrayList Reservation reserveInput
-ArrayList String walkInInput
-Calendar now
-Scanner scan
-Thread testBenchInThread
+initializeArrayListTB()
+EnterTestBench()
+getTime() : Calendar
+setParkingSpace(int)
+run()
ControllerOut
-String lp
-Thread controllerOutThread
+ControllerOut()
+run()
Controller
-String lp
-int floor
-boolean isEnterInterfaceBusy
-Reservation current
-Thread controllerThread
+Controller()
+run()
Reservation
-int reservationID
-int reservationType
-int customerID
-int carID
-int parkingSpot
-Calendar reservationTimeIn
-Calendar reservationTimeOut
-Calendar actualTimeIn
-Calendar actualTimeOut
+Reservation(all above)
+Reservation(int custID, int type, int car_ID, Calendar in, Calendar out)
+confirmArrival(Calendar in)
+confirmExit()
+extendReservation(Calendar end) : int
+changeReservation(Calendar newIn, Calendar newOut) : int
+deleteReservation() : boolean
+getAllStartReservations(Calendar c) : ArrayList Reservation
+getAllEndReservations(Calendar c) : ArrayList Reservation
+getAllStartReservationsID(Calendar c) : ArrayList Integer
+Reservation_to_Database(Reservation r)
+getReservationID() : int
+getCustomerID() : int
+getCustomer() : Customer
+getCarID() : int
+getCar() : Car
+getReservationType() : int
+getParkingSpot() : int
+getFloorNumber() : int
+setActualTimeIn(Calendar c)
+setActualTimeOut(Calendar c)
+reservationInformation() : ArrayList Calendar
+getReservedTimeIn() : Calendar
+getReservedTimeOut() : Calendar
Accoutant
-float RATE
-float OVER_RATE
-float MAX_DAY
-float newLineBalance
+getTimeDifference(Calendar calFirst, Calendar calSecond) : float
+getOverStayTime(Reservation r) : float
+getPrice(Reservation r) : float
+chargeCustomer(Reservation r) : float
+addToBalance(float newCharge) : float
LotMonitor
-int overbook
+availableParkingSpot(Calendar start, Calendar end) : int
+availablityForAParkingSpot(int pSpot, Calendar time1, Calendar time2) : boolean
+reserveParkingSpace(int pSpot, int resID, Calendar start, Calendar finish) : boolean
+unReserveParkingSpace(int pSpot, Calendar start, Calendar finish) : boolean
+reservationCheckIn(int resID, int spotID)
+reservationCheckOut(int spotID)
+garageOccupancy(Calendar time) : float
+isCurrentlyFull(Calendar time) : boolean
+currentStatusOfParkingSpaces() : ArrayList Integer
ParkingSpace
+reserveParkingSpace(int pSpot, int reservationID, Calendar start, Calendar finish) : boolean
+optimizeParking(Calendar c) : int
Elevator
-int floor
-boolean occupied
-Door entryDoor
-Door exitDoor
+Elevator()
+setOccupancy(boolean)
+atBottom() : boolean
+atFloor(int) : boolean
+Reservation makeRes()
+Reservation delRes()
+Reservation editRes()
+getFloor() : int
+goToFloor() : int
+isOccupied() : boolean
+isReady() : boolean
Door
-String status
+Door()
+closeDoor()
+openDoor()
+isOpen() : boolean
+isClosed() : boolean
Gate
-boolean paid
+setPaid()
+hasPaid() : boolean
+openGate()
+closeGate()
Camera
-String lp
+Camera()
+getLP() : String
+setLP()
Sensor
-boolean status
-int sid
-long start
-float timeElapsed
+Sensor(int)
+carParks()
+carLeaves()
+getID() : int
+getElapsedTime() : float
+isEmpty() : boolean
ExitInterface
-Reservation curReservation
-Calendar now
-JTextField total
-JTextField overstay
-JPanel exitCards
-String CHARGEPANEL
-String BYEPANEL
+addComponentToPane(Container pane)
+goToPayScreen(String lp, Calendar c)
+goToCard(String s)
+setTotal(float f)
+setOverstay(float f)
+createAndShowGUI()

Interface Specification

Due to the large amount of functions and variables used throughout the project and due to the relative ease of determining what the function or variable is meant to do by the name of it, not all functions or variables will be described below. They are documented with comments in our code for additional assistance along with all functions or variables being included in the Class Diagram that follows this Interface Specification. Important or difficult to understand variables and functions will be discussed while most of the variables for the GUI's and functions that serve primarily has helpers will be ignored. Once again, consult the source code or the Class Diagram for the most detail explanation of these functions or variables.

Database

- Dependencies
  - Everything depends on this class. Without this class, nothing can run.
- Variables
  - Connection con, Statement stmt, ResultSet rs
    - These are the ways to connect to the database
  - ints spotsPerFloor, numFloors, numParkingSpots
    - The parameters of the garage
  - int parkingStartNumber
    - used for an even distribution of parking spaces
  - int numLicensePlates,maxCarsPerCustomer,maxCustomersPerCar,maxReservationsPerCustomer
    - These are the restrictions used in the database
- Functions
  - connectToDatabase()
    - Opens connection with login
  - closeDatabase()
    - Closes connection

Database_Constructors

- Dependencies
  - Database
    - This class needs to access the database to retrieve and update values.
  - Customer, Cars, Reservation

- This class creates the above objects by accessing the database or updates the database given one of these three objects.

- Variables
  - n/a
- Functions
  - ID_to_Customer, ID_to_Reservation, ID_to_Car
    - These functions create their respective object by calling the database with the known identification number.
  - Reservation_to_Database, Car_to_Database, Customer_to_Database
    - These functions take their respective edited object and update all of its fields to the Database. These functions make the amount of MySQL commands to write at a minimum

Database Helpers

- Dependencies
  - Independent of the other classes.
- Variables
  - n/a
- Functions
  - Calendar_to_SQLTime, Calendar_to_DatabaseCalendar, SQLTime_to_Calendar
    - Since a Calendar object is used for date and time, these functions convert to or from the calendar object to SQLTime or to the columns in the Database
  - incrementDateColumn
    - Used to traverse the parkingspaces database by changing the column name to the next

ParkingSpace

- Dependencies
  - Database
    - This class needs the database for receiving and storing data. This parkingspaces tables hold the availability of all the parking spots.
- Variables
  - n/a
- Functions
  - reserveParkingSpace

- Performs the proper operations to reserve a parking by writing to the parkingspaces database

Car

- Dependencies
  - Database
    - This class needs the database for receiving and storing data. The Cars table holds the information related to all cars.
  - Database_Constuctors
    - For updating an entire Customer to downloading an entire customer, this class is critical for easy updating and downloading of Customer objects.
  - Customer
    - Variables in the car class store references to the Customer class. The important information stored in the Database is more of less stored related to customers and not to cars.
- Variables
  - Int carID
    - How the database and all other objects can reference this object and get its information. This is unique to this car.
  - String licensePlate
    - each car has this unique identifier
  - ArrayList<Integer> customerIDs
    - List of all the customers associated with this car
- Functions
  - Car(all variables)
    - Creates a Car object if all of its fields are known. Used by the database
  - Car (String lPlate, int customerID)
    - Creates a car object and links it to a customer. This handles accesses to the database.
  - removeCustomer (int id), addCustomer (int id)
    - links or unlinks a customer that is associated with this car
  - Reservation checkout(String plate, Calendar time)
    - When a car is about to leave, the database receives its plate and Calendar object and after updating the proper fields of the reservation and parkingspaces databases, returns the Reservation object so that the customer can confirm payment

Customer

- Dependencies
  - Database
    - The Database holds all information related to all the Customers. The Customer class can create Customer objects from the Database or it can upload changed information to the database.
  - Database_Constuctors
    - For updating an entire Customer to downloading an entire customer, this class is critical for easy updating and downloading of Customer objects.
  - Reservations
    - The important aspect of a parking garage is that reservations can be made. The Customer class stores only the reservation ID numbers, so it depends on the Reservation class to store all information related to the reservation (i.e. times and parking spots)
  - Cars
    - LicensePlate information is critical to identifying a car to a customer. The Cars class stores all the license plates and who is related to those cars.
  - Accountant
    - In order to keep track of the balance of each customer, the Accountant class is needed to calculate the time a car was in the parking garage and then to write it to the Customer class.
- Variables
  - Int customerID
    - How the database and all other objects can reference this object and get its information. This is unique to this customer.
  - String CustomerName
    - Name of this customer
  - Float currentBalance
    - Maintained by the accountant but accessed when this customer wishes to maintain his account
  - ArrayList<Integer> carIDs
    - All the ID's of cars that are associated with this customer
  - ArrayList<Integer> preferences

- For future use, this will be accessed when finding parking spaces to try to find the best parking space for a customer

➢ Int currentReservation, ArrayList<Integer> upcomingReservations, ArrayList<Integer> pastReservation

- Contains ID links to reservations that are associated with this customer. By having reservations, the amount of variables are reduced for each customer.

❖ Functions

➢ Customer(all above)

- Creates a Customer object if all of its fields are known. Used by the database

➢ Customer(String name, int floor, int handicap, int spot)

- Creates a Customer and updates the database for this customer. Takes their name and their preferences.

➢ updateBalance(float newBalance)

- Used by the accountant to keep track of how much each customer owes

➢ addCar(int carID)

- links a car to a customer and a customer to a car

➢ ArrayList<Customer> plate_to_Customers(String lPlate)

- When a car arrives at the parking garage, since multiple people can be associated with a car, this functions returns all those customers that are associated with the car that has pulled up

Reservation

❖ Dependencies

➢ Database

- The database holds all the reservations and the associated information with it across sessions. With no database, there would be no reservations.

➢ Database_Helpers

- This class continues critical functions to traverse the database for making or viewing reservations.

➢ Database_Constuctors

- For updating an entire Reservation to downloading an entire customer, this class is critical for easy updating and downloading of Reservation objects.

➢ Lot Monitor & Parking Spaces

        - The Reservation class needs to use the Lot Monitor in order to check if reservations can be made. These classes also help save the parking space from being double booked.
    - Customer
        - All reservations are linked to a customer and each customer has a list of reservations. They are used together to reduce the overhead in each of the classes. They are mutually dependent.
    - Car
        - For each reservation, a car is part of the reservation. Therefore, each Reservation is dependent of the Car Class.
- Variables
    - Int reservationID
        - How the database and all other objects can reference this object and get its information. This is unique to this reservation.
    - Int reservationtType
        - To be implemented later, this can be used to store if this reservation is a walk-in, walk-in registered, registered reservation, or a multi-day reservation
    - Ints customerID, carID, parkingSpot
        - These are the ID's of the customer, car, and parkingspot associated with this reservation. This makes it simple to access these components of the system.
    - Calendars reserervationTimeIn, reservationTimeOut, actualTimeIn, actualTimeOut
        - These Calendar objects are used to keep track of when the customer wants to arrive/leave and when they actually do arrive and leave. This is used as a quick access point to know this information about a customer instead of accessing and searching the database to learn this.
- Functions
    - Reservation (all above)
        - Used by the database to create an entire Reservation object
    - Reservation (int customerID, int type, int carID, Calendar in, Calendar out)
        - The most basic information required to make a reservation. If a reservation can be made, this function creates it and returns the reservation. If it cannot be made, the reservationID of the returned Reservation object is -1
    - confirmArrival and confirmExit()
        - updates the actualTimeIn/out field of the Reservation Object

- extendReservation
  - if any (current or future) reservation wants to be extended, this checks the database if it can be extended and then either extends the reservation or returns an error
- changeReservation
  - if a reservation wants to be completely changed, this is called instead of extend reservation in order to delete the current reservation and return a new reservation of the customer's liking
- deleteReservation
  - removes all references across objects and the database to a reservation in the future
- ArrayList<Integer> getAllStart/End/Current Reservations (Calendar c)
  - Used primalrily by the testbench to return all the reservationID that are starting at a given time, ending at the time, or are currently within the parkingspaces table

LotMonitor

- Dependencies
  - Database
    - In order to view any information stored throughout time, it needs to access to the databases to retrieve the information. Also, information is updated to and stored in the database.
  - Reservations
    - All input into the parkingspaces table are reservations. If one needs to learn about a reservation or a customer, it must happen through viewing the Reservation Object
  - Database_Helpers
    - In order to enter in and view data from the Lot Monitor, this class has very many useful functions to traverse the database.
- Variables
  - Int overbook
    - To be used later. This variable denotes how much to overbook the garage for
- Functions
  - Int availableParkingSpot(Calendar start, Calendar end)
    - Looks through the parkingspaces table to determine if there is an opening parking space during two given times. Returns the parkingspaceID is there is such a parking space
  - Boolean availabilityForAParkingSpot (int pSpot, Calendar start, Calendar finish)

- Used in various functions. This looks to see if a single parking space is open during two given times

➢ Booelan reserveParkingSpace,unReserveParkingSpace(int pSpot, Calendar start, Calendar end)

- For a given parking space and time period, the parkingspaces database is either make unavailable or opened up for other reservations to be made or not made at this time

➢ reservationCheckIn (int pSpot, Calendar start, Calendar end)

- A car has arrived and now wants to confirm that they will be staying. This updates the customer and reservation objects/table along with the parkingspaces tables of this event

➢ reservationCheckOut (int spotID)

- charges customer by using the accountant database, clears parking spaces if this car is leaving early and updates the parkingspaces table

➢ float garageOccupancy(Calendar time)
booelan isCurrentlyFull (Calendar time),
ArrayList<Integer>CurrentStatusOfParkingSpaces()

- These three functions access the parkingspaces table to find out properities of the parking parage at the current time or a future time.

Accountant

❖ Dependencies

➢ Database

- Obviously, nothing can happen with the database holding and storing the information.
- Most of the information is stored in the Accountant Database

➢ Reservation

- Accountant needs a reservation to compute the price a customer should be charged by looking at over stay and total stay time.

➢ Customer

- Individual charges are stored to each Customer.

❖ Variables

➢ Float RATE, OVER_RATE, MAX_DAY

- These floats are used to calculate the price of someone that stayed in the garage

➢ Float newLineBalance

- This value is loaded at start up and is the current amount of money that is within the system

❖ Functions

- Float getTimeDifference (Calendar calFirst, calSecond)
  - Computes the difference in hours between the two calendars
- getOverStayTime, get Price (Reservation r)
  - used together to calculate the price of customer owes
- Float charageCustomer (Reservation r)
  - This is how much a customer has been charged for their stay
- Float addToBalance (float newCharge)
  - This simple increases or decreases the newLineBalance

Controller

- Dependencies
  - Database
    - The Controller lets all the devices that have to responsibilities dealing with cars leaving to talk to the database
  - Camera
    - The Controller sends the license plates of the simulated car to the camera to replicate the actual behavior of a car pulling up.
  - Elevator
    - The Controller must tell the Elevator to move once the elevator interface confirms that the car is read to park.
  - EnterInterface
    - The Controller allows communication between the interface, the camera, and the elevator.
- Variables
  - String lp
    - Holds the current license plate of the simulated car currently entering the parking garage.
  - int floor
    - Holds the floor number the current car needs to park in. Passed to the elevator to tell it what floor to go to.
  - Reservation current
    - Holds the current reservation object, to extract needed data like floor number.
- Functions
  - Controller()
    - Constructor of the Controller. Starts the controller thread when called

- Run()
  - The runnable method needed for a thread. Contains the operations of the controller

ControllerOut

- Dependencies
  - Camera
    - ControllerOut sends license plates to the camera to simulate cars leaving
  - ExitInterface
    - Communicates with the camera and gate via the ControllerOut
- Variables
  - String lp
    - Holds the license plate of the current car leaving
- Functions
  - ControllerOut()
    - Constructor for the controller. Starts the ControllerOut thread when called
  - Run()
    - The runnable method needed for a thread. Contains the operations of the controller

Camera

- Dependencies
  - Controller
    - Talks with the controller to communicate cars entering
  - ControllerOut
    - Talks with the controller to communicate cars leaving
- Variables
  - String lp
    - Holds the license plate of the car that just arrived or left
- Functions
  - Camera()
    - Constructor for the camera
  - String getLP()
    - Returns the string
  - setLP()

- sets the string, called when

Sensor

- Dependencies
  - Database
    - Communicates to the database when a car leaves and parks
- Variables
  - Boolean status
    - Indicates whether or not the spot is occupied or not
  - int sid
    - the sensor's numeric id
  - long start
    - the time the car parked in the spot
  - float timeElapsed
    - the amount of time that the car was in the space
- Functions
  - Sensor(int i)
    - Constructor to initiate the sensor, sets the numeric id and the status to empty
  - carArrives()
    - sets the start time to the time the car arrives and the status to occupied
  - carLeaves()
    - sets the time elapsesd to the difference of the current time and the start time and sets the status to empty

EnterInterface

- Dependencies
  - Camera
    - Gets passed the license plate from the camera to the user interface
  - Controller
    - Communicates with the other devices controlled by the Controller
  - Database
    - Gets the required information from the database that is need to display to the customer
- Variables

- Customer curCustomer
- Car curCar
- Reservation curReservation
  - All three of these hold the current objects so different function can access their information

❖ Functions

- makeLicensePlateScreen()
  - make the screen that lets the customer confirm that his license plate was read correctly
- makePersonalScreen()
  - Displays the user's name and current reservation. User can choose to confirm, edit, or say there are a different customer
- makeRegistrationIDScreen()
  - Allows the user to enter their Registration ID if their license plate was not recognized or they are driving a different car
- makeThankYouScreen()
  - create the screen the displays a message parting the customer
- makeCreateScreen()
  - creates the screen that allows the user to create or edit their reservation
- makeWelcomeScreen()
  - Welcomes the user into the elevator
- makeMultiplePersonScreen()
  - Allows the customer to choose who there are if multiple customers associated with the car
- goToCard(String s)
  - Allows programming to easily switch screens
- goToLPScreen(String s)
  - Allows extra information to be passed to the LP screen to display the license plate
- Reservation getCurrentReservation()
  - Returns the current reservation to the controller
- setComboBoxes()
  - Sets the combo boxes on the make reservation screen to the current time to ease reservation creation

ExitInterface

❖ Dependencies

- Camera
- Gate
- ExitInterface
- Accountant

- Variables
  - Reservation curReservation
    - Holds the current reservation so different functions may access it
  - Calendar now
    - Holds the current time from the testbench
- Functions

Door

- Dependencies
  - Elevator
    - Opens and closes when the elevator gets to the correct floor or needs to move
- Variables
  - Status
    - Either open or closed
- Functions
  - closeDoor()
    - changes status to closed
  - openDoor()
    - changes status to open

Elevator

- Dependencies
  - Controller
    - Controller tells the elevator when it can move and where to move, as well as being told where the elevator is
  - EnterInterface
    - Enter interface runs when the elevator is ready
- Variables
  - Int floor

        - The current floor the elevator is at
    - Boolean occupied
        - Status of the elevator
    - Door entryDoor
        - The front door of the elevator
    - Door exitDoor
        - The back door of the elevator
- Functions
    - goToFloor()
        - changes the floor number
    - isReady()
        - checks if the elevator is ready to take another car

MobileInterface

- Dependencies
    - Database
        - Needs to communicate with the database to create and edit reservations
- Variables
    - n/a
- Functions
    - createAndShowGUI()
        - creates the interface for viewing

EnterTestbench

- Dependencies
    - Controller
        - How the testbench communicates with the other devices
    - EnterInterface
        - Simulated car customers interact with the system with the interface
    - Camera
        - Gets and sends simulated license plates to devices
- Variables
    - Int hour

        - Holds the current hour
    - Int minute
        - Holds the current minute
    - Int parkingspace
        - Holds the current parkingspace
- Functions
    - Run()
        - Runs the testbench by sending simulated cars into the garage

ExitTestbench

- Dependencies
    - Camera
        - Sends simulated license plates to the Camera
    - Gate
        - Tells the gate to open and close once the customer pays
    - ExitInterface
        - Communicates with the other devices when the testbench triggers events
    - ControllerOut
        - Helps the testbench communicate with the other devices
- Variables
    - n/a
- Functions
    - Run()
        - Runs the testbench to simulate cars leaving

Design Patterns

Our project made use of a few of the design patterns that were mentioned in the lectures and book of Software Engineering. Specifically we used the proxy pattern, the decorator pattern, and the command pattern.

To start, the design pattern that was used in the backend of the project will be discussed before the patterns used on the front end of the project. The command pattern greatly helped our group coordinate the many functions of the hardware devices. Being that the parking garage was meant to be

implemented in the real world, certain devices would have to be used to make the garage operational. Devices such as cameras, and sensors were used to be able to track cars and process information into the database. Instead of having a class for every camera and every sensor controlling certain functions, we used a hardware controller class. This class coordinated the many functions and capabilities of the hardware devices, and processed the information that they were making available. It was much easier than trying to use a different class for each piece of hardware for each time it was used. Other than organizing information, it also made programming methods and functions for these pieces of hardware easier and more centralized.

Both the backend and the frontend of this project were influenced by different design patterns that streamlined and simplified our project. The interface that people use to register into the system and make reservations is our website. For monetary reasons, this website is run on a local host server, but assuming we acquired a working web domain, this website could easily run on that as well. This website is easy to use for customers and makes the online registration process quite simple and quick. This website followed certain aspects of the proxy design pattern. It is a simple interface that handles accessing the database and inputting, or changing information. All the customer has to do is enter information for account registration or reservation information, and the information is not only recorded into the parking "system," but is loaded into the database. No work is involved for anyone that works there or any system administrators. This obviously makes life easier for the people that are overseeing the system and are watching over the parking garage. It allows the employees to focus on what's happening in real-time at the garage, and not have to worry about inputting reservations. It handles the entire process of accessing the database with ease, and although it did take extra time to code and implement, it was definitely worth the convenience in the end.

Finally, we used the decorator design pattern to provide even more convenience for customers. Along with the website, our parking garage offers yet another way to make or extend reservations. We allow these services by mobile phone. At the moment, we only allow extensions of current reservations, which is incredibly useful if you realize you are going to run over your allotted time. Being able to use the decorator pattern, we were able to find a special case that saves the customer time and energy, and hopefully make them have a higher opinion of our garage. The more happy, the customer is, the more likely he is to become a repeat customer. The goal of the project is to gain as many repeat customers as possible. It is clear that the satisfaction gained by the customers on the front end of our system, and the

workers on the backend of this system gained much satisfaction from our system due in large part to the implementation of the above design patterns.

Object Constraint Language Contracts

- Controller
    - Preconditions:
        - lp = null
    - Postconditions:
        - lp = null
- Camera
    - Preconditions:
        - lp = null
    - Postconditions:
        - lp = null
- Sensor
    - Preconditions:
        - status = true
    - Postconditions:
        - None
- Door
    - Preconditions:
        - status = “closed”
    - Postconditions:
        - None
- Elevator
    - Preconditions:
        - floor = 1
        - occupied = false
    - Postconditions:
        - floor = 1
        - occupied = false
- Gate

    - Preconditions:
        - status = “closed”
    - Postconditions:
        - status = “closed”

- Accountant
    - Preconditions:
        - accountBalance ≥ 0
    - Postconditions:
        - accountBalance ≥ 0

System Architecture and System Design

Architectural Styles

There are many types of architectural styles available for a system design. Event-driven architecture produces an event, or a significant change in states, then consumes, detects, and reacts that event. This is very useful when parts of the system need to wait for other parts of the system to perform a task, and then they can execute upon detection of that task being completed. Another style is front end-back end, where the front-end deals with collecting input, and translating it to a form the back end can use. The front end is also known as the interface between the user and the back end. The back end deals with the underworking of the system – the parts inside the black box, where external programmers aren’t sure how a result is reached; they just know it does correctly. Database-centric architecture is self-explanatory, and relates to software architectures in which databases have an important role in the system. A pipeline is a style where variables in the system travel through many parts of the system while going from the input to the output. Finally, the client-server model distributes tasks over multiple machines out of necessity or efficiency. Each of the styles are seen in a specific part of our system.

Event-Driven Architecture

When looking at the system as a whole, it is easy to see that everything is based on events. Everything from making a reservation, and updating an account, to the entire process of parking and exiting the garage, to the utility functions that provide the framework and information needed to make the system run smoothly are all driven by events. The event emitters are customers and to a smaller extent administration. Everything that occurs in the system occurs because a customer takes some action, be it

making a reservation online, or leaving the garage after having been parked. The nature of the project and the way in which it is meant to work guaranteed that our system would follow this style the closest.

Front End – Back End

Another style that permeates through the structure of this program is Front End – Back End. Clearly the databases are the back end of the program. They store all the information and make the decisions that drive the program. But the rest of the program is put into place to transfer information presented to the system (website, hardware) into language that the databases can understand and use. Then, to interact with the databases in a meaningful way, we have developed several interfaces, the front end, that allow reservations to be made, or accounts to be viewed.

Database-Centric Architecture

The back end of the system will be designed and constructed resembling this model. The databases play an important and influential role in the system. They store everything from client accounts and information to parking information, and reservation details, along with the current state of the garage. All of this information will be stored in files in which information will be inputted into a table like design. Many of the main utility functions of the program are also run directly on the database instead of middle-tier classes. This will work for us because the functions have easy access to the information and updating the data will occur in the database itself, which increases efficiency.

Pipeline

The pipeline style is used in the data area of the program. Many of the variables that are inputted into the system travel between a few classes before they get to the end point. This is especially true with the information that is inputted into the website, and read from the hardware. These variables are ultimately passed to modules that make decisions.

Client-Server Model

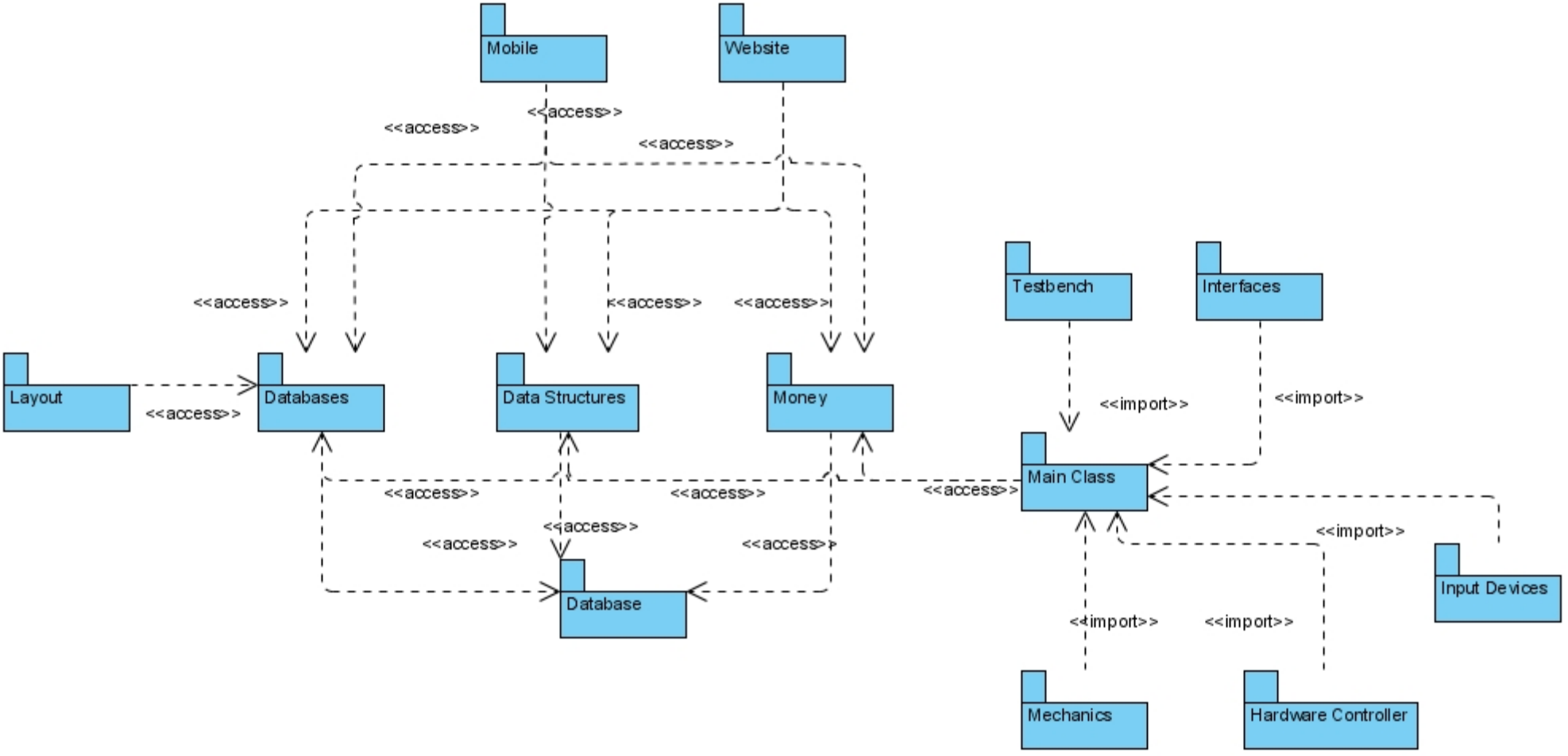
The website that is being used by both the clients and the administration will be of the Client-Server Model. It allows for security and ease of use by both groups of users. It will deviate from the normal characteristics of a Client-Server Model because all of the information and data will not be stored on the website servers. It will be instead stored in databases. However, it will be loaded onto the website servers from the database. An advantage in using this model for the website will be that it will be easy to update information.

Identifying Subsystems

This project had many packages associated with the organization of the devices. There are packages that integral to the system, and others just exist to test the system, since we did not have real input from cameras, etc. The first package is the *database* package. The *database* package deals opening and closing the database, as well as all functions that help make database programming easier. The *databases* package contains test drivers to test the functions of the database, as well as database constructors. The *dataStructures* package contains the three classes for the main types of data structures passed: Car, Customer, and Reservation, as well as the Lot Monitor. The *hardwareController* package contains the two controllers, Controller, and ControllerOut. Each one controls the testbench that simulates cars entering and leaving, respectively, as well as oversee communication between different hardware components. The *inputDevices* package contains the Camera and Sensor classes that represent the actual hardware that would be in the system. The *interfaces* package contains the code to draw the interface for the elevator keypad, and the exit screen. The *mechanics* package contains the classes for the elevator doors, elevator, and exit gate. The *mobile* package contains a replacement for what would be a text messaging interface for someone who quickly wants to make/edit a reservation via their phone. The *money* package contains the Accountant class which handles all the financial transactions of the system. Finally, the *testbench* package contains the two test benches that simulate cars entering and leaving the parking garage. This package would not be necessary for a real parking garage.

The databases are separate since they will be storing different information. The Accounts Database will hold the information related to the cars and people while the Parking Database will hold the information related to the spots. These two databases will be kept synchronized with information related to each other. This synchronization will occur through the Reservations package. This design choice is because anything related to parking spaces will require a reservation and this reservation will be required to access the accounts.

Hardware is its own package. This design choice comes from the fact that we will have to simulate all of the electronic devices anyways so there should be a package that will contain all of the simulate values. For example, camera will have a complex function to “read in license plate” while a sensor will simply have a function “is occupied.”


Mobile
Website
<<access>>
<<access>>
<<access>>
<<access>>
<<access>>
<<access>>
Testbench
Interfaces
Layout
<<access>>
Databases
Data Structures
Money
<<import>>
<<import>>
Main Class
<<access>>
<<access>>
<<access>>
<<access>>
<<access>>
<<import>>
Input Devices
Database
<<import>>
<<import>>
Mechanics
Hardware Controller

Mapping Subsystems to Hardware

Our system **does need** to run on multiple computers. These computers will be used by the employees of the parking garbage, the customers of the parking garage, the system for maintaining a database, and the system for processing requests of its services. Each of the above computers that the system will run on will have different access points to the system, different intentions, and varying levels of access to the operations of the system. Below is a list of the computers that the system will run on and the main modules that will run on these different computers:

1. Database Server
    a. Receive database information requests
    b. Send information from database
    c. Perform computations for expected no shows
2. Web server
    a. Access user information from database and give it to user's web browser
3. Controller Computer
    a. Maintains communications with the sensors on their status
4. Computer for Entering the Garage
    a. Determine the reservation status of the customer
    b. Instruct car to parking space
5. Computer for Exiting the Garage
    a. Charge user for amount of time in the garage
6. Computer in Administrator's Office
    a. View parking garage's status
7. Customer's Computers
    a. Make account and register car
    b. View past history of charges
    c. Make reservations
8. Customer's Cell Phones
    a. Make reservations

Persistent Data Storage

The system **does need** to save data that will outlive a single execution of the system. This information is related to customer's account, parking reservations, and activity within the garage. Ideally, the data will be stored in such a way will allow multiple processes to access the database at the same time, massive amount of data to be stored at once, and have fast lookup time.

The way we are going to store the information is via a SQL database. This is a relationship database, which fits our requirements for storing database. The current SQL database in use is the Microsoft SQL Server 2008. The server is implemented locally on a team member's computer and will be developed to allow external requests of information from other computers. This database allows for fast up scaling to large quantities of data, something our system demands because of the many possible reservations times, customers, cars, and parking spaces our system must support.

The persistent objects that will be able to maintain themselves or their state across several HTTP requests are each of the columns in the SQL database that are listed below.

Below are the two databases that will be implemented with the SQL database. One is for customers and cars to track their reservations; the second is meant to work with the reservations of spaces.

The first maintains reservation and account information per customer.

For the second, we expect that the parking garage will be at maximum capacity most of the time, so having a full table will require a large amount of storage but will allow for fast lookup and data calculations. The database will expand when necessary to add additional columns for dates and times. These will increment every 15 minutes.

Customer Database Columns:

- Customer ID
- Customer Name
- Current Balance
- Car ID 1
- Car ID 2
- Car ID 3
- Car ID 4

- Car ID 5
- Pref_Floor
- Pref_Handicap
- Pref_Space
- CurrentReservation
- Upcoming Reservation ID 1
- Upcoming Reservation ID 2
- Upcoming Reservation ID 3
- Past Reservation ID 1
- Past Reservation ID 2
- Past Reservation ID 3

Car Database Columns:

- Car ID
- licensePlate
- Customer ID 1
- Customer ID 2
- Customer ID 3
- Customer ID 4
- Customer ID 5

Reservation Database Columns:

- Reservation ID
- Reservation_Type
- Customer ID 1
- Car ID
- Parking Space ID
- Reserved Arrival Time
- Reserved Departure Time
- Actual Arrival Time
- Actual Departure Time

Accountant Database Columns:

- Transcation ID
- Charge
- Hours
- Parking Spot
- Customer ID
- Customer Name
- Car ID
- License Plate
- New line balance

Parking Space Database Columns

- Parking Space ID
- Char_Handicap
- Current Reservation ID
- Reservation ID at January 1, 2011 at 12:00 am
- Reservation ID at January 1, 2011 at 12:15 am
- Reservation ID at January 1, 2011 at 12:30 am
- Reservation ID at January 1, 2011 at 12:45 am
- Reservation ID at January 1, 2011 at 1:00 am
- Reservation ID at January 1, 2011 at 1:15 am
- Reservation ID at January 1, 2011 at 1:30 am
- Reservation ID at January 1, 2011 at 1:45 am
- Etc…

Network Protocol

In order to facilitate communication between the parking garage reservation website and the parking garage database, the Java JDBC communication protocol will be utilized. This protocol involves a Java API which provides methods for easy communication between the Java application and the database server, such that the website can seamlessly access and update values within the database. The JDBC network protocol is especially suited for relational databases, which will be especially useful when handling a database filled with relationships between customers, vehicles, and reservations. Each customer can be affiliated with multiple vehicles and each vehicle can be affiliated with multiple

customers. Furthermore, each customer or vehicles can be tagged with multiple reservations. In this situation, where there are a multitude of potential overlaps, relational databases prove to be extremely efficient in reducing the number of tables and objects involved. When a customer is using the website to access his account, the website will be able to access the database by using the Java JDBC in real time to gather information that the customer is requesting. Such information will be supplied in an efficient manner because the database will be constructed and optimized to provide output quickly for the expected queries. Thus, the Java JDBC network protocol proves to be the best choice within a system where a remote Java application must communicate with a database.

We used Apache Tomcat, which is an open source software implementation of the Java Servlet and JavaServer Pages technologies, to allow customers to logon on to a website on the internet. By allowing this behavior customers can create, edit, and manage their accounts. They can make new reservations and associate cars with themselves.

Global Control Flow

Execution Orderness:

Our system is event-driven since it is almost completely based on user input. An example of a part of our system that is ¬process-driven is the license plate reader, which is on a loop, to read a license plate, and then display a user interface to the customer. However, after this the events of the system are determined by customer behavior, what options they select, what reservations they hold, etc.

Time Dependency

Our system charges customers for how long they stay, so there must be some implementation of a timer in the system to determine how long they stayed. It's periodic in the sense that it repeats for each customer who enters, parks, and leaves the parking garage.

Concurrency

For the web database, if two users want to access the same information at the same time, we would need multiple threads for that. (i.e. a husband and wife, both logon on to edit a license plate, on two different accounts.) For this we just need a simple lock around sections where data is being edited and then read, so wrong values are not used. Otherwise, only one customer can park in any spot at one time, so each sensor is single-threaded. Only one customer can be in the elevator and use the keypad, so that can be single-threaded. Also, only one customer can leave at a time, so that camera is single-threaded. Finally, the camera that reads license plates is also single-threaded since it only reads one at a

time. The controller itself must multi-threaded since it needs to send requests from multiple sensors and cameras to the main system. For this, the requests will need to be put a queue, since the controller can only deal with one at a time. We need to make sure there is not starvation for one of the input devices.

Hardware Requirements

- Touch screen display
  - minimum 800x600 pixels. The touch screen display will be the main way of the customer communicating with the system and entering information into the database.
- Hard drive
  - At least 10 GB of space. This will be the local hard drive for the garage where the local, cached information is stored. It'll hold the information about the parking spots and the customers.
- Network Connection
  - A 1Mbps connection to support website and database transfers. A fast connection will allow the online customer database to interact with the system at the garage to keep track of reservations and which customers showed up.
- RAM
  - Minimum 2 GB of RAM to process the information from the touch screen. Without a good amount of RAM, the touch screen system UI will be slow and customers will be stuck in the UI menus waiting for them to load
- CPU
  - 2GHz processor to process the times and customer information efficiently. A slow processor will again make menu navigation a headache. Fast speed and efficiency will keep the customers happy and make things move faster.
- Sensors
  - IR sensors to detect whether or not a car is in a parking spot.
- Camera
  - 5 Megapixel Resolution for optimal reading of license plates

Algorithms and Data Structures

Algorithms

Future editions of this parking garage system will employ an overbooking algorithm to maximum the usage of the parking spaces to the greatest degree possible. This algorithm predicts the number of no-shows, overstays, understays, and walk-in based on historical data and accepts the appropriate amount of reservations to make sure the parking garage is always running at 100% occupancy or as close to that as possible. The overbooking algorithm takes the total number of parking spots within the parking garage and subtracts the number of parking spots that are predicted to be unavailable at a certain time. The difference provides the number of parking spots that are predicted to be open for reservations. The exact algorithm is as follows:

$$P_F = P_T + U_P - O_P - W_P - (R_C \cdot N) \cdot (R_G \cdot N)$$

Where:

P_T = Total # of parking spaces within garage

R_C = Confirmed reservations

R_G = Guaranteed reservations

N = No-show factor for reservations

O_P = predicted overstays

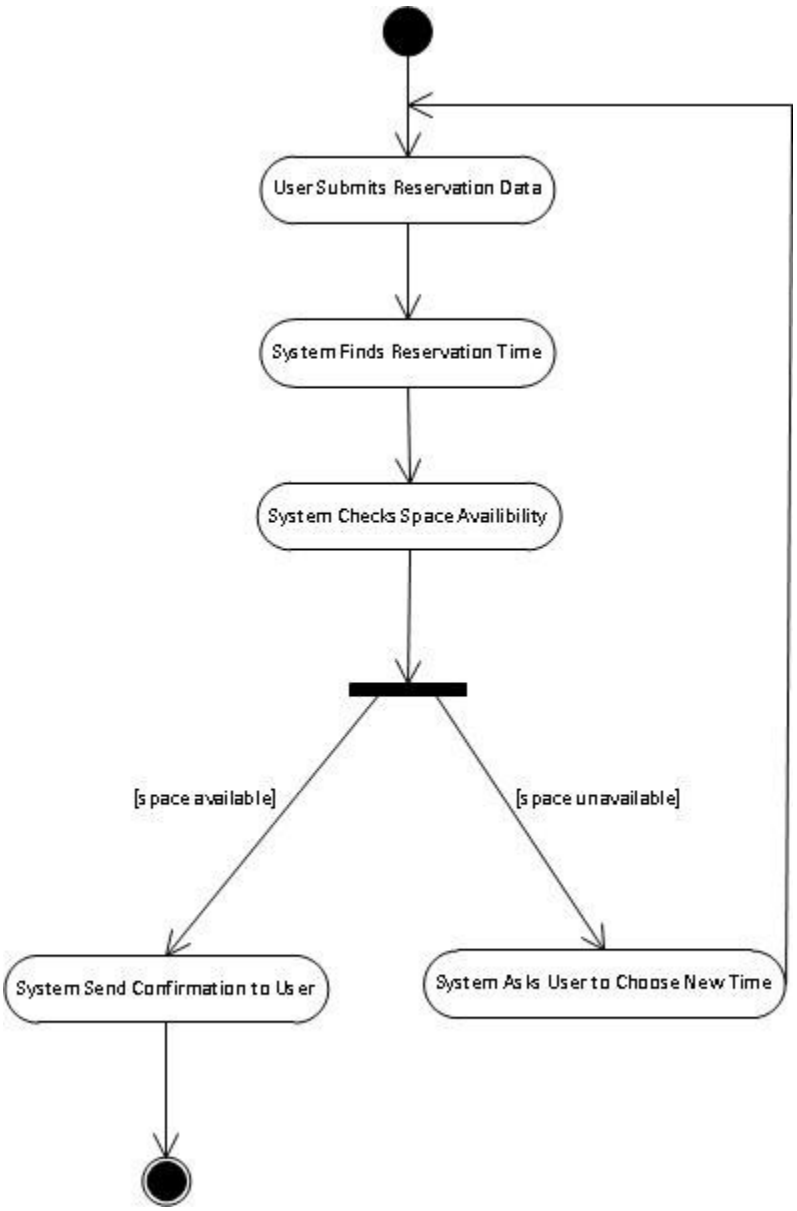
U_P = predicted understays

W_P = predicted walk-ins

P_F = # of parking spaces that are currently free

The confirmed reservations account for one time reservations made by registered customers and the guaranteed reservations account for contract reservations made by registered customers. The no-show factor is based on historical data gathered during parking garage operation and it represented the percentage of customers that actually show up to claim their reserved parking space. The overstay count is predicted based on the percentage of cars within the garage that overstayed their parking duration in the past and the understay count is predicted based on the percentage of cars that understayed their

parking duration in the past. The walk-in count is predicted based on the percentage of cars within the garage which are simply walk-in customers who have not made a reservation. Thus, this algorithm relies heavily on the ability to collect, store, and analyze the parking garage data.



A different algorithm that will be implemented is an algorithm to solve the issue of maximizing the amount of cars within the database. Due to the nature of how cars sign up for parking spaces, the garage can be under-utilized during parts of the day, resulting in money lost. For many reservations and many parking spaces, trying to optimize the database is no simple solution. This problem is similar to a famous computer science NP hard problem: “bin packing”. Since it is in the set of NP hard problems,

that means there are no known efficient solution to solve this problem. Therefore, approximations must be used to determine the best parking garage fit. Then again, this all depends on customer arriving and leaving on time. If the future, an algorithm will be implemented that will try to move reservations arrange if a customer stays late. This will involve moving the next reservation to another parking space and if there are none, moving another reservation. This continues for about 10 repetitions before it is abandoned. For optimizing the database, an algorithm similar to the linear time solution algorithms of the bin packing problem will be used. It will try to move reservations that begin and end at close times next to each other in order to free up space into larger blocks. This is similar to defragmentating the database. This type of optimizing would occur on future days of reservations. It is dangerous to run this optimizing on the current day.

Data Structures

Using the best objects is critical for this parking garage project to reduce the demand on the database. The theory is that by creating a object from the information stored in the database and then allowing the other programmers to treat everything as an object, their lives are easier and they never need to know that a database exists. That is how this group programmed the database back end. For test drivers and the website, calls to Reservation, Car, and Customer were handled as if they were objects. This made everyone's lives much easier.

They important part of this is being able to convert the links between these Objects to other objects. These are denoted by ID numbers and each Car, Customer, and Reservation has a unique ID number that corresponds to a table in the database. For example, given a car, one can determine all the customers and then get all the reservations associated with that car. Another example is that each customer has references to reservations but that is all the information that it stores. Having all the reservations stored in one place in the database makes everything extremely simple. This is something this group worked hard on for demo #2 after demo #1.

Customer Object contains the following items

- Int Customer ID
- String Customer Name
- Float Current Balance
- Int Car ID 1
- Int Car ID 2

- Int Car ID 3
- Int Car ID 4
- Int Car ID 5
- Int Pref_Floor
- Int Pref_Handicap
- Int Pref_Space
- Int CurrentReservation
- ArrayList<Integer> Upcoming Reservations
- ArrayList<Integer> Past Reservations

Car Object contains the following items

- Int Car ID
- String licensePlate
- ArrayList<Integer> Customer IDs

Reservation Object contains the following items

- Int Reservation ID
- Int Reservation_Type
- Int Customer ID 1
- Int Car ID
- Int Parking Space ID
- Calendar Reserved Arrival Time
- Calendar Reserved Departure Time
- Calendar Actual Arrival Time
- Calendar Actual Departure Time

Storing Mass Quantities of Data in Memory

We rarely store much data in memory because when objects are read and edited, they are now no longer needed. Garage collection will clean them up.

If we see the need to store mass amounts of data in memory for search, fetch, and edit, we will use red-black trees. The reason for this is threefold, they sort the data by the field we decide. They are always balanced, and self-balance so little maintenance is required on the user end to keep them organized.

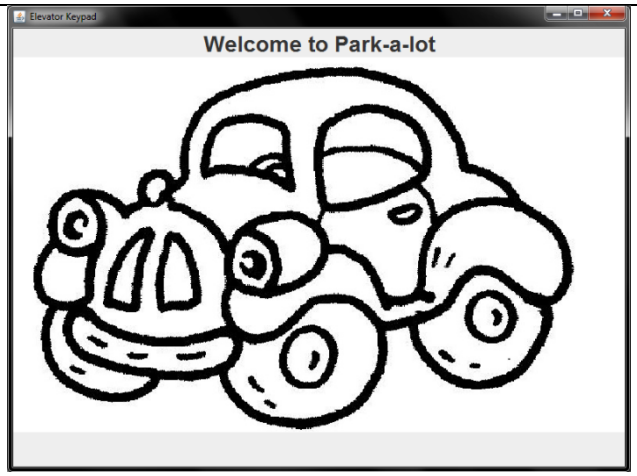
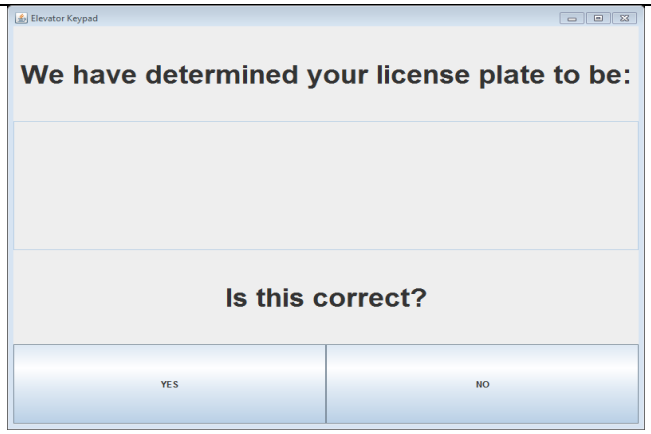
They also search very quickly, on the order of log2n. This will depend on the criteria of the master system, for our demo, we will have to scale this aspect down to get a reasonable initialization time.

Other Data

All other permanent storage of data will be done with a database, which will be read when required. This is the storage device that stays alive if power is lost and is critical for smooth operation of our parking garage.

User Interface Design and Implementation

The following description is the final interface of the UI after the completion of the project's coding

Upon the arrival of a customer to the entrance of the garage, a splash screen is displayed welcoming them to the garage as a placeholder while the camera of the garage scans their license plate to get their identity. Figure 1 is the current splash screen displayed while the camera does its job.	 Figure 1
Once the license plate is read, an image of the license plate is displayed in the screen shown in Figure 2. The image of the license plate is scanned using a picture-to-text converter and displays the license plate accordingly. Assuming the license plate is correct, the customer can confirm this by clicking Yes to answer the question "Is this correct?" If the customer hits no it will attempt to rescan the plate or allow the	 Figure 2

customer to enter their plate number.

If the customer determines that the license plate is indeed theirs, the screen displayed in Figure 3 shows up and shows them their current reservation assuming they have one. Upon entering this screen the customer has the option to confirm their reservation or possibly edit the reservation. Upon choosing the edit option, they are prompted and asked what they want to change their reservation to, or if they are content with their current reservation they can confirm it and go about their day as previously planned.

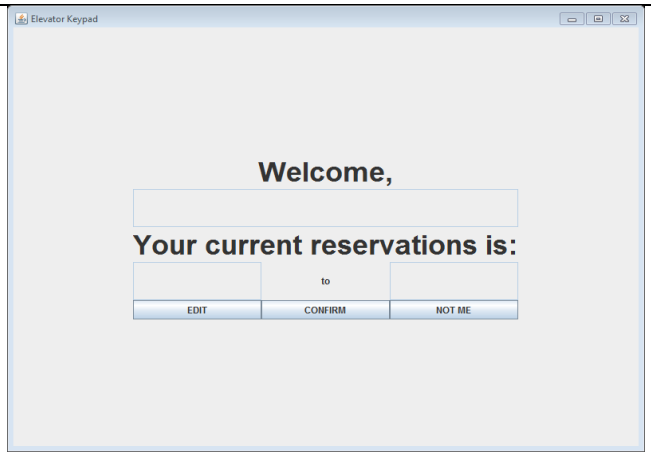


Figure 3

If the license plate is not correctly recognized or the customer clicks “this isn’t me” at the previous screen, the customer is brought to the screen in Figure 4 where they are asked to manually enter their Registration ID. If they type in a correct number, they will be brought back to the screen that shows them their current reservation and will be allowed to edit or confirm. If in they choose the skip option assuming they don’t have a registration ID they are assumed to be a walk-in customer and get

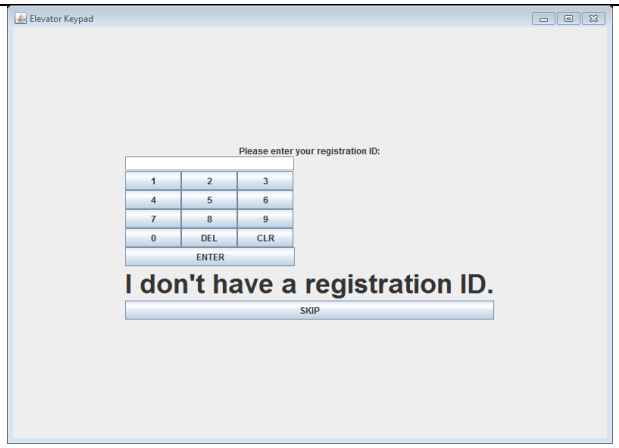
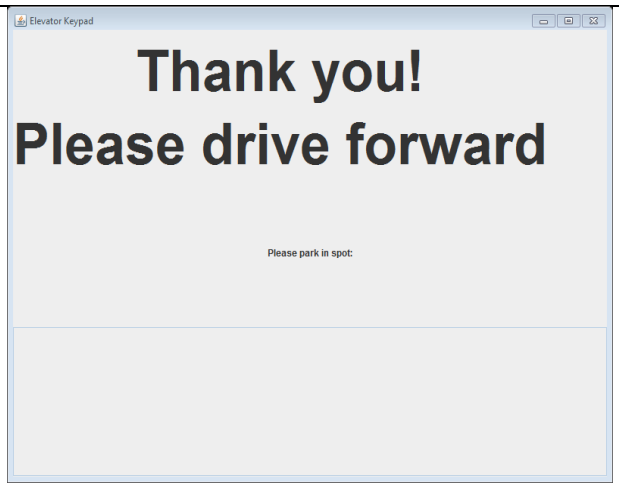
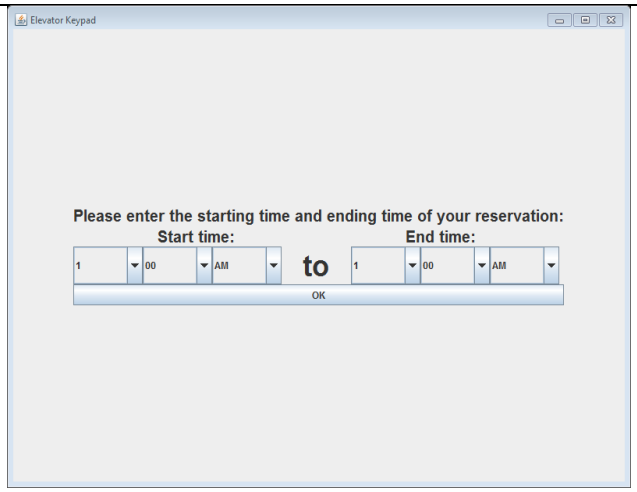
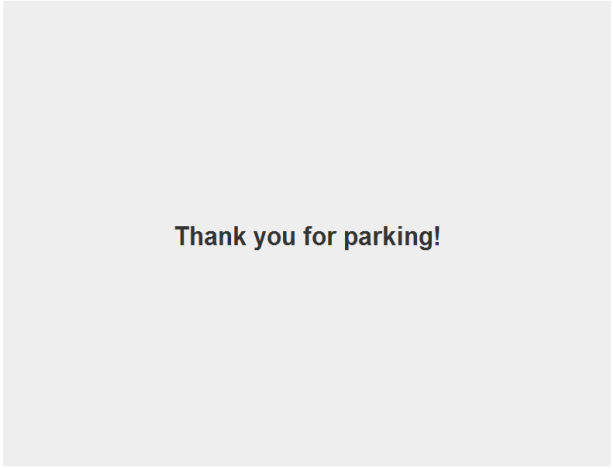
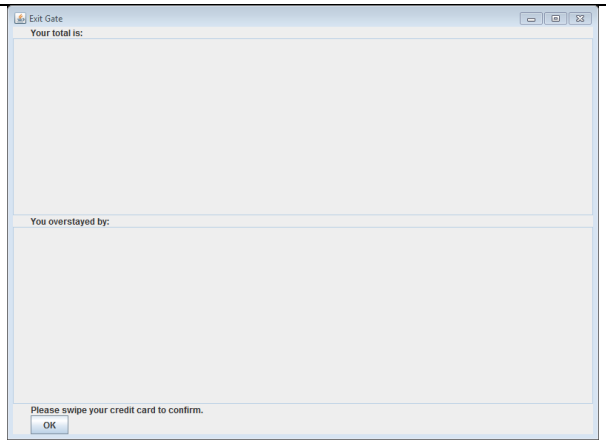


Figure 4

to choose their reservation time.	
Upon confirming their reservation or entering their walk in times for arrival and departure the customer is prompted by the screen to drive forward to denote a success of them making it through the system. The screen shown in Figure 5 will show them which spot to park in as they drive into the elevator. Once the elevator knows which spot you are going to it can tell which floor you are going to and ready itself to take the customer and their car to the appropriate floor.	 Figure 5
If the customer is registered and wished to edit their reservation or if they are a walk-in with no reservation they are brought to the screen shown in Figure 6. They can choose their start and end times via the dropdown menus shown. Once they hit ok they are brought back to the screen in Figure 3 and asked to confirm the reservation once more. This gives the customer time to check if they are signed up for what they want to be signed up for and allows them to edit it once more if they for some reason made a mistake inputting the numbers.	 Figure 6

This is the splash screen customers will see upon exiting the garage. Figure 7 will be used as a placeholder while the camera figures out what license plate their car is using. Upon entering this screen, it means the customer is on their way out, and is done with their parking experience.	 Figure 7
The screen displayed in figure 8 shows the customer how much they will be charged based on whether or not they left on time, late, or very early. If they are getting charged extra, the charge will be explained by showing the customer how many minutes they overstayed. The customer can confirm the information displayed by the screen in Figure 8 by swiping their credit card and paying the required amount. After paying they are shown the screen in Figure 7, thanking them for parking at Park-A-Lot.	 Figure 8
The screen depicted in Figure 9 is that pretending to be the input screen for a cell phone. The way it works is that the customer may type in either extend or make to extend their next reservation or make a new reservation. Assuming they choose to make a new one, they will need to add two arguments, a start time, and an end time. If they choose to extend, they will need to enter the number of	 Figure 9

minutes they wish to extend their reservation. This is very helpful for changing and making parking reservations on the go. The output is what the garage will send back, confirming their inputs or giving them errors as a response.	

History of Work and Current Status of Implementation

There were a lot of key accomplishments and milestones that occurred throughout the duration of the creation of the project. Even some of the smallest feats were actually huge accomplishments toward the success of the overall project. Getting two functions to work together was a milestone each and every time it worked without failure. Seeing a code compile without any runtime or exceptions for the first time was also a big deal each time it happened. Each little thing that worked correctly contributed to the overall success of the project and with one miniscule error, the whole project could have potentially not worked or just flat out crashed altogether.

The current status of the project is far from a fully finished project, but still an impressive program nonetheless. In its current state there are many functions that at a glance may look simple but actually took a lot of work to have working correctly. The first thing that needed to be in place was the SQL database. Without the database the Java code and the Tomcat implementation would have been useless. Creating different levels of hierarchy in the SQL database was a huge milestone for the project. With a database of customer information as well as reservations, it became possible to manipulate the database accordingly. From there coding all of the functions in java to manipulate the database became the center of focus. Using the combined code of the group to manipulate the database the number of necessary functions seemed to become endless.

A function would be necessary to add a user to the database, give them a car associated to their account, get their phone number as well as assign them a unique membership ID. On top of that once the user was created, said user needed to be able to create and manipulate their reservations in the reservation database. This called for specific classes to be made to allow manipulation of the reservations database linking the reservation to the customer and assigning times, parking spaces, and reservation IDs to each customer. Once a car had been thrown into the reservation database with a start and end time, this required the database to check to see if the car had arrived for its scheduled time

early, late, or on time. Being early or late by more than 15 minutes would result in different penalties whether it is more money, or having to switch spots since one was not open due to the timing being slightly off. For example if a car was scheduled to leave, and another car was scheduled to arrive at the same time as that car was leaving, but the second car showed up early, it would technically have nowhere to park since the assumption was that it was going to take the leaving car's place.

Exception handling was a big part of the project as you can tell. People leaving late or arriving early threw monkey wrenches into the programming and caused more headaches for the coding and the operation of the garage in general. Each exception was coded into the program to make sure every car was accounted for and that people who overstayed their welcome were punished. This caused the creation of the accounting classes which dealt with the money of the garage. When a car left it would calculate the time it came in, the time it was supposed to leave, and the time it actually left. If an overstay was detected it would add a significant amount of money to the original payment amount and charge the credit card of the customer automatically.

This was another milestone for the programming of the system. The ability to automatically take and record payments became a very necessary step for the garage. Not having to manually go over payments with every customer and allowing the garage to do it automatically is very handy. Since the system can figure out when a car was supposed to leave vs. when a car actually left, it is easy to tell when a car overstayed its welcome and therefore the driver would need to be charged accordingly. Giving the system a set of dynamic rates for hourly stay and overstay penalties allows all calculations to be done automatically and since each customer can register with a credit card number, verifying the validity of the card is as easy as sending a signal to an ATM.

As for the overall completeness of the project, the project achieved what it was set to do. As the deadline grew nearer and nearer, the amount of possibilities to expand upon the project increased exponentially and more ideas kept coming as a result of functions working properly. As a result of things being completed, the amount of new ideas got better, and it seemed as if the project was less and less finished. The goals established by report 2 however were all met and the overall demo was a success.

Conclusions and Future Work

Obviously as with any project, there is never a time where one is completely satisfied with their end work. There is always some feature that could be expanded upon, some menu that could look just a

little bit better, or some function that could have been coded just a bit more optimally. There is also a good amount of mental and physical limitations to everyone which puts the amount of work they can accomplish at a plateau. These can be as simple as time restraints or as complex as not knowing the correct syntax to make a section of code compile. Technical challenges can sometimes be easy to solve, but a lot of the time require a lot of prior knowledge to understand or hours of studying to fully comprehend the solutions to them.

The very first technical challenge that arose for many of the group members was the inability to get the Java Eclipse Ganymede to work synchronously with the MySQL database as well as the Apache Tomcat software. We learned after the demo had been completed that there was actually a program that would install all 3 on a machine automatically, removing the entire headache involved with linking them, and doing what seemed to be the hardest part in a matter of minutes. As with any group project, one of the hardest parts in the end wound up being making everyone's code work together as one, instead of as separate entities. If a function didn't return the correct type of variable, or the classes were not imported correctly, the function would ultimately not work. Despite the amount of time allotted for the project, getting everyone's busy schedules to line up in terms of free time was also hard to do. Meetings for clubs and social events as well as homework for other classes made it very difficult to get everyone together in one place at the same time.

One of the nice things about coding mainly in Java Eclipse Ganymede is the way it handles coding errors and syntax errors. If there is an error during compilation, four out of five times eclipse is able to understand the error and fix it to the best of its abilities. Obviously this does not always work but it did make a lot of the error-checking painless and easy on everyone. While the eclipse platform did have a good way of error checking it would sometimes lead to problems when the preferred solution was not the correct way and the solution would sometimes cause other parts of the program to function incorrectly. This proved to be a pretty big challenge since it was not always a good idea to trust the built in error handling of eclipse since it sometimes became more hurtful than helpful. A lot of the time taken to code was debugging and having five members with conflicting ideas on top of a program that thought it knew the answer was not always an easy setup to work with.

The Software Engineering class really did help us with the overall completion of the demo and project coding. Using diagrams and use cases to map out the project and gain a better understanding of the functions we were dealing with made it much simpler to code and get everything to work. With mapped out functions and arrows telling everyone what each function needed to return and to where, it made

the coding portion of the project much smoother for everyone involved. If a member wanted to know what type of variable their function should output or another member needed that function's output to use as an input for their function, it was much easier to look at the diagram than to either read code or have to physically ask the other member what their function was doing. Without the class I do not think it would have been half as easy to code the functions or to keep everything organized. If there was no knowledge of use case diagrams or object-oriented design, making the layout for the code and project would have been very close to impossible.

To be honest, the class did do a good job of teaching us as a group how much effort actually goes into something as straightforward as making software to automate a garage. At the beginning of the course a large amount of the class probably said to themselves "We have all semester? Easy." But after the procrastination kicked in, and the amount of time they actually had dedicated to other classes, clubs, organizations, and fraternities hit them, they realized how hard working on a deadline in and how much work actually goes into creating a successful product. Debugging, working as a group, and making code function when multiple people are working on it is not as easy as it at one point seemed. This class proved to be a great learning experience not just in terms of schoolwork but also in life. Cooperation and time management can't be taught in a classroom environment and this was many of our first time working on something so big and important.

There were also a number of things that could have made the project a bit easier as well. For starters, most of us had focused for the most part through college on MATLAB and C++. None of the classes we have taken thus far really focused on Java much except for the second half of the Programming Methodologies II course which only had a small java coding project at the end of it. A class dedicated to Java programming should really be incorporated into the ECE curriculum since even though MATLAB is very useful for the mathematics courses, it doesn't really help us out when working with complex UIs and in depth functions that don not always work around number crunching. A majority of the group also had no prior knowledge of Tomcat or MySQL. Database management was the bulk of the project's focus and going into it without much if any prior knowledge about the two platforms made starting the project a difficulty for many of us. If there was a database coding class or a class on languages other than just C++ and MATLAB, I feel as if the project would have gone smoother for not only this group, but a lot of the other groups. Members with high school knowledge of Java and SQL seemed to excel much more than those who did not have such prior training and essentially had an easier time doing things quickly and more efficiently.

As the coding came to a close, it became apparent there were a lot of different areas in which we could have improved the code, the functionality and the overall appearance/presentation of the project. The most obvious areas where we could have improved on the design were the website, the mobile interface, and the admin functionality. One of the key problems with the project is that there was no accurate real-time clock that was being updated. Using time as a metric makes many new functions possible since instead of always operating in the current time, it would be much easier to check things in the close future or not-so-distant past. If for example it was possible to keep track of the current time, logs could have been taken in real time allowing for the database to be able to go back in time to look at parking spaces, reservations, and also install alerts for things that were going to happen in the near future.

With the mobile interaction idea, which involved customers being able to text in either make or extend followed by times in the format <make/extend> <timein/extension time> <timeout/NULL> , the possibilities seemed endless. Using the cell phone text message as somewhat of an input terminal for the database, it could have been cool to add the function to fully manage the account via text messaging. For example a menu could have been returned to the customer in the form of a text message and depending on the message received by the database in response to the menu; the customer could have full control over their account and not just their upcoming and current reservations. With the idea of a clock, it could have also been cool to have the ability of the database to send alerts to the management or the customer via text message saying “hey you have a reservation in fifteen minutes” or “your reservation is set to expire in 15 minutes, would you like to extend Y/N?” Small things like this are what customers like to see out of companies since a function like that would drastically cut down the amount of overstays and fines for customers while also keeping the garage running smoothly at all times.

The website also could have been improved upon. In its current state, the functionality of it is very limited. In the future it would have been nice to add the ability for the customer to see their account balance, their recent charges to their account, as well as view their upcoming reservations. Online support is becoming a very integral part of many if not all companies in this day and age and people like to be able to sit down at their computers and view any and everything going on in their daily life. Email notifications for upcoming reservations or email based alerts for possible mess-ups on their part would again make the garage run smoother and keep the customers happy. Customer satisfaction is always a key part in the success of any business and a parking garage should not be any exception. Given the

amount of understanding of Java at the end of the project, it would have also been cool to incorporate some sort of app for the iPhone or Android platforms.

Even text messaging is starting to phase out of everyday activities as smartphones are becoming the new standard. Since the coding platform for Android is very similar to that of Java Eclipse, coding the functionality of the database into a remote platform for the droid would have probably been somewhat trivial at the conclusion of the project. If the management could view what was going on with the garage in real time from their phone, it would make it that much easier to run the garage without having to do any physical work. Alerts could be sent to the workers in the form of texts or droid notifications/alerts and redirecting phone calls directed at the garage straight to workers without them having to be present at the garage would open new doors to work-from-home options at the garage itself. The Android platform is a growing one and has proven to become a very profitable platform to code for with many people becoming overnight millionaires just for making an app a large population of people find useful. Since commuters are rampant in major cities all over the world, getting this software running at many different garages all over the country could return a lot of profit for us as the designers as well as the garages attracting new customers with better, newer technologies.

Bibliography

Bruegge, Bernd and Allen H Dutoit. Global Control Flow: Object-Oriented Software Engineering: Using UML, Patterns, and Java. Prentice Hall, 2010.

Marsic, Ivan. Software Engineering. New Brunswick, 2009.

Microsoft. Chapter 3: Architectural Patterns and Styles. n.d. 10 March 2011 <http://msdn.microsoft.com/en-us/library/ee658117.aspx>.

Miles, Russ and Kim Hamilton. Learning UML 2.0. Ed. Eric McLaughlin and Mary O'Brien. Sebastopol: O'Reilly, 2006.

Oracle. Hierarchy For All Packages. n.d. 2 March 2011 <http://download.oracle.com/javase/1.5.0/docs/api/overview-tree.html>.

Sun Microsystems, Inc. Java Look and Feel Design Guidelines. Mountain View, 1999.

Visual Paradigm. VP Galley. n.d. 7 March 2011 <http://www.visual-paradigm.com/VPGallery/index.html>.