System Specification and Design: Parking Garage Automation

Matt Edwards, Eric Wasserman,
Abdul Hassan, Juan Antialon

Group Number 2

Project URL: http://code.google.com/p/parking-garage-automation/
http://www.park-a-lot.vacau.com/

Effort Estimation

The table below is a breakdown of the individual member's contributions to each part of this report.

	Abdul	Matt	Eric	Juan
Summary of Changes (*5 points*)		100%		
Customer Statement of Requirements (*6 points*)		50%		50%
Glossary of Terms (*4 points*)		80%		20%
Functional Requirements Specification (*37 points*)	20%	30%	50%	
Nonfunctional Requirements (*6 points*)		80%	20%	
Domain Analysis (*25 points*)		50%	50%	
Interaction Diagrams (*30+10 points*)			100%	
Class Diagram and Interface Specification (*10+10 points*)	50%	50%		
System Architecture and Design (*22 points*)	40%			60%
Algorithms and Data Structures (*4 points*)		100%		
User Interface (*8 points*)	100%			
History of Work and Current Implementation Status (*5 points*)		50%		50%
Conclusions and Future Work (*5 points*)		60%		40%

References (*3 points*)		100%		
Project Management (*10 points*)		50%	50%	
Software Coding (*? points*)	100%			
Total Points Contribution	34.2	67.1	77.2	21.5

Table of Contents

Summary of Changes

Since this report is a collation of our first and second reports with revisions included, here we specify the major changes we have made between the first, second, and third reports.

First Report - Third Report

Section	List of Changes
Customer Statement of Requirements	● Revised Problem and Solution to make it clearer; ● Updated Business Policies to more accurately reflect goals of Park-A-Lot; ● Simplified Business Requirements table, split up complex requirements into simpler ones, and added a few new ones towards the end; ● Added Prices and Fees section to explicitly show penalty fees.
Use Case Descriptions	● UC-2 Park was expanded to include the case of a customer arriving early; ● Modified UC-4 to View Reservations; ● UC-7, 8 and 9 were all added; ● UC-13 MonthlyBilling was added to handle monthly billing of registered customers.
Use Cases (Fully Dressed)	● Same modifications as above, most notably expanding UC-2 Park.
User Interface Design	● Included screenshots and navigation paths for newly designed and updated user interface.

Second Report - Third Report

Section	List of Changes
Interaction Diagrams	● Added new section, Theoretical Diagrams, to better explain what happened with UC-2 Park, why we did not implement it, and what could be done with it in the future; ● For the remaining interaction diagrams, included info about how Model-View-Controller architecture ensures all design principles to a high level; ● Updated UC-2 Park and UC-10 Authenticate with design patterns; ● Added new use cases from modified Report 1.
Class Diagram	● Updated class diagram to include new classes added

	since last iteration; ● Data Types and Op Sigs updated to include changes to code since last iteration; ● Included short section on Design Patterns and how they influence our class diagram; ● Added OCL section.
Data Structures and Algorithms	● Included parking lot bitmap data structure; ● Included algorithm for scheduling reservations using swapping; ● Included algorithm for determining best alternative reservation; ● Included algorithm for determining discount pricing.
User Interface Implementation	● Combined this with UI design from Report 1 into one large section.
Code Appendix	● Added an appendix for C++ code of reservation swapping algorithm.

Customer Statement of Requirements

Statement of Goals

The main goals of this software development project is to maximize the occupancy and revenue of a parking garage and develop a user-friendly mechanism that helps customers find and reserve available parking in the garage, either in advance or at the time of parking.

Problem Statement

The problems that are faced is as follows:

Our customer owns a parking garage that lacks a computerized system for handling the logistics of parking vehicles efficiently, thus he/she is currently losing profit because of the lost opportunity of not maximizing the available parking spaces. At times of peak parking in the area, the garage may not be filled to maximum capacity since there are only primitive ways to indicate to customers that spots are available inside.

Second, there may be a congestion problem inside the garage since customers might search for an available spot when no spots are actually open. This congestion would decrease repeat business, because no customer likes working with an inefficient system.

Finally, some customers may be discouraged by the possibility that driving to an area would inconvenience them if no parking were to be available. Since the garage has no way to signal to customers remotely to their homes that parking is available in a certain garage at a certain time, those customers that choose to stay at home rather than “risk-it” are lost sources of revenue.

Proposed Solution

In order to increase profits and reduce personnel costs a computerized system will be put into place to address all three problems suggested above.

The first problem of signaling to customers who may drive by the garage and not know if any parking is available can be solved by implementing a display at the entrance of the garage indicating if room is available to house customers driving in off the street. The second problem, congestion inside of the garage, can be solved by assigning customers parking space numbers of spots in the garage, so that the customer will always know where the next available spot is. Finally, the third problem of not knowing if parking will be available for a given date and time will be solved by implementing a web-based system (a website) that will allow customers to create accounts and create reservations using that account. These reservations will be simple guarantees that parking will be available for the selected date and time, removing the fear of being stuck in an area with no place to park.

To implement these changes, several new pieces of hardware will need to be adapted for use in the garage. The system is based upon a multi-level garage that has a vehicle elevator between levels. Cars may only enter the garage through the elevator, and can leave through a one-way exit ramp. This elevator can only accommodate passenger vehicles, so large trucks and other high-capacity vehicles will need to be excluded from parking here.

The remaining hardware will handle detecting if a vehicle has entered the garage, detecting a vehicle leaving the garage, and determining if a vehicle is occupying a spot in the garage.

The final piece of our solution is a delineation between customer types: registered customers versus walk-ins. Registered customers have created accounts with the parking garage's online service, and may make reservations in advance. Walk-ins are not associated with the online service and can only park when they drive in off the street. Since this business will derive a majority of its profits from repeat business of registered customers, we elect to have the ground level of the parking garage allow only walk-ins to park, whereas the upper levels will be reserved for registered customers fulfilling reservations. It is possible that at some point in the future these restrictions could be eased, however the initial software solution will be kept simple.

The parking and reservation system will be nicknamed "Park-A-Lot".

Devices

To accomplish the goals state above, Park-A-Lot will implement the following devices:

S1 & S2. The cameras will be installed to act as license-plate readers. S1 is the camera at the elevator and S2 is the exit camera. The cameras will be using the license-plate recognition system that are often used in tolls. The basic idea is that when a car arrives at the elevator platform, S1 reads the registration number and later on S2 will read the registration number of the vehicle leaving.

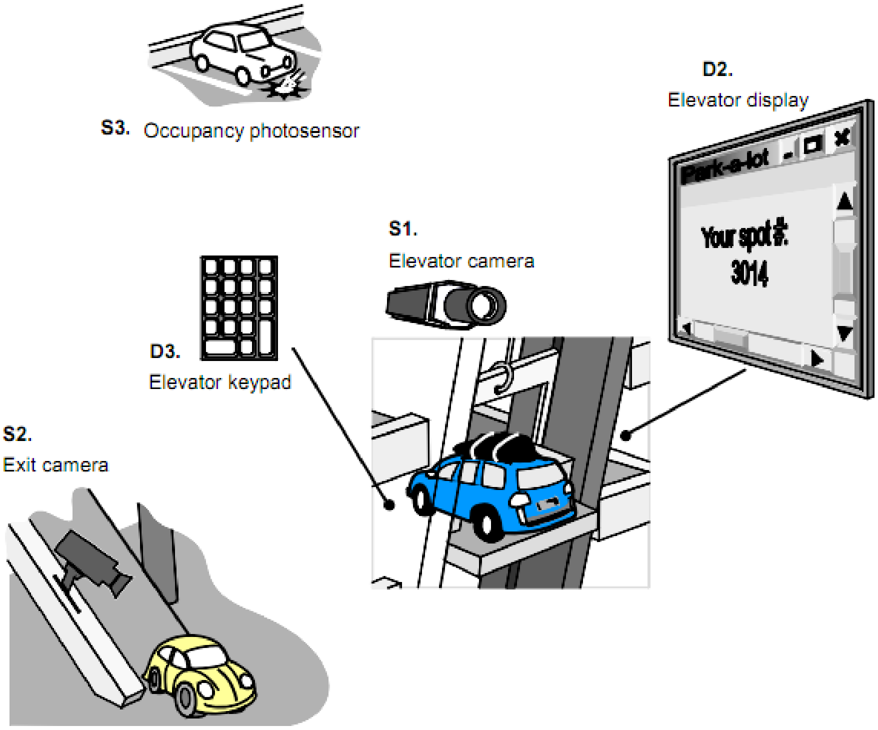
S3. There will be a sensor installed in every spot in the garage; this will help us determine whether the spot is available or not.

D1. This digital display allows walk-ins to check on the availability of the ground level parking, and has a built in credit card reader along it's side. If the ground level is full, the display will indicate so.

D2. The digital display at the elevator will display different messages according to the specific situation. The messages that might appear in this display are: denied access to upper levels for non-registered customers or change/edit in the reservation of a registered customer.

D3. The keypad will be used in the elevator in cases when the registration number is not recognized by the camera. In this scenario, the customer will have to key in the confirmation number given by the system at the time of the reservation.

Although, this is not specifically a device, there will be a one-way barrier at the end of the exit driveway so that no vehicles will try to enter the higher levels of the garage.



Types of Customers

1. **Registered Customers -** have an online account with Park-A-Lot, can make reservations online and receive monthly bills, and may park on the upper levels of the garage for reservations or the lower level as a walk-in.
2. **Unregistered Customers -** do not have an online account and cannot make reservations in advance. These customers must pay at the time they park, and may only park on the ground level.

Types of Reservations and Parking

1. **Recurring Reservation -** A reservation that is scheduled to recur on any day(s) of the week, starting from a start date and recurring until and end date.
2. **Confirmed Reservation** – A reservation that occurs only once, on the chosen date and time.

The following applies to all types of reservations:

1. The reservation must be scheduled starting either on an hour or on a half-hour (i.e. 8:00am or 8:30am, but not 8:45am), and the length must be in 30-minute increments.
2. The reservation may only be extended in 30-minute increments.
3. The customer is charged a fixed rate per hour (may include special discount pricing).
4. The customer is given a 30 minute, non-extendable grace period. If the customer misses his or her reservation, the customer will be charged a no-show penalty.

1. **Reservation Parking** – When a register customer arrives at the garage to fulfill a pre-existing reservation, either a recurring or confirmed reservation.
2. **Walk-In Parking** – When a customer (either registered or unregistered) arrives at the garage without an existing reservation. Parking for unregistered customers is dependent on the available spaces in the lowest level of the garage. Parking for registered customers depends on any available space in the garage.

Assumptions

Although our software solution will attempt to cover as many situations and scenarios as possible, the following general assumptions will be made.

A1. The camera's license-plate recognition system does not fail, meaning it is correct all the time, regardless if the plate is dirty or has damages. Also, if the registration number is not recognized by the system then is is assumed that the car does not correspond to any registered customer.

A2. If the elevator camera's license-plate recognition system does not identify the registration number and the customer fails to provide a correct one then the system will display a message on elevator display telling them to back up from the elevator. If this occurs then it is assumed that the customer obediently leaves the elevator.

A3. The sensors that detect the occupancy of the spots work correctly all the time, disregarding any malfunctioning of the devices. Also, every time a sensor detects occupancy it is because a vehicle is there and not another object.

A4. The elevator will lift the car to the corresponding deck and will not make any mistakes.

A5. The customer will not fail to park at his or her assigned parking spot.

A6. If the system recognizes the vehicle's registration number then it is assumed that the customer driving the car is a registered customer. Specific scenarios in which a non-registered customer borrows a car from a registered customer and when a car is stolen from a registered customer are not considered.

A7. Organizations and companies may register accounts, meaning there may be multiple vehicles and people contained within this account. Also, large organizations might make multiple reservations during the same time period so that multiple employees may utilize the parking garage.

A8. Lastly, the customer has access to his or her email and a cellphone. However it is not assumed that the customer will check his or her email frequently or that the customer has a smart phone capable of accessing web pages on the Internet or downloading apps.

A9. Parking garages are open 24 hours a day. A reservation can be made for any time of the day.

Business Policies

1. **Pricing -** A recurring reservation will cost less per hour than a confirmed reservation or walk-in, since the promise of using the spot (or facing a penalty) will increase profits. Confirmed reservations will be priced equal to walk-in reservations since there is no penalty for no-shows of a confirmed reservation.
2. **Cancellations -** It should be possible to cancel a reservation once it has been made but prior to the start date-time of the reservation. Any reservation can be canceled before the start date-time of the reservation, and that spot will then enter the pool of parking spaces eligible to be doled out to customers parking in the garage. Reservations cannot be canceled after the end of the grace period since the spot will have already been put back into the pool of rent-able spaces.
    a. Canceling a recurring reservation after the start date-time of the reservation but before the end of the grace period results in a charge to the registered customer’s account for the time the spot is held.
    b. Canceling a confirmed reservation after the start date-time of the reservation but before the end of the grace period results in no charge to your account.
    c. Any cancellation immediately places the spot back into the pool of eligible parking spaces.
3. **Minimum Parking Time -** The minimum parking time for any type of reservation will be 30 minutes. All parking times must be in 30 minute increments.
4. **Discounts -** Discounts should be given to customers who arrive and depart consistently on time. A solid performance record would be 90-95% accuracy. Statistical data will be collected on each registered customer and discounts will be given accordingly. Unregistered customers becoming registered customers will have a blank record, regardless of their past transactions with the system.

5. **Rain Check Credit -** If a customer arrives at the parking garage with a reservation but all spots are occupied, the customer will be issued a credit to his or her account in the amount of 110% of the price they expected to pay for the current reservation. This credit can be applied to defray the costs of future bills, but will never be given as a cash refund - more like store credit, which can only be used to pay for future parking in the garage.

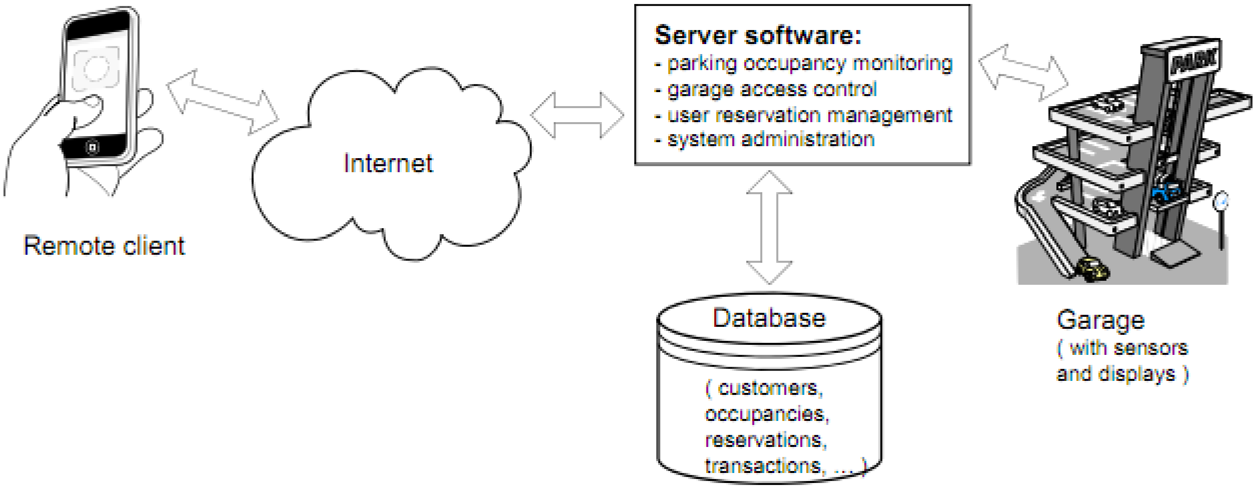
Prices and Fees

The following prices and fees will apply to the Park-A-Lot garage system.

Penalty Rate for Overstays	Billed at 150% of the regular hourly rate until car leaves the lot.
Not arriving during grace period	Billed 0.5 hours at the hourly rate.
Canceling a reservation within 30 minutes of the start time	Billed 0.5 hours at the hourly rate.

User Interaction Requirements

Below is an overview diagram of the Park-a-Lot system concept.



The system will have the following requirements and design specifications.

1. A database that will contain:
    a. All registered customers' information;
    b. All the parking reservations (past, current and future);
    c. State of all available parking spots in the garage;
    d. All vehicles registered to customer accounts;
    e. A record of all customer transactions (i.e. garage usage history, past reservations, punctuality or missed reservations);
    f. Pricing information;
2. A system administrator role which will be allowed to
    a. View the registered customers' profiles and customer statistics;
    b. Set the various prices and rates for the different services provided (i.e parking fee for a reserved interval, overstays, and the no-show fee).
3. The customer must be able to be able to make a reservation by selecting a date and time interval using the website for Park-a-Lot. If their desired time slot is available then the system should allow the customer to make the parking reservation.
4. If necessary or desired, the customer is able to modify his or her existing reservation(s) no later than 30 minutes before the start time.
5. If necessary or desired, the customer is able to extend his or her reservation end time no later than 30 minutes prior to the end of the reservation.
6. The customer needs to be notified of his or her parking spot number upon arrival at the garage. Since the Park-a-Lot system does not have a driver-guidance system, the customer will need this number to find the correct spot.

7. If, for any circumstance, the camera's recognition system fails to recognize the vehicle's registration number then the customer will be asked to log in using the keypad and his or her customer ID number to retrieve a reservation from his or her account.
8. If the customer uses a mobile phone with internet access or another mobile computing device to access Park-A-Lot, the software has to have an easy and simple interface that allows the customer to manage his or her account and reservations quickly. In order to accomplish this task, the number of inputs (data entries) necessary should be kept to a minimum.

Business Requirements Table

Identifier	Requirement
REQ0	Any person may create a registered customer account using the Park-A-Lot website. To register, a customer must supply an email address and password combo, their first and last name, and a valid credit card before they can make any reservations.
REQ1	A registered customer may park to fulfill a reservation. The elevator camera will recognize the plate on the car, and a spot number will be displayed to the customer in which they should park.
REQ2	A registered customer may park as a walk-in. If the car's license plate number is recognized then they will have to enter in the desired end time on the elevator keypad. If the license plate number is not recognized, the customer will have to enter his or her customer ID and password and then enter the desired park time. The system will then look for an available parking spot on any level of the garage.
REQ3	When an unregistered customer enters the garage their license plate number will not be recognized by the system and they will have to indicate that they wish to park as a walk-in. The system will then search for available parking spots in the ground level and assign one to the customer.
REQ4	In the scenario that the customer does not show up on time for a reservation then the spot will be held for a "grace period". If the customer arrives at the garage to park before the grace period expires then he or she will be able to park for the duration of the reservation but will be billed for the the entire period reserved. All types of reservations have a

	fixed grace period of one half-hour.
REQ5	Missing a reservation occurs when a customer fails to arrive before the end of the grace period. A missed reservation cannot be fulfilled, and the time of the grace period that the spot was held for will be billed to the customer. If the customer arrives for a reservation late he or she will be prompted to park as a walk in.
REQ6	In the scenario that the customer leaves the garage before his/her reservation expires then it is considered an understay. When this occurs the customer will still have to pay for the full reserved time. The sensor will recognize that the spot is vacant or available and will inform the system so that the spot can be assigned for another reservation.
REQ7	In the scenario that the customer does not leave on time and stays over his/her reserved time, this is called an overstay. The customer will be billed for any overage time at 1.5 times the hourly rate, known as the penalty rate.
REQ8	A registered customer may edit a reservation’s end time no later than half an hour before the current end time if and only if there is space available in the garage for the new end time. This procedure can be done unlimited times as long as there are available spots in the garage. The new end time can be extended or decreased by at most 90 minutes at a time.
REQ9	A customer may cancel a reservation up to 30 minutes before the reservation start time without being billed for any of the reservation. Cancellations within 30 minutes of the start time cause the customer to be billed for the grace period.
REQ10	Each registered customer can have multiple unfulfilled reservations under his/her customer account.
REQ11	In the scenario that a registered customer arrives to the garage but it is full because other customers have overstayed the system will ask a registered customer to leave without parking and he or she will be credited a rain check to his or her account. Unregistered customers will simply be told there is no vacancy.
REQ12	Registered customers will be billed at the end of a month for charges incurred in that month, and the system will create a statement which will be emailed to the customer containing all of the parking and penalty fees, with each parking reservation itemized on the bill. The bill can be paid with the credit-card on file with the customer’s account. The bill

	must be paid within 25 days of it being emailed to the customer.
REQ13	Each garage will have a system administrator who can set and change prices and penalty fees for that garage. The system administrator will also be able to view the garage history report which will include meaningful information regarding overstays, understays, etc.
REQ14	Registered Customers can register any number of vehicles under their account. Each vehicle must abide by the state's license plate format when registering. If a vehicle has a vanity license plate then the user will still be able to register that vehicle. A registered customer will also be allowed to edit or delete existing vehicles.
REQ15	Registered Customers can make a single reservation for the parking garage through their account on the website by entering the start time, end time, and date of a desired reservation. The customer will be notified if the garage is full and cannot handle a reservation for a given time or date.
REQ16	Registered Customers can reserve multiple parking spots at once. A recurring reservation is defined by the day(s) of the week on which it will recur, and the date by which the recurrence should end, in addition to the start time and end time of the reservation. Recurring reservations will be scheduled starting on the date selected, and then repeated on all selected days of the week until the end date is reached.
REQ17	Reservations must be made in 30 minutes increments, and must start either on the top of an hour or at the half-way point between hours.
REQ18	Vehicles may only be registered to one user at a time.
REQ19	If the license plate registered for the reservation differs from what license plate parked at the garage, the user will be notified during the next login to the website and asked to register the vehicle with his or her account.
REQ20	Registered Customer can arrive early for a reservation, and be offered the option to park as a walk-in and have that parking carry over to their reservation.

Identifier	Requirement	Use Case
REQ0	Any person may create a registered customer account using the Park-A-Lot website. To register, a customer must supply an email address and password combo, their first and last name, and a valid credit card before they can make any reservations.	UC-5
REQ1	A registered customer may park to fulfill a reservation. The elevator camera will recognize the plate on the car, and a spot number will be displayed to the customer in which they should park.	UC-2
REQ2	A registered customer may park as a walk-in. If the car's license plate number is recognized then they will have to enter in the desired end time on the elevator keypad. If the license plate number is not recognized, the customer will have to enter his or her customer ID and password and then enter the desired park time. The system will then look for an available parking spot in the ground level.	UC-2
REQ3	When an unregistered customer enters the garage their license plate number will not be recognized by the system and they will have to indicate that they wish to park as a walk-in. The system will then search for available parking spots in the ground level and assign one to the customer.	UC-2
REQ4	In the scenario that the customer does not show up on time for a reservation then the spot will be held for a "grace period". If the customer arrives at the garage to park before the grace period expires then he or she will be able to park for the duration of the reservation but will be billed for the the entire period reserved. All types of reservations have a fixed grace period of one half-hour.	UC-2
REQ5	Missing a reservation occurs when a customer fails to arrive before the end of the grace period. A missed reservation cannot be fulfilled, and the time of the grace period that the spot was held for will be billed to the customer. If the customer arrives for a reservation late he or she will be prompted to park as a walk in.	UC-2
REQ6	In the scenario that the customer leaves the garage before his/her reservation expires then it is considered an understay. When this occurs the customer will still have to pay for the full reserved time. The sensor will recognize that the spot is vacant or available and will inform the system so that the spot can be assigned for another reservation.	UC-2

REQ7	In the scenario that the customer does not leave on time and stays over his/her reserved time, this is called an overstay. The customer will be billed for any overage time at 1.5 times the hourly rate, known as the penalty rate.	UC-2
REQ8	A registered customer may edit a reservation's end time no later than half an hour before the current end time if and only if there is space available in the garage for the new end time. This procedure can be done unlimited times as long as there are available spots in the garage. The new end time can be extended or decreased by at most 90 minutes at a time.	UC-7
REQ9	A customer may cancel a reservation up to 30 minutes before the reservation start time without being billed for any of the reservation. Cancellations within 30 minutes of the start time cause the customer to be billed for the grace period.	UC-7
REQ10	Each registered customer can have multiple unfulfilled reservations under his/her customer account.	UC-4
REQ11	In the scenario that a registered customer arrives to the garage but it is full because other customers have overstayed the system will ask a registered customer to leave without parking and he or she will be credited a rain check to his or her account. Unregistered customers will simply be told there is no vacancy.	UC-13
REQ12	Registered customers will be billed at the end of a month for charges incurred in that month, and the system will create a statement which will be emailed to the customer containing all of the parking and penalty fees, with each parking reservation itemized on the bill. The bill can be paid with the credit-card on file with the customer's account. The bill must be paid within 25 days of it being emailed to the customer.	UC-13
REQ13	Each garage will have a system administrator who can set and change prices and penalty fees for that garage. The system administrator will also be able to view the garage history report which will include meaningful information regarding overstays, understays, etc.	UC-6, UC-11, UC-12
REQ14	Registered Customers can register any number of vehicles under their account. Each vehicle must abide by the state's license plate format when registering. If a vehicle has a vanity license plate then the user will still be able to register that vehicle. A registered customer will also be allowed to edit or delete existing vehicles.	UC-8, UC-9

REQ15	Registered Customers can make a single reservation for the parking garage through their account on the website by entering the start time, end time, and date of a desired reservation. The customer will be notified if the garage is full and cannot handle a reservation for a given time or date.	UC-1
REQ16	Registered Customers can reserve multiple parking spots at once. A recurring reservation is defined by the day(s) of the week on which it will recur, and the date by which the recurrence should end, in addition to the start time and end time of the reservation. Recurring reservations will be scheduled starting on the date selected, and then repeated on all selected days of the week until the end date is reached.	UC-1
REQ17	Reservations must be made in 30 minutes increments, and must start either on the top of an hour or at the half-way point between hours.	UC-1
REQ18	Vehicles may only be registered to one user at a time.	UC-8
REQ19	If the license plate registered for the reservation differs from what license plate parked at the garage, the user will be notified during the next login to the website and asked to register the vehicle with his or her account.	UC-8, UC-2
REQ20	Registered Customer can arrive early for a reservation, and be offered the option to park as a walk-in and have that parking carry over to their reservation.	UC-2

Glossary of Terms

Cancelled Reservation - a reservation that has been cancelled, either by the customer at least thirty minutes prior to the start date time of the reservation or by the expiration of the grace period.

Confirmed Reservation - a reservation made in advance with a fixed grace period of thirty minutes.

Customer ID - the email address a customer uses to register with the online Park-A-Lot system.

Elevator Camera - an image capturing camera capable of utilizing image recognition to determine the license plate number on a vehicle inside the parking garage elevator.

Elevator Display - an LCD screen located inside the parking garage elevator for displaying information.

Elevator Keypad - a 102/105-key keyboard located inside the parking garage elevator for inputting information.

Exit Camera - same as elevator camera, and capable of using image recognition on license plates of vehicles exiting the garage.

Extended Reservation - A reservation that has been extended past the original end time.

Grace Period - the amount of time a customer is allowed to be late to check in for a reservation before the reservation is cancelled.

Missed Reservation - when a customer fails to arrive before the grace period is over.

No-Show Penalty - A fee a customer is assessed if he or she fails to fulfill a reservation before the expiration of the grace period.

No Vacancy - The parking garage is full and is not walk-in parking at this time.

Overstay - when a customer fails to depart at the scheduled time.

Penalty Rate - the rate a customer who overstays his or her reservation is charged for that overage time, 1.5 times the hourly rate.

Rain Check Credit - it is a credit given to the customer when he or she arrives at the parking garage and no spots are available to park in.

Reservation - an arrangement to park in a parking garage for a fixed amount of time at a certain fee per hour.

Recurring Reservation - a reservation, made in advance with a variable grace period, that occurs on some regularly repeating schedule.

Reservation Parking - when a customer parks at the garage to fulfill a pre-existing reservation (either recurring or confirmed).

Spot Sensors - sonar sensors capable of determining whether a parking spot inside the garage is occupied with a car or not.

System Administrator - person who will have deep access to the system, and be able to alter business requirements such a parking prices.

Understay- when a customer leaves the garage before his scheduled time of departure.

Walk In Parking - when a customer parks on-the-spot at a parking garage without a pre-existing reservation.

Functional Requirements Specification

Stakeholders

A stakeholder is anyone who has interest in this system (users, managers, sponsors, etc.).

1. Registered Customers
2. Unregistered Customers
3. Garage Owner / Company
4. System Administrator
5. Security Personnel
6. Nearby Retail / Entertainment

Actors and Goals

An actor is anyone who will directly interact with the system. The two types of actors are initiating and participating.

1. Initiating
    a. Registered Customer
    b. Unregistered Customer
    c. System Administrator
    d. Timer
2. Participating
    a. Elevator Keypad
    b. Elevator Display
    c. Elevator Camera
    d. Spot Sensors
    e. Exit Camera
    f. Database
    g. Event Log

Use Cases

Casual Description of Use Cases

UC-1 Reserve

A registered customer wants to make a reservation (either one-time or recurring). After being authenticated by the system, the system will show a reservation form. The user will input all required information (date and time, length of stay, etc.) and submit. The system will then validate the information submitted and create the users reservation, storing the data in the database.

UC-2 Park

A customer (registered or unregistered) arrives at the garage and wants to park. The customer drives up to the elevator, where the elevator camera will read the customers license plate number.

If the customer is a registered customer, the elevator display will show the customers reservation information as well as assign a spot to park. The elevator will then bring the car to the correct floor where the spot resides so that the user can park.

If the customer is an unregistered customer, the system will check for open spots on the ground floor, and inform the customer of open spots. The customer will then swipe his/her credit card before driving though to park.

Finally, if the customer arrives early to his or her reservation, the following applies:

1. Check their license plate for a future reservation, and if found ask if they are there for that reservation.
    a. If yes, and if the upcoming reservation is within some time limit (30 minutes before the start time), then if there is available parking they may park, and be charged for the additional time at the rate per hour of their soon to be occurring reservation.
    b. If the user arrives very early (more than 30 minutes before start time), they will have to park as a walk-in until the start time of their reservation. This would also require us to make a note that the walk-in reservation would coincide directly with their upcoming scheduled reservation, meaning that the customer should not be expected to leave the walk-in parking and re-enter the garage to fulfill their scheduled reservation. We will assume the walk-in parking will continue directly to the actual reservation scheduled. The billing should be at a walk-in rate for the walk-in, and at the regular rate for the reservation.
    c. Finally, if no spots are available, the customer will be told they cannot park at this time.
2. If the license plate isn't recognized, we prompt for credentials, and follow the instructions given in (1) if the customer is registered with a reservation in the future for that day.

3. If all else fails, the customer can park as a walk-in.

UC-3 Manage Account
A registered customer wants to change their account details (email, password, address, credit card info, etc.). After first being authenticated by the system, the customer will be presented with a pre-filled form with all of their existing information. The customer will make whatever changes he/she wishes to make and submit. The system will validate the information and save it in the database.

UC-4 View Reservations
A registered customers wants to make changes to existing reservations on their account. After first being authenticated, the user will be presented with a list of all of their current and future reservations as well as an option to edit a reservation. The edit reservation option is a sub use case.

UC-5 Register
An unregistered customer wants to register a new account. The system will show a registration form, which the user will fill out and submit. The system will then validate the submitted information (name, address, email, password, credit card info) and store it in the database, assigning the user a unique customer id.

UC-6 Manage Garage
A system administrator wants to manage the garage remotely. After being authenticated by the system, the administrator will be presented with options to set parking prices, inspect usage history, as well as view current usage. All options are sub-use cases described later.

UC-7 Edit Reservation
A registered customer wants to edit a reservation (change the end time or cancel it). After first being authenticated by the system, the customer will choose which reservation they wish to change and submit. The system will then present the user with a form to either change the end time of the reservation or cancel it. Canceling it can only be done at least 30 minutes before the start of the reservation and changing the end time of a reservation must be done at least 30 minutes before the original end time. The system will then mark the reservation as edited if the end time is changed or canceled if it was canceled and inform the customer.

UC-8 Register Vehicle
A registered customer wants to register a vehicle for his/her account. After first being authenticated by the system, the system will show a form, which the user will fill out and submit. The system will then validate the submitted information (license plate number, state, color) and store it in the database, assigning the user a unique vehicle id.

UC-9 Edit Vehicle

A registered customer wants to change the details of a registered vehicle (license plate number, state, color). After first being authenticated by the system, the customer will be presented with a list of their current vehicles. The customer will choose one to edit. The system will then present the customer with a form to fill out. The customer will also have the option to delete the vehicle. The customer will make whatever changes he/she wishes to make and submit. The system will validate the information and save it in the database.

UC-10 Authenticate User

A registered customer wants to log in to the system. The system will present a log in form (email, password) which the customer will fill out and submit. The system will search for and find the customer in the database. The system will then start a session for the user, which will last until the user logs out or closes his/her browser.

UC-11 Set Prices

A system administrator wants to set the prices for his/her garage. After first being authenticated by the system, the system will present a form displaying all acceptable***** prices. The administrator will then choose a price and submit. The system will then validate and store the price in the database.

***** Acceptable prices would need to be shown so that the administrator cannot set arbitrarily high (or low) prices.

UC - 12 Inspect Usage History

A system administrator wants to view the usage history of his/her garage. After first being authenticated by the system, the system will gather all statistical data about their garage (overstay/understay percentages, etc.)

UC - 13 Monthly Billing

The timer will perform the use case every 30 days. For each registered customer, the timer will extract all reservations not canceled for the previous month and create a bill out of the data but adding up the reservation charges and penalties. The bill will then be emailed to the registered customers.

Actor’s Goal Tables

Initiating Actors	Actor’s Goal	Use Case Name
Registered Customer	To obtain a reservation for a parking spot for a given duration in advance.	Reserve (UC-1)
Registered Customer	To park in a parking spot to fulfill a reservation.	Park (UC-2)
Registered Customer	To manage the details of customer account.	ManageAccount (UC-3)
Registered Customer	To view existing reservations in the customer’s account.	ViewReservations (UC-4)
Registered Customer	To create a new vehicle with a valid license plate number, state, and color.	RegisterVehicle (UC-8)
Registered Customer	To edit an existing vehicle’s license plate number, state, or color, or to delete it.	EditVehicle (UC-9)
Unregistered Customer	To park in a parking spot for a given duration.	Park (UC-2)
Unregistered Customer	To register for an account and become a Registered Customer.	Register (UC-5)
System Admin	To manage the parking garage prices and view parking usage history and statistics.	ManageGarage (UC-6)
Timer	To send out a monthly bill to each customer including all of their reservations charges plus penalties.	MonthlyBilling (UC-12)

Participating Actor	Actor’s Goal	Use Case Name
Elevator Keypad	To consume information.	UC-2
Elevator Display	To display information.	UC-2
Elevator Camera	To obtain license plate info from car.	UC-2
Spot Sensors	To determine if spot is occupied.	UC-2
Exit Camera	To determine when a car leaves.	UC-2
Database	To store and manage data.	All use cases

Fully-Dressed Description of Use Cases

Use Case UC-1:	**Reserve**
Related Requirements:	REQ15, REQ16, REQ17
Initiating Actor:	Registered Customer
Actor's Goal:	To reserve a parking spot for a future date and time.
Participating Actors:	Database
Preconditions:	Registered Customer is currently logged into the System. *include::AuthenticateUser* (UC-10)
Postconditions:	The System reserves the requested date and time for the customer in the Database.

Flow of Events for Main Success Scenario:

→ 1. **Registered Customer** selects menu option "Make Reservation".

→ 2. **Registered Customer** selects the desired date, start time, and end time for the reservation, along with any additional options (i.e. for a recurring reservation the customer may wish to extend the grace period).

← 3. **System** (a) checks the reservation **Database** for available reservations, (b) notifies **Registered Customer** that the reservation is made, and (c) updates the **Database** to include the new reservation.

Flow of Events for Extensions (Alternate Scenarios):

3a. **System** cannot find an available reservation for the specified date and time.

← 1. **System** (a) logs the attempted reservation and (b) signals to the **Registered Customer**.

→ 2. **Registered Customer** selects a different date and time to make a reservation.

3. Same as in step 4.

3b. **System** finds a conflicting reservation from **Registered Customer**'s account.

← 1. **System** notifies customer that he or she already has a reservation or part of a reservation during that date and time.

→ 2(a). **Registered Customer** can choose to cancel the existing reservation in favor of the new reservation. *include::EditReservation* (UC-7). Then same as in step 4.

→ 2(b). **Registered Customer** can choose to book overlapping reservations. Then same as in step 4.

Use Case UC-2:	**Park**
Related Requirements:	REQ1, REQ2, REQ3, REQ4, REQ5, REQ6, REQ7, REQ19, REQ20
Initiating Actor:	Registered Customer, Unregistered Customer (collectively Customer)
Actor's Goal:	To park in the garage.
Participating Actors:	Elevator Display, Elevator Camera, Elevator Keypad, Spot Sensor Exit Camera, Database
Preconditions:	The elevator is currently empty.
Postconditions:	System marks reservation as completed in database.

Flow of Events for Main Success Scenario:

→ 1. **Customer** enters the elevator.

2. **Elevator Camera** recognizes license plate of vehicle.

← 3. **Elevator Display** displays any reservations returned from the **Database** the customer may have for the current date.

→ 4. **Customer** selects a reservation.

← 5. **System** (a) assigns an optimal spot to the **Customer** and (b) displays the parking spot on the **Elevator Display**.

→ 6. **Customer** exits the elevator.

← 7. **Spot Sensor** notifies the **System** when a vehicle is parked in the assigned parking space.

← 8. **Spot Sensor** notifies the **System** when a vehicle is no longer parked in the assigned space.

9. **Customer** exits parking garage and is recorded by **Exit Camera**.

Flow of Events for Extensions (Alternate Scenarios):

2a. **Elevator Camera** fails to recognize license plate of vehicle.

← 1. **Elevator Display** prompts **Customer** for either (1) a registered customer ID or (2) a method of payment for a walk in.

→ 1(a). **Registered Customer** enters ID number, password, and license plate number into
Elevator Keypad.

1(b). *include::AuthenticateUser* (UC-10)

1(c). Same as in step 3.

→ 2(a). **Customer** slashes credit card to pay.

← 2(b). **Elevator Display** prompts user for reservation length.

→ 2(c). **Customer** enters reservation duration into **Elevator Keypad**.

2(d). Same as in step 5.

3a. **Customer** arrives early for reservation.

1. If the reservation start time is within 30 minutes of the current time, then

→ 1(a). If parking spaces are available in the garage, then modify the start time of the reservation by 30 minutes so that **Customer** may park immediately.

1(b). Same as in 3.

2. If the reservation start time is more than 30 minutes from the current time, then

2(a). Same as in (3b).

2(b). Make note that the **Customer** will not be required to leave the assigned spot to fulfill his or her upcoming reservation.

3b. **Customer** does not have any existing reservations.

← 1. **System** prompts user for reservation length.

→ 2. **Customer** enters reservation duration into **Elevator Keypad**.

3. Same as in step 5.

3c. **Customer** arrives after grace period has expired.

← 1. **Customer** is informed that grace period has expired and is offered the chance to park as a walk-in if space is available.

2. Same as in 5.

Use Case UC-3: Manage Account

Related Requirements:	None
Initiating Actor:	Registered Customer
Actor's Goal:	To edit the details of a customer's account.
Participating Actors:	Database
Preconditions:	Registered Customer is currently logged in to the system. *include::AuthenticateUser* (UC-10)
Postconditions:	Changes to Database are committed.

Flow of Events for Main Success Scenario:

→ 1. **Registered Customer** selects menu option "Manage Account".

← 2. **System** (a) displays current user account details, and (b) prompts **Registered Customer** to make changes to desired fields.

→ 3. **Registered Customer** makes the necessary changes to the form.

← 4. **System** verifies that the changes made are valid.

Use Case UC-4: **View Reservations**

Related Requirements:	REQ10
Initiating Actor:	Registered Customer
Actor's Goal:	To view existing reservations and edit any reservations.
Participating Actors:	Database
Preconditions:	Registered Customer is currently logged in to the system. *include::AuthenticateUser* (UC-10)
Postconditions:	Changes to Database are committed.

Flow of Events for Main Success Scenario:

→ 1. **Registered Customer** selects menu option “View Reservations”.

← 2. **System** displays any active or future reservations returned from the **Database** the customer may have and displays options for editing reservations.

→ 3. **Registered Customer** selects one of the reservations or multiple reservations (if they are recurring) to edit and follows the corresponding instructions or does not select any reservation.

← 4. **System** (a) stores the changes made in the Database, and (b) signals to the **Registered Customer** the successful change.

Flow of Events for Extensions (Alternate Scenarios):

3a. Selected activity entails editing the reservation.

1. *include::EditReservation*(UC-7).

Use Case UC-5: **Register**

Related Requirements:	REQ0
Initiating Actor:	Unregistered Customer
Actor's Goal:	To create an account and become a Registered Customer.
Participating Actors:	Database
Preconditions:	Unregistered Customer has a valid email address with which to register.
Postconditions:	The System stores all of the newly Registered Customer's information in the Database.

Flow of Events for Main Success Scenario:

→ 1. **Unregistered Customer** accesses **System** and selects menu option "Create Account".

→ 2. **Unregistered Customer** fills in personal info: name, address, state, zip, phone number, email address, password, billing credit card number and submits the info.

← 3. **System** (a) checks that all fields have been filled in, (b) verifies the email address is valid and unique within the **Database**, (c) verifies the credit card information is valid, and (d) updates the database to include the new **Registered Customer**.

→ 4. New **Registered Customer** can now make reservations in advance.

Flow of Events for Extensions (Alternate Scenarios):

3a. **System** identifies that not all of the fields have been filled in on the registration form.

← 1. **System** (a) detects error and (b) signals to the **Unregistered Customer** that they must complete the form and resubmit.

→ 2. **Unregistered Customer** fills in the missing data fields and resubmits the form.

3. Same as in step 3.

3b. **System** identifies that email address is invalid or has already been registered with the website

← 1. **System** (1) notifies the **Unregistered Customer** that the email address is invalid prompts the **Unregistered Customer** to change that information or (2) detects that the email address has already been registered in the system and alerts the **Unregistered Customer** that the email address has already been registered and to attempt to log into the account.

→ 1(a). **Unregistered Customer** changes the email address field and resubmits the form.

1(b). Same as in step 3.

2(a). **Registered Customer** leaves registration area.

3c. **System** could not verify the credit card information.

← 1. **System** prompts the **Unregistered Customer** to re-enter their credit card information.

→ 2. **Unregistered Customer** changes the information in the credit card field and resubmits.

3. Same as in step 3.

Use Case UC-6: **Manage Garage**

Related Requirements:	REQ13
Initiating Actor:	System Admin
Actor's Goal:	To set parking prices or inspect usage history.
Participating Actors:	Database, Event Log
Preconditions:	System Admin is currently logged in to the system. *include::AuthenticateUser* (UC-10)
Postconditions:	Changes to System are committed and System Admin is logged out.

Flow of Events for Main Success Scenario:

→ 1. **System Admin** selects menu option “Manage Garage”.

← 2. **System** displays options for: (a) setting prices, (b) inspecting parking usage history.

→ 3. **System Admin** selects one of the options from Step 2 and performs management activities.

→ 4. **System Admin** commits changes to **System**.

5. **System** verifies and commits changes to **Database**.

6. **System Admin** logs out of his or her account.

Flow of Events for Extensions (Alternate Scenarios):

3a. Selected activity entails setting parking garage prices.

1. *include::SetPrices* (UC-11).

3b. Selected activity entails viewing access history.

1. *include::InspectUsageHistory* (UC-12).

Use Case UC-7: **EditReservation** (sub-use case)

Related Requirements:	REQ8, REQ9
Initiating Actor:	Registered Customer
Actor's Goal:	To extend an existing reservation.
Participating Actors:	Database
Preconditions:	Some reservations exist for the Registered Customer in the Database.
Postconditions:	Extended reservation is marked as extended in the Database and the start and/or end time is updated accordingly.

Flow of Events for Main Success Scenario:

← 1. **System** (a) displays reservation(s) that the **Registered Customer** selected, and (b) prompts the **Registered Customer** to change the end time or cancel the reservation(s).

→ 2. **Registered Customer** makes the desired selections / changes.

← 3. **System** (a) checks the database to see if the extensions can be made, and (b) notifies the **Registered Customer** that the reservations have been extended.

4. **System** updates the database to include the changes to the reservation(s).

Flow of Events for Extensions (Alternate Scenario):

3a. **System** identifies that the extension cannot be made due to reservation conflict.

← 1. **System** notifies the **Registered Customer** that the parking deck is completely booked and the reservation cannot be extended.

2. Same as in step 1 above.

Use Case UC-8: **RegisterVehicle**

Related Requirements:	REQ14
Initiating Actor:	Registered Customer
Actor's Goal:	To register a vehicle.
Participating Actors:	Database

Preconditions:	Registered Customer is currently logged in to the system. *include::AuthenticateUser* (UC-10
Postconditions:	The system stores the new vehicle information in the database.

Flow of Events for Main Success Scenario:

→ 1. **Registered Customer** selects menu option “Register Vehicle”.

→ 2. **Registered Customer** enters the license plate number, the state, and the color of the vehicle.

← 3. **System** verifies that the license plate number is valid based on the state selected.

4. **System** stores the new vehicle information in the database.

Flow of Events for Extensions (Alternate Scenario):

3a. **System** identifies that the license plate number is not valid.

← 1. **System** notifies the **Registered Customer** that the license plate number is not valid.

2. Same as in step 2 above.

Use Case UC-9:	**EditVehicle**

Related Requirements:	REQ14
Initiating Actor:	Registered Customer
Actor’s Goal:	To register a vehicle.
Participating Actors:	Database
Preconditions:	Registered Customer is currently logged in to the system. *include::AuthenticateUser* (UC-10
Postconditions:	The system stores the new vehicle information in the database.

Flow of Events for Main Success Scenario:

→ 1. **Registered Customer** selects menu option “Register Vehicle”.

→ 2. **Registered Customer** enters the license plate number, the state, and the color of the vehicle.

← 3. **System** verifies that the license plate number is valid based on the state selected.

4. **System** stores the new vehicle information in the database.

Flow of Events for Extensions (Alternate Scenario):

3a. **System** identifies that the license plate number is not valid.

← 1. **System** notifies the **Registered Customer** that the license plate number is not valid.

2. Same as in step 2 above.

Use Case UC-10: **AuthenticateUser** (sub-use case)

Related Requirements:	None
Initiating Actor:	Registered Customer, System Admin (collectively User)
Actor's Goal:	To be positiviely identified by the system.
Participating Actors:	Database
Preconditions:	The database contains user account information.
Postconditions:	None

Flow of Events for Main Success Scenario:

← 1. **System** prompts the user for their customer ID and password.

→ 2. **User** supplies a valid customer ID and password.

← 3. **System** (a) verifies that the customer ID and password are valid, and (b) signals to the user the identification validity.

Flow of Events for Extensions (Alternate Scenarios):

2a. **User** enters an invalid customer ID and password combination.

← 1. **System** (a) detects error, (b) marks a failed attempt, and (c) and signals to the **User** the credentials are invalid.

← 1(a). **System** (a) detects that the count of failed attempts exceed the maximum number, (b) informs the user to try again in 15 minutes, and (c) exits the screen.

→ 2. **User** supplies a valid customer ID and password.

3. Same as in step 3.

Use Case UC-11: **Set Prices** (sub-use case)

Related Requirements:	REQ13
Initiating Actor:	System Admin
Actor's Goal:	To set the parking prices and penalty fees for a particular garage.
Participating Actors:	Database
Preconditions:	System Admin is currently logged into the system. *include::AuthenticateUser* (UC-9)
Postconditions:	Price changes are stored in the Database.

Flow of Events for Main Success Scenario:

→ 1. **System Admin** selects the menu option "Set Prices".

← 2. **System** prompts for the garage location.

→ 3. **System Admin** selects the appropriate garage(s).

← 4. **System** prompts for pricing options including recurring reservation parking rate, confirmed reservation parking rate, penalty fees, etc.

→ 5. **System Admin** enters in the desired prices.

← 6. **System** prompts the user to confirm the desired changes.

→ 7. **System Admin** selects "Yes, confirm the pricing changes".

← 8. **System** updates the **Database** to include the new pricing options for the garage.

Flow of Events for Extensions (Alternate Scenario):

7a. **System Admin** selects "No, I have made a mistake".

1. Same as in step 4..

Use Case UC-12: **Inspect Usage History** (sub-use case)

Related Requirements:	REQ13
Initiating Actor:	System Admin
Actor's Goal:	To examine the usage history of a particular garage.
Participating Actors:	Database
Preconditions:	System Admin is currently logged into the system.

	include::AuthenticateUser (UC-9)
Postconditions:	None

Flow of Events for Main Success Scenario:

→ 1. **System Admin** selects the menu option “Inspect Usage History”.

← 2. **System** prompts for search criteria including garage location, start date, and end date.

→ 3. **System Admin** specifies the search criteria and submits.

← 4. **System** prepares a database query that best matches the actor’s search criteria and retrieves the records from the **Database**.

→ 5. **Database** returns the matching records.

← 6. **System** (a) additionally filters the retrieved records to match the actor’s search criteria, (b) renders the remaining records for display, and (c) shows the result for the **System Admin** to view.

Use Case UC-13: **MonthlyBilling**

Related Requirements:	REQ11, REQ12
Initiating Actor:	Timer
Actor’s Goal:	To generate and send a monthly bill to every registered customer in the database.
Participating Actors:	Database
Preconditions:	It is the last day of the month.
Postconditions:	None

Flow of Events for Main Success Scenario:

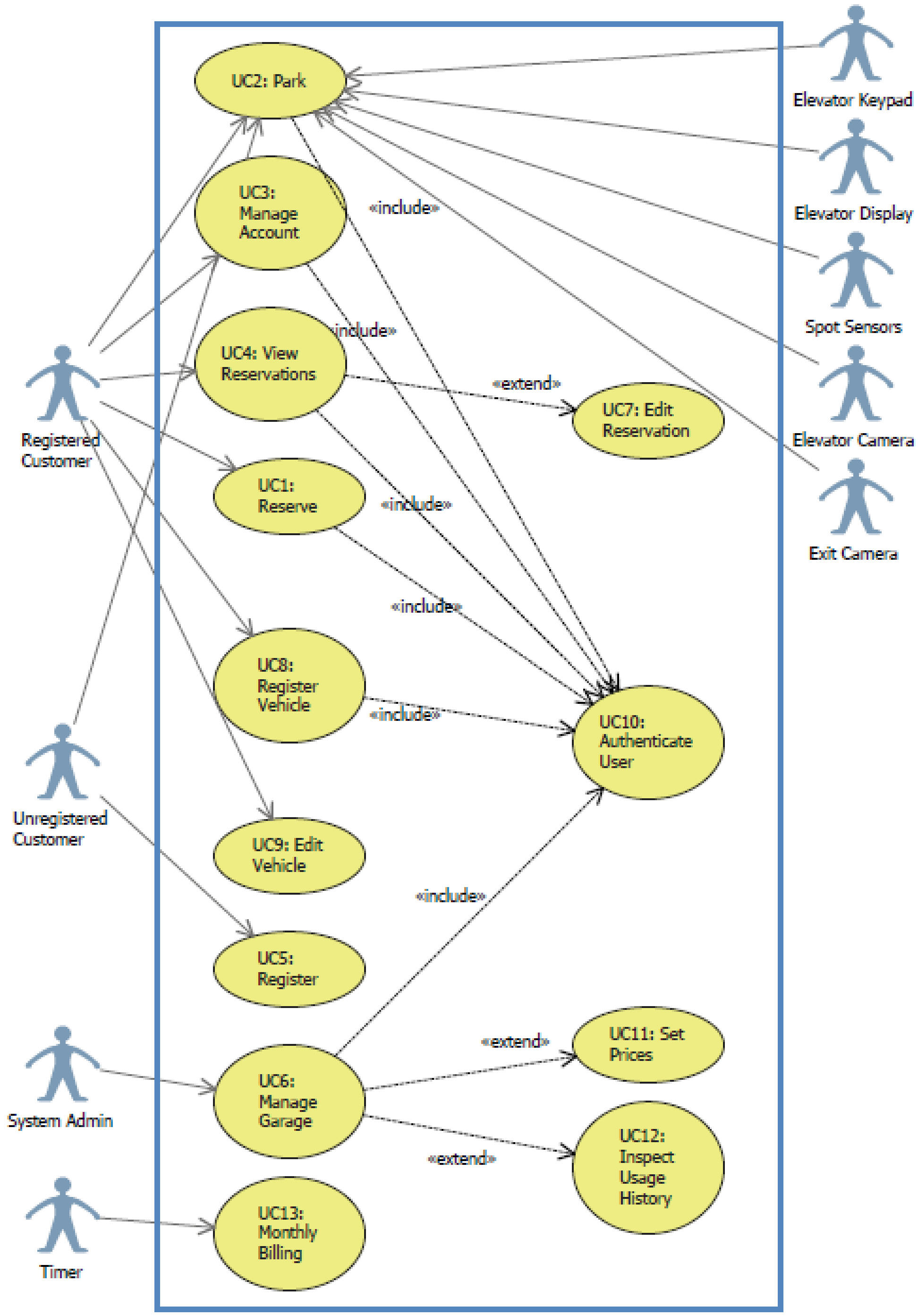
→ 1. **Timer** notifies the **System** that it is the last day of the month

← 2. **System** queries the **Database** for all registered customer IDs.

← 3. **System** queries the **database** for all accumulated parking hours and fees for each registered customer.

← 4. **System** (a) calculates the total charges (b) generates a bill from the calculations, and © emails one to each registered customer.

Use Case Diagram



System Requirements - Use Case Traceability Matrix

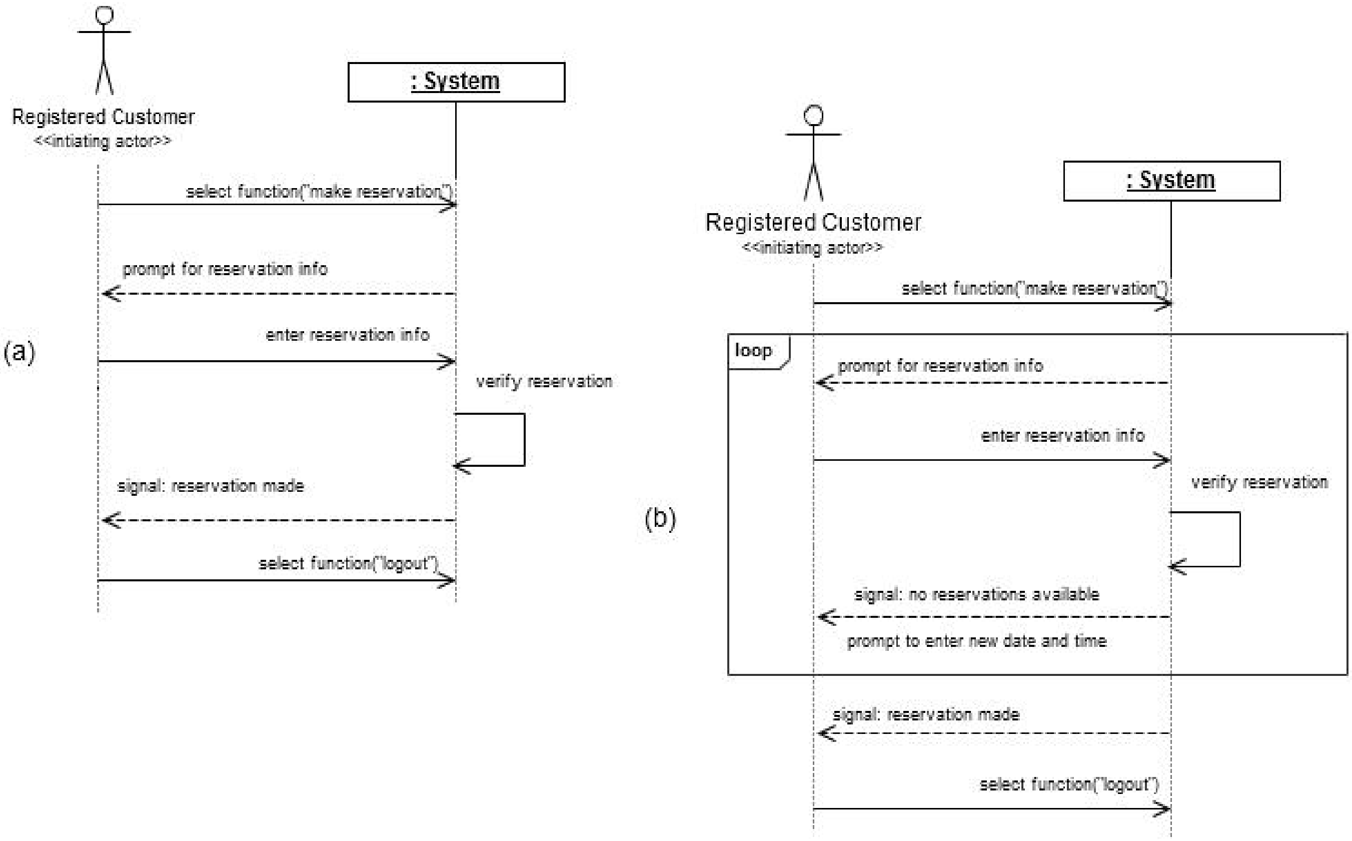
Identifier	Requirement	Use Case
REQ0	Any person may create a registered customer account using the Park-A-Lot website. To register, a customer must supply an email address and password combo, their first and last name, and a valid credit card before they can make any reservations.	UC-5
REQ1	A registered customer may park to fulfill a reservation. The elevator camera will recognize the plate on the car, and a spot number will be displayed to the customer in which they should park.	UC-2
REQ2	A registered customer may park as a walk-in. If the car's license plate number is recognized then they will have to enter in the desired end time on the elevator keypad. If the license plate number is not recognized, the customer will have to enter his or her customer ID and password and then enter the desired park time. The system will then look for an available parking spot in the ground level.	UC-2
REQ3	When an unregistered customer enters the garage their license plate number will not be recognized by the system and they will have to indicate that they wish to park as a walk-in. The system will then search for available parking spots in the ground level and assign one to the customer.	UC-2
REQ4	In the scenario that the customer does not show up on time for a reservation then the spot will be held for a “grace period”. If the customer arrives at the garage to park before the grace period expires then he or she will be able to park for the duration of the reservation but will be billed for the the entire period reserved. All types of reservations have a fixed grace period of one half-hour.	UC-2
REQ5	Missing a reservation occurs when a customer fails to arrive before the end of the grace period. A missed reservation cannot be fulfilled, and the time of the grace period that the spot was held for will be billed to the customer. If the customer arrives for a reservation late he or she will be prompted to park as a walk in.	UC-2
REQ6	In the scenario that the customer leaves the garage before his/her reservation expires then it is considered an understay. When this occurs the customer will still have to pay for the full reserved time.	UC-2

	The sensor will recognize that the spot is vacant or available and will inform the system so that the spot can be assigned for another reservation.	
REQ7	In the scenario that the customer does not leave on time and stays over his/her reserved time, this is called an overstay. The customer will be billed for any overage time at 1.5 times the hourly rate, known as the penalty rate.	UC-2
REQ8	A registered customer may edit a reservation's end time no later than half an hour before the current end time if and only if there is space available in the garage for the new end time. This procedure can be done unlimited times as long as there are available spots in the garage. The new end time can be extended or decreased by at most 90 minutes at a time.	UC-7
REQ9	A customer may cancel a reservation up to 30 minutes before the reservation start time without being billed for any of the reservation. Cancellations within 30 minutes of the start time cause the customer to be billed for the grace period.	UC-7
REQ10	Each registered customer can have multiple unfulfilled reservations under his/her customer account.	UC-4
REQ11	In the scenario that a registered customer arrives to the garage but it is full because other customers have overstayed the system will ask a registered customer to leave without parking and he or she will be credited a rain check to his or her account. Unregistered customers will simply be told there is no vacancy.	UC-13
REQ12	Registered customers will be billed at the end of a month for charges incurred in that month, and the system will create a statement which will be emailed to the customer containing all of the parking and penalty fees, with each parking reservation itemized on the bill. The bill can be paid with the credit-card on file with the customer's account. The bill must be paid within 25 days of it being emailed to the customer.	UC-13
REQ13	Each garage will have a system administrator who can set and change prices and penalty fees for that garage. The system administrator will also be able to view the garage history report which will include meaningful information regarding overstays, understays, etc.	UC-6, UC-11, UC-12
REQ14	Registered Customers can register any number of vehicles under	UC-8, UC-9

	their account. Each vehicle must abide by the state’s license plate format when registering. If a vehicle has a vanity license plate then the user will still be able to register that vehicle. A registered customer will also be allowed to edit or delete existing vehicles.	
REQ15	Registered Customers can make a single reservation for the parking garage through their account on the website by entering the start time, end time, and date of a desired reservation. The customer will be notified if the garage is full and cannot handle a reservation for a given time or date.	UC-1
REQ16	Registered Customers can reserve multiple parking spots at once. A recurring reservation is defined by the day(s) of the week on which it will recur, and the date by which the recurrence should end, in addition to the start time and end time of the reservation. Recurring reservations will be scheduled starting on the date selected, and then repeated on all selected days of the week until the end date is reached.	UC-1
REQ17	Reservations must be made in 30 minutes increments, and must start either on the top of an hour or at the half-way point between hours.	UC-1
REQ18	Vehicles may only be registered to one user at a time.	UC-8
REQ19	If the license plate registered for the reservation differs from what license plate parked at the garage, the user will be notified during the next login to the website and asked to register the vehicle with his or her account.	UC-8, UC-2
REQ20	Registered Customer can arrive early for a reservation, and be offered the option to park as a walk-in and have that parking carry over to their reservation.	UC-2

System Sequence Diagrams

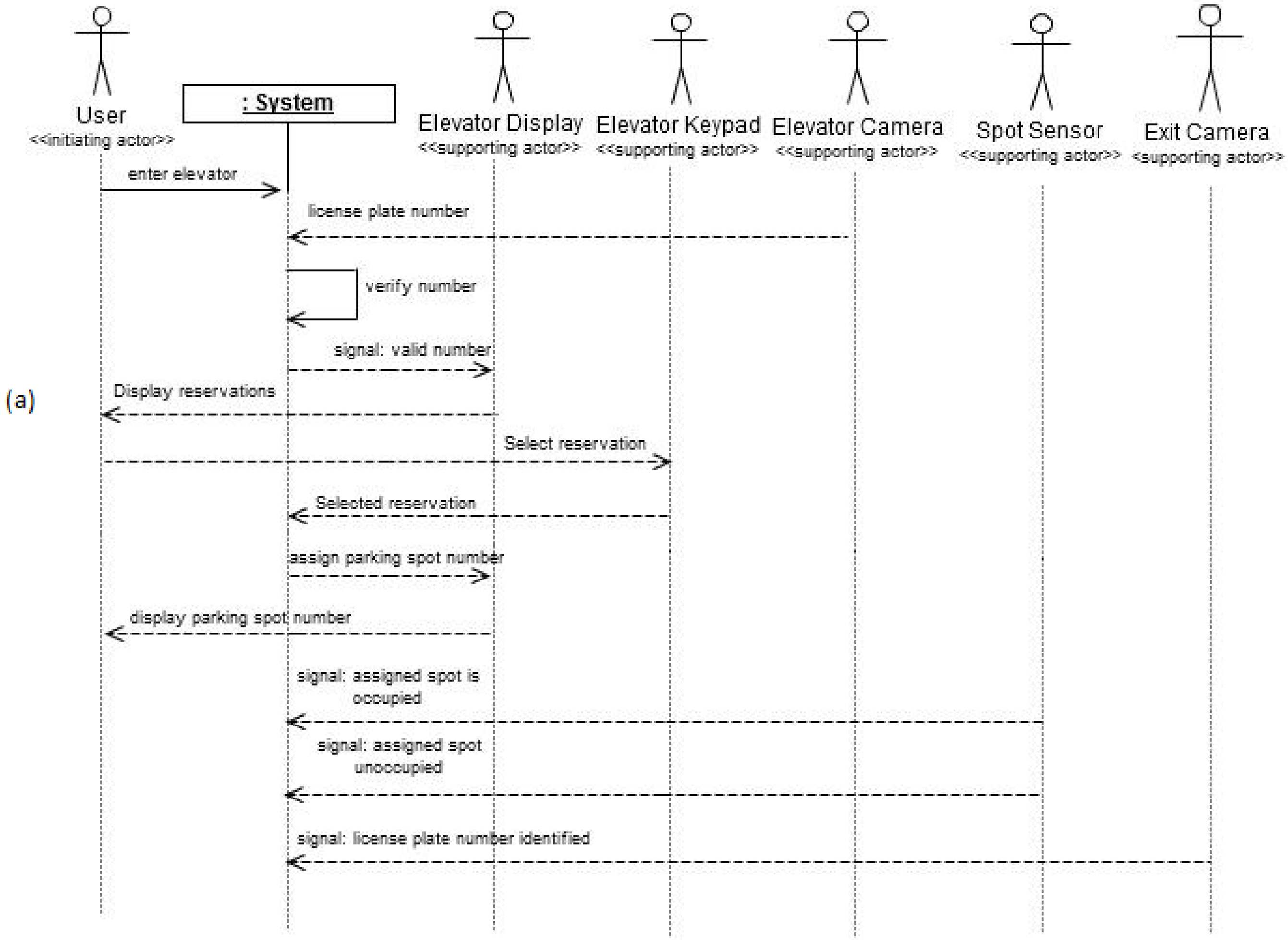
System Sequence Diagram UC - 1



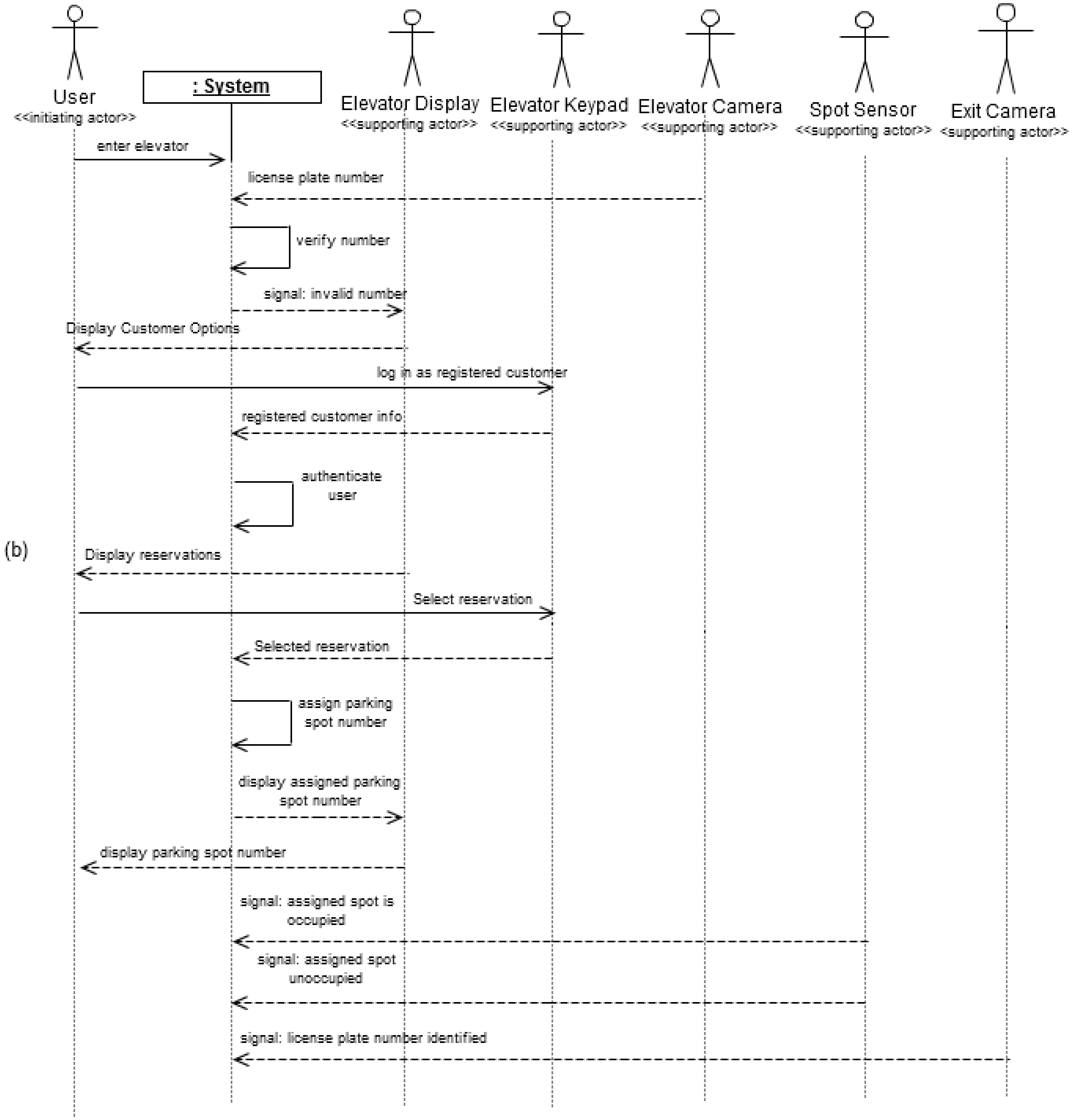
Registered Customer
<<intiating actor>>
: System
select function("make reservation")
prompt for reservation info
(a)
enter reservation info
verify reservation
signal: reservation made
select function("logout")
Registered Customer
<<initiating actor>>
: System
select function("make reservation")
loop
prompt for reservation info
enter reservation info
verify reservation
(b)
signal: no reservations available
prompt to enter new date and time
signal: reservation made
select function("logout")

The System sequence diagram for UC-1 can be split up into two parts. Part (a) describes the sequence of events for the success scenario. The user requests a reservation and there are available reservations to be given out. Part b describes the sequence of events for the alternate scenario. The user requests a reservation at a specific date and time but the there are no available reservations to be given out. The user will continue to enter in a new date and time until an available reservation can be given out.

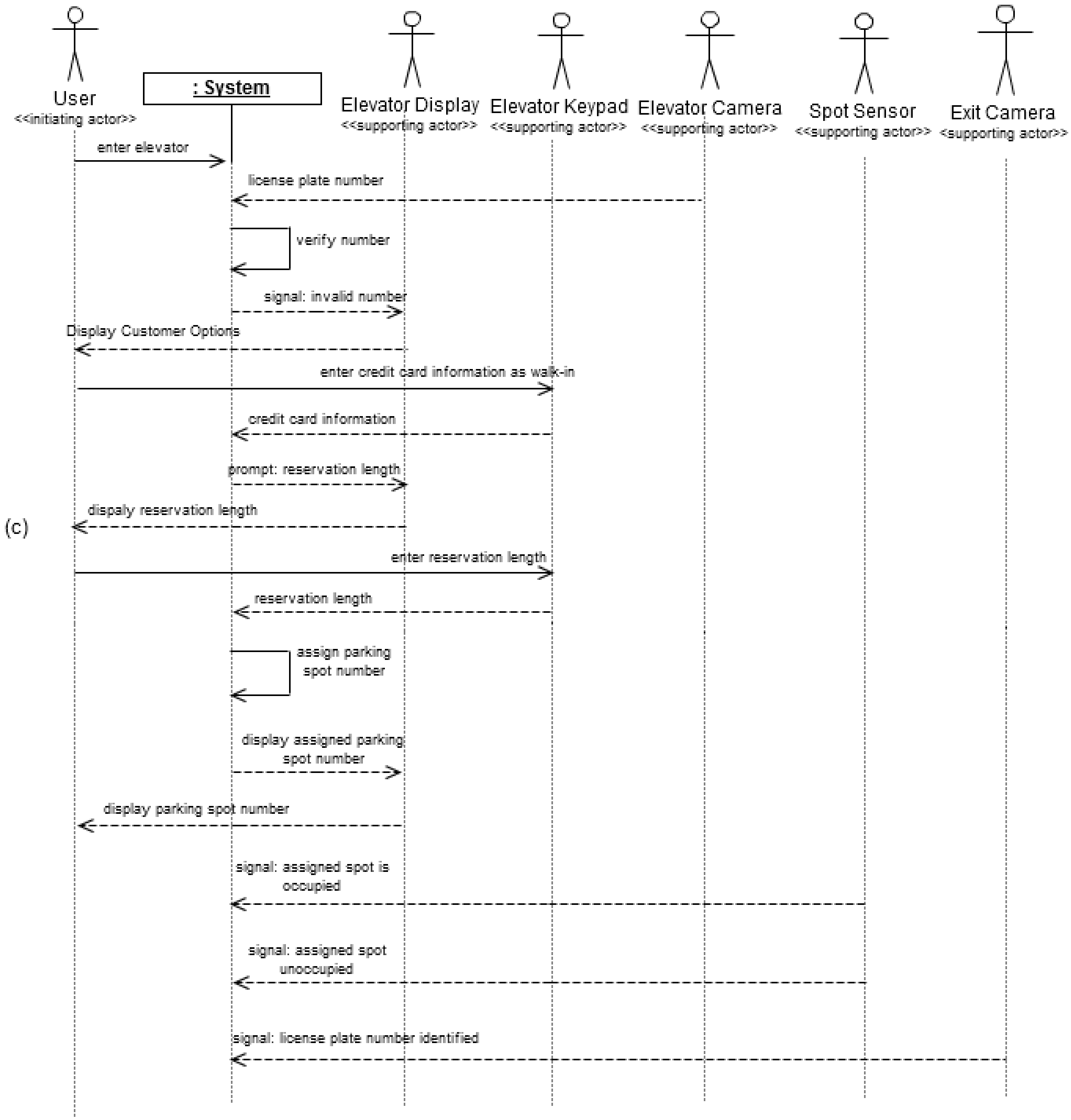
Sequence Diagram UC - 2



The system sequence diagram for UC-2 can be split into three parts. Part a describes the sequence of events for the success scenario. The customer enters the elevator, the elevator camera recognizes the the customer’s license plate number, and the system assigns the user a parking spot number.



Part (b) of the system sequence diagram for UC-2 describes the sequence of events for an alternate scenario. The customer enters the elevator but the elevator camera does not recognize the customer's license plate number. The customer then signs in as a registered customer and selects a reservation. The system then assigns the user a parking spot number.



Part (c) of the system sequence diagram for UC-2 describes the sequence of events for another alternate scenario. The customer enters the elevator but the elevator camera does not recognize the customer’s license plate number. The customer then swipes their credit card and selects a reservation as a walk-in customer. The system then assigns the user a parking spot number.

Non-Functional Requirements

Fault-tolerance

- Park-A-Lot should remember the details of a user's interaction if the user interface should disconnect from the system.
- Park-A-Lot should quickly recover from a malfunction when a customer is inside the elevator.

Usability

- The interface should provide customers with access to all relevant use cases with the fewest number of mouse clicks and key strokes.

Reliability

- Park-A-Lot should function correctly even if a customer inputs invalid entries into a reservation request form.
- Park-A-Lot should not lose a reservation through the use of persistent storage and regular backup.

Performance

- The Park-A-Lot elevator display should always display the correct output to the customer.
- The Park-A-Lot system should minimize connection times to the database and provide a quick and painless experience to the customer.
- Initially, Park-A-Lot can support at least 100 customers and 1,000 reservations. Over time should seek to increase these numbers ten-fold or more.

Security

- Other customers or unauthorized users should not have access to or be able to edit a customer's account details or reservations.

Effort Estimation Using Use Case Points

Unadjusted Actor Weight (UAW)

Actor Name	Description of relevant characteristics	Complexity	Weight
Registered Customer	Registered Customer is interacting with the system via the website or with the elevator keypad and display.	Complex	3
Unregistered Customer	Unregistered Customer is interacting with the system via the website (to create an account) or with the elevator keypad and display.	Complex	3
System Admin	The System Admin is interacting with the system via the website (to set prices and view access history).	Complex	3
Timer	Timer is another system which interacts with our system through a defined API.	Simple	1
Elevator Camera	Same as Timer.	Simple	1
Database	Database is another system interacting through a protocol.	Average	2
Spot Sensor	Same as Timer.	Simple	1
Elevator Keypad	Same as Timer.	Simple	1
Elevator Display	Same as Timer.	Simple	1
Exit Camera	Same as Timer.	Simple	1

UAW = 6 x Simple + 1 x Average + 3 x Complex = 6x1 + 1x2 + 3x3 = 17

Unadjusted Use Case Weight (UUCW)

Use Case	Description	Category	Weight
Reserve UC-1	Simple user interface. 3 steps for the main success scenario. 1 participating actor (Database).	Simple	5
Park UC-2	Complex user interface. More than 7 steps for all the scenarios. 6 participating actors (Elevator Display, Elevator Camera, Elevator Keypad, Spot Sensor, Exit Camera, Database).	Complex	10
Manage Account UC-3	Simple user interface. 4 steps for the main success scenario. 1 participating actor (Database).	Simple	5
View Reservations UC-4	Complex user interface. 4 steps for the main success scenario. 1 participating actor (Database).	Average	10
Register UC-5	Simple user interface. More than 7 steps for all the scenarios. 1 participating actor (Database).	Average	10
Manage Garage UC-6	Simple user interface. 6 steps required for the main success scenario. 1 participating actor (Database).	Average	10
Edit Reservation UC-7	Simple user interface. 4 steps required for the main success scenario. 1 participating actor (Database).	Simple	5
Register Vehicle UC-8	Simple user interface. 4 steps required for the main success scenario. 1 participating actor (Database).	Simple	5
Edit Vehicle UC-9	Simple user interface. 4 steps required for the main success scenario. 1 participating actor (Database).	Simple	5
Authenticate User UC-10	Simple user interface. 7 steps for all the scenarios. 1 participating actor (Database).	Average	10
Set Prices UC-11	Simple user interface. 8 steps for the main success scenario. 1 participating actor (Database).	Average	10

Inspect Usage History UC-12	Complex user interface. 6 steps for the main success scenario. 1 participating actor (Database).	Complex	15
Monthly Billing UC-13	Simple user interface. 4 steps for the main success scenario. 1 participating actor (Database).	Simple	5

UUCW = 6 x Simple + 5 x Average + 2 x Complex = 6x5 + 5x10 + 2x15 = 110

UUCP = UAW + UUCW = 17 + 110 = 127

Technical Complexity Factor (TCF)

Technical Factor	**Description**	**Weight**	**Perceived Complexity**	**Calculated Factor (Weight x Perceived Complexity)**
T1	Distributed, Web-based system.	2	5	2x5 = 10
T2	User expect good performance and no down times.	1	3	1x3 = 3
T3	End-users expect efficiency.	1	3	1x3 = 3
T4	Internal processing is relatively simple except reservation swapping (UC-1).	1	4	1x4 = 4
T5	Reusability is a must have feature.	1	4	1x4 = 4
T6	Ease of install is not important because its a web based system.	0.5	0	0.5x0 = 0
T7	Ease of use is very important.	0.5	3	0.5x3 = 1.5
T8	Portability could be important for future improvements (phone app).	2	0	2x0 = 0
T9	Ease to change is required.	1	3	1x3 = 3

T10	Concurrent use is required.	1	3	1x3 = 3
T11	Security is a significant concern.	1	4	1x4 = 4
T12	No direct access for third parties.	1	0	1x0 = 0
T13	No unique training needs.	1	0	1x0 = 0

TCF = Constant-1 + Constant-2 x Technical Factor Total =
0.6 + 0.01 x (10 + 3 + 3 + 4 + 4 + 1.5 + 3 + 3 + 4) = 0.955

Environmental Complexity Factor (ECF)

Environmental Factor	**Description**	**Weight**	**Perceived Impact**	**Calculated Factor (Weight x Perceived Impact)**
E1	Beginner familiarity with the UML- based development.	1.5	3	1.5 x 3 = 4.5
E2	Some familiarity with application problem.	0.5	3	0.5 x 3 = 1.5
E3	Some knowledge of object-oriented approach.	1	2	1 x 2 = 2
E4	Some knowledge of lead analyst.	0.5	2	0.5 x 2 = 1
E5	Highly motivated but lost one team member.	1	4	1 x 4 = 4
E6	Stable requirements expected.	2	1	2 x 1 = 2
E7	All staff is part-time (have other class work to do as well).	-1	4	-1 x 4 = -4
E8	Programming language of average difficulty will be used.	-1	3	-1 x 3 = -3

ECF = Constant-1 + Constant-2 x Environmental Factor Total =
1.4 - 0.03 x (4.5 + 1.5 + 2 + 1 + 4 + 2 - 4 - 3) = 1.16

Use Case Points (UCP)

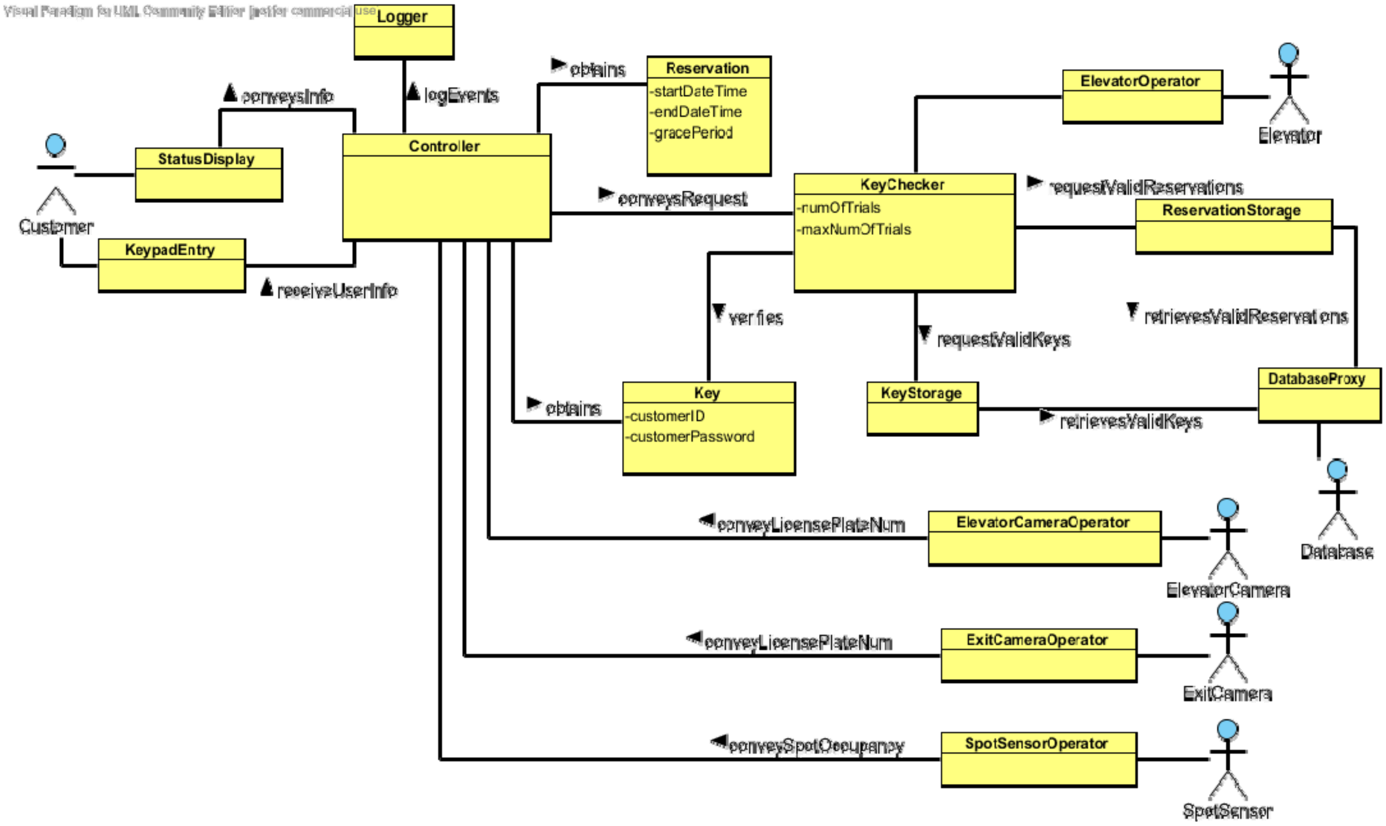
Therefore, the total of our use case points effort estimation will be:

UCP = UUCP x TCF x ECF = 127 x 0.955 x 1.16 = 140.7

Domain Analysis

Domain Models

Domain Model for UC-2



Reasons for model selection:

- **Cohesion -** The domain elements contain enough information to be completely independent objects (no overlap in knowledge between objects) and represent concrete ideas within the system. Therefore, our model has high cohesion since each object has several responsibilities, but does not attempt to do too much work.
- **Coupling -** The coupling of objects in our diagram is low, mainly because we have separated the key checking ability out from the Controller object. Key checking is a main concern in this model, and therefore deserves it's own object. The smaller objects surrounding KeyChecker help it accomplish its job. There is a high degree of coupling between Controller and many of the physical objects in the parking garage, but that is impossible to avoid since the controller needs to be in communication with all of the cameras and sensors in order to instruct other objects when to complete their tasks.
- **Expert Doer Principle -** The model satisfies this principle because it divides the task of checking keys and processing information about those checked keys into two distinct objects. The KeyChecker is the expert on checking customer authentication information,

and informs the Controller of customer authenticity. The Controller is then able to quickly display information about Reservations to the customer.

(D - doing; K - knowing; N - neither)

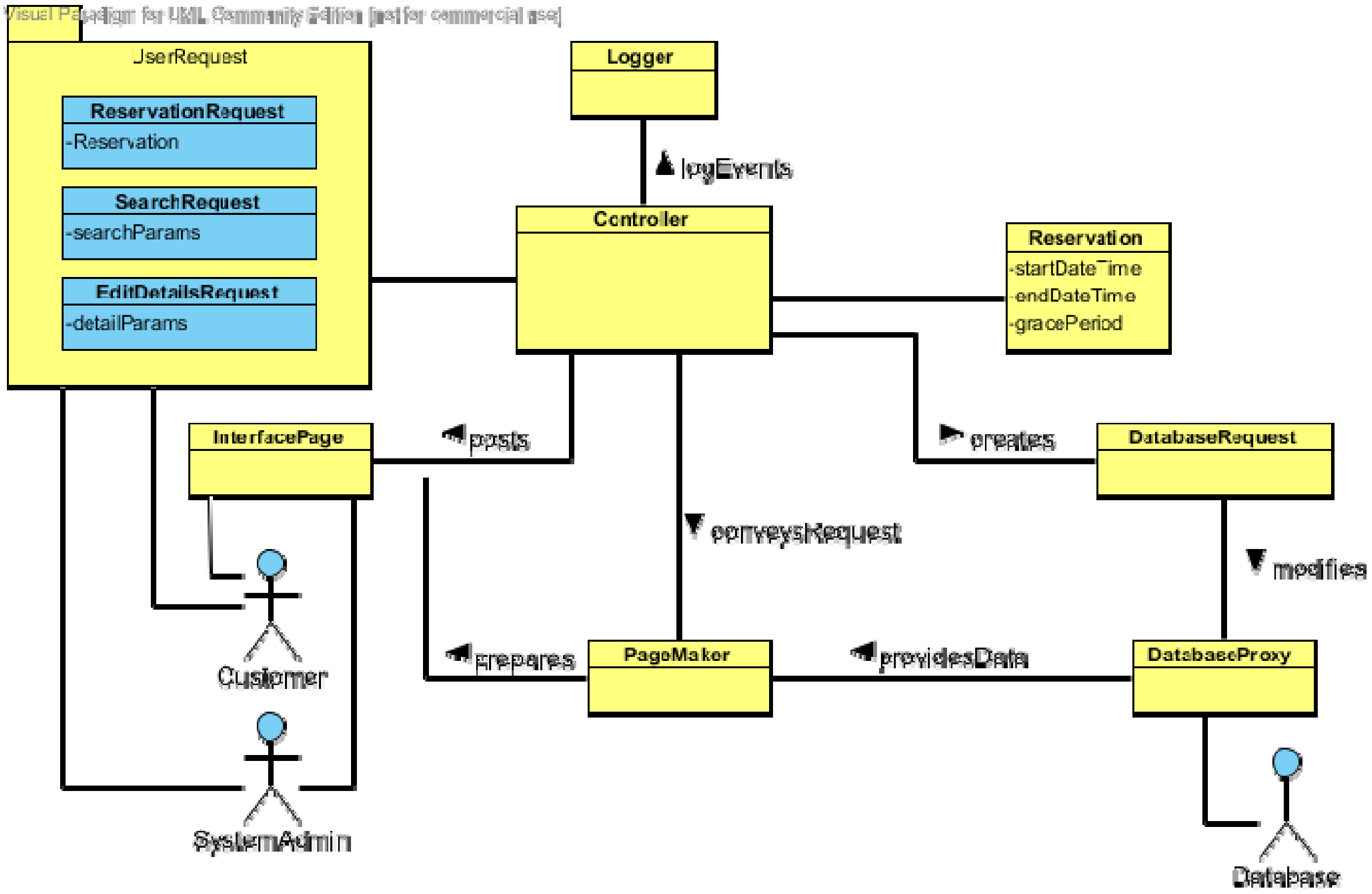
Responsibility Description	**Type**	**Concept Name**
Coordinate actions of concepts associated with this use case and delegate the work to other concepts.	D	Controller
Shows the actor the current context, what actions can be done, and the outcomes of the previous actions.	N	StatusDisplay
Container for the customer ID and password that the user entered in.	N	KeypadEntry
Container for user’s authentication data, including Customer ID and password.	K	Key
Verify whether or not the key-code entered by the user is valid.	D	KeyChecker
Container for the collection of valid keys associated with the users.	K	KeyStorage
Container for user’s existing reservations.	K	Reservation
Container for the collection of reservations associated with each user.	D	ReservationStorage
Operate the elevator camera to identify a car’s license plate number in the elevator platform.	D	ElevatorCameraOperator
Operate the elevator to move to the correct floor and open the entrance door.	D	ElevatorOperator
Operate the spot sensor to determine if a car is parked in the parking spot.	D	SpotSensorOperator
Operate the exit camera to identify the car’s license plate number that is exiting the garage.	D	ExitCameraOperator
Log all interactions with the system in persistent storage.	D	Logger

Concept Pair	**Association Description**	**Association Name**
Controller ↔ StatusDisplay	Controller passes information concerning the current context, what actions can be done, and the outcomes of the previous actions	conveysInfo
Controller ↔ KeypadEntry	Controller receives user input information from KeypadEntry.	receiveUserInfo
Controller ↔ Logger	Controller logs information to persistent storage about system interactions.	logEvents
Controller ↔ Reservation	Controller obtains reservation information from Reservation container.	obtains
Controller ↔ KeyChecker	Controller requests KeyChecker validate customer ID/password input and receives any reservations that ID is linked to.	conveysRequest
Controller ↔ Key	Controller obtains verified user information from Key container.	obtains
ElevatorCameraOperator ↔ Controller	ElevatorCameraOperator conveys available license plate number to Controller when ElevatorCamera detects a car in the Elevator.	conveyLicensePlateNum
ExitCamera Operator ↔ Controller	ExitCameraOperator conveys available license plate number to Controller when ExitCamera detects a car leaving the parking garage.	conveyLicensePlateNum
SpotSensor Operator ↔ Controller	SpotSensorOperator conveys true/false if a spot is filled to Controller.	conveySpotOccupancy
KeyChecker ↔ Key	KeyChecker verifies if Key matches given user account information.	verifies
KeyChecker ↔ KeyStorage	KeyChecker requests a list of valid keys from KeyStorage container.	requestValidKeys

KeyChecker ↔ Reservation Storage	KeyChecker retrieves a list if reservations from ReservationStorage container.	requstValid Reservations
KeyChecker ↔ Elevator Operator	KeyChecker tells the ElevatorOperating what floor to go to.	signalOperateElevator
Reservation Storage ↔ DatabaseProxy	ReservationStorage queries DatabaseProxy for valid reservations.	retrievesValid Reservations
KeyStorage ↔ DatabaseProxy	KeyStorage queries DatabaseProxy for valid keys.	retrievesValidKeys

Concept	**Attributes**	**Attribute Description**
Reservation	Start Date Time	Start time for the reservation of a particular actor.
	End Date Time	End time for the reservation of a particular actor.
	Grace Period	Grace period for the reservation of a particular actor.
Key	Customer’s ID	ID number of customer.
	Customer’s Password	Password for the customer.
Key Checker	Number of Trials	Counter to track how many times the user has unsuccessfully entered in their key.
	Max Number of Trials	Maximum allowable times a user can enter in their key unsuccessfully before they are asked to leave.

Domain Model for Remaining Use Cases



Reasons for model selection:

- **Cohesion -** The responsibilities assigned to each object in this model are not as great as they first appear, since the UserRequest object is actually a collection of three distinct objects which are used depending upon customer or system administrator input. The ReservationRequest is used in the case that the customer seeks to create a reservation, the SearchRequest in the case the customer or system administrator is searching for a set of reservations in the database, and the EditDetailsRequest in the case where a customer needs to edit some info in his or her account, or a system administrator needs to edit the details of an account. This gives each object a clearly defined set of responsibilities. The remaining objects are simply there to create and display the data fetched from the database, and each have a unique function, thus promoting high cohesion.
- **Coupling -** While there are many responsibilities assigned to each object, no object has an excessive amount of work to do. Each object has a distinct task to perform, and the Controller oversees these tasks and coordinates their efforts. The coupling is lower in this domain model than in the previous one, since we have reduced the interactions between Controller and all other objects, while still maintaining a distinct set of tasks for each object.

- **Expert Doer Principle -** As in the previous domain model, this model also conforms to the expert doer principle since all of its objects that know information are the objects performing the tasks. Controller passes any information obtained from the user input to a DatabaseRequest that performs actions through the DatabaseProxy on the database. Also, the PageMaker creates the page using information from the DatabaseProxy which gets any information needed from the database using the DatabaseRequest passed to it. The Controller orchestrates this entire exchange. Therefore, each object that knows the information passes it to the correct object to do something with it.

(D - doing; K - knowing; N - neither)

Responsibility Description	**Type**	**Concept Name**
Coordinate actions of concepts associated with this use case and delegate the work to other concepts.	D	Controller
Shows the actor the current context, what actions can be done, and outcomes of the previous actions.	K	InterfacePage
Container for three distinct request types: a reservation request containing reservation data, a search request containing search parameters, and an edit details request containing changes to customer account details.	K	UserRequest
Render retrieved records into an HTML document.	K	PageMaker
Container for user's requested reservations.	K	Reservation
Prepare a database query that best matches the actor's search criteria and retrieve the records from the database.	D	DatabaseProxy
Form specifying details that need to be changed in the database.	K	DatabaseRequest
Log all interactions with the system in persistent storage.	D	Logger

Concept Pair	**Association Description**	**Association Name**
Controller ↔ InterfacePage	Controller posts the InterfacePage with the help of PageMaker.	posts
Controller ↔ UserRequest	Controller obtains user information from UserRequest and forms a DatabaseRequest to act on that information.	receiveUserInfo
Controller ↔ Logger	Controller logs information to persistent storage about system interactions.	logEvents
Controller ↔ Reservation	Controller obtains reservation information from Reservation container.	obtains
Controller ↔ PageMaker	Controller conveys a request to PageMaker to prepare an HTML page that it will display to the customer through InterfacePage.	conveysRequest
Controller ↔ Database Request	Controller creates a DatabaseRequest from the UserRequest to specify which actions should be performed on the Database.	creates
PageMaker ↔ InterfacePage	Page Maker prepares the Interface Page.	prepares
DatabaseRequest ↔ DatabaseProxy	DatabaseRequest modifies the Database through DatabaseProxy to insert, delete, modify, or retrieve records in the Database.	modifies

Concept	**Attributes**	**Attribute Description**
Reservation	Start Date Time End Date Time Grace Period	Start time for the reservation of a particular actor. End time for the reservation of a particular actor. Grace period for the reservation of a particular actor.
Search Request	Search Parameters	Start Date, end date, customer ID.
Reservation Request	Reservation (s)	Date, start time, end time, type of reservation.
Edit Details Request	Detail Parameters	Customer ID, password, email address, name, address, zip code, state, phone number, credit card number, license plate number(s).

System Operation Contracts

Operation	Park
Preconditions	● parkingSpot = “unoccupied” , The parking spot will be occupied as soon as the customer exits the elevator.
Postconditions	● parkingSpot = “unoccupied”

Operation	Authenticate User
Preconditions	● Set of valid customer IDs is known to the system and is not empty ● numOfTrials < maxNumOfTrials ● numOfTrials = 0, for the first trial of the current user
Postconditions	● numOfTrials = 0, if the entered Customer ID contained in the set of valid keys.

Mathematical Models

There are no mathematical models utilized in this design.

Interaction Diagrams

Theoretical Interaction Diagrams

For UC-2, when the customer enters the parking garage, we have created a few interaction diagrams that would be useful in an actual commercial implementation but not for our demonstration purposes. We will not be demonstrating these interaction diagrams during the demo, simply because we cannot afford to build a garage with all of the required hardware to actually implement the signals our system would be receiving. However, we will include a way to simulate this in the coding of our final project.

The first diagram demonstrates the decisions to be made when a customer enters the garage elevator. If the customer is Registered and has an existing reservation, then they are called to UC-2 Registered Customer. If the customer is Registered but does not have an existing reservation then they are either called to UC-2 Registered Customer or UC-2 Walk In. If the customer is unregistered then they are called to UC-2 Walk In.

The interaction diagram entitled UC-2 Registered Customer walks through a Registered Customer logging into the keypad with their password and then selecting which reservation they wish to fulfill. If the customer does not have an existing reservation then they will create one if it is available. The customer will then be called to UC-2 Park.

The interaction diagram entitled UC-2 Walk In, walks through a Registered or Unregistered customer reserving a spot (if one is available) and then proceeding to UC-2 Park. This interaction diagram utilizes UC-1 Reserve.

The interaction Diagram entitled UC-2 Park, walks through the customer going to the correct floor, parking in the assigned parking spot and then leaving when the reservation is over. The interaction diagram utilized the Elevator Operator, the Spot Sensor Operator, the Elevator Camera Operator, and the Exit Camera Operator. These concepts are used to move the elevator to the correct floor, identify if a specific parking spot is full, recognize a car’s license plate number, and notify the system when the customer has left the garage. In our demonstration, we cannot utilize these concepts, so we have implemented them by using user interaction. The demo interaction diagrams will be explained in detail in the next section on implemented Interaction Diagrams.

Implemented Interaction Diagrams

All of the interaction diagrams demonstrate our use of the MVC (model, view, controller) framework. By using this type of framework, we can separate business logic from the controller and from the views. Within our architecture we have 5 different MVC groups listed below.

- User
- Reservation
- Vehicle
- Admin
- Timer

This means that we have 5 different controllers, 5 different models, and 5 different views, one corresponding to each group.

The following table lists the MVC group along with the Use Cases it is responsible for.

MVC	Use Cases
User	UC-3, UC-5, UC-10
Reservation	UC-4, UC-7
Vehicle	UC-8, UC-9
Admin	UC-6, UC-11, UC-12
Timer	UC-13

The MVC architecture is actually a very proven way of ensuring that we meet the high standards of the Expert Doer Principle, High Cohesion and Low Coupling. Since the business logic is extracted to the models, the customer-facing logic to the views, and the communication logic between models and views to controllers, we have assured that each interaction diagram that implements this architecture will benefit.

The following explains how each design principle is satisfied.

Expert Doer Principle

In an attempt to keep as much data encapsulated within each class as possible, our selection aligns well with the expert doer principle. The main acting classes are the Controllers, which delegate tasks to the Models and pass data to the Views to display. All of the validation and business logic is kept within the Models. This design principle ensures that the Models are the classes that know the information, since the Views are simply display elements (usually HTML), and the Controllers never know what information they are passing to the Models.

A common analogy might be that of a Cook-Server-Customer. The cook (model) prepares the food and knows what ingredients it contains; the server (controller) bring the food out to the customer; and the customer (views) consumes the food. The models know the information and manipulate it, which is exactly what the expert doer principle requires. Both the controller and views have no clue about the information, they only pass it and display it, but never operate on it.

Therefore, we have almost completely satisfied the expert doer principle.

High Cohesion Principle

The MVC architecture also has a very high level of cohesion, since all of the three elements work together but retain their own distinct logical functions.

Models *know* information and manipulate it at the will of the user and system. Controllers *pass* information between models and views and *control* its flow. Views *display* this information but do not know anything about it, or how to manipulate it.

Therefore, each element of the MVC architecture has it's own unique function. There is no overlapping functionality between any of the three core components here, because of the well-defined separation of business logic from display logic. As such, we have completely satisfied the high cohesion principle.

Low Coupling Principle

The architecture also supports excellent low coupling, since we do not need to have any two components of the system in constant communication. The view collects information, send it to the model with the help of the controller, and the model operates on that data and sends it back along to the view, using the controller. Never does one element operate on data then pass it along to be operated on again.

This abstraction of function with the MVC architecture ensures that our classes are loosely coupled, needing only to communicate with each other once or twice during an operation to accomplish it.

Notes and Conventions

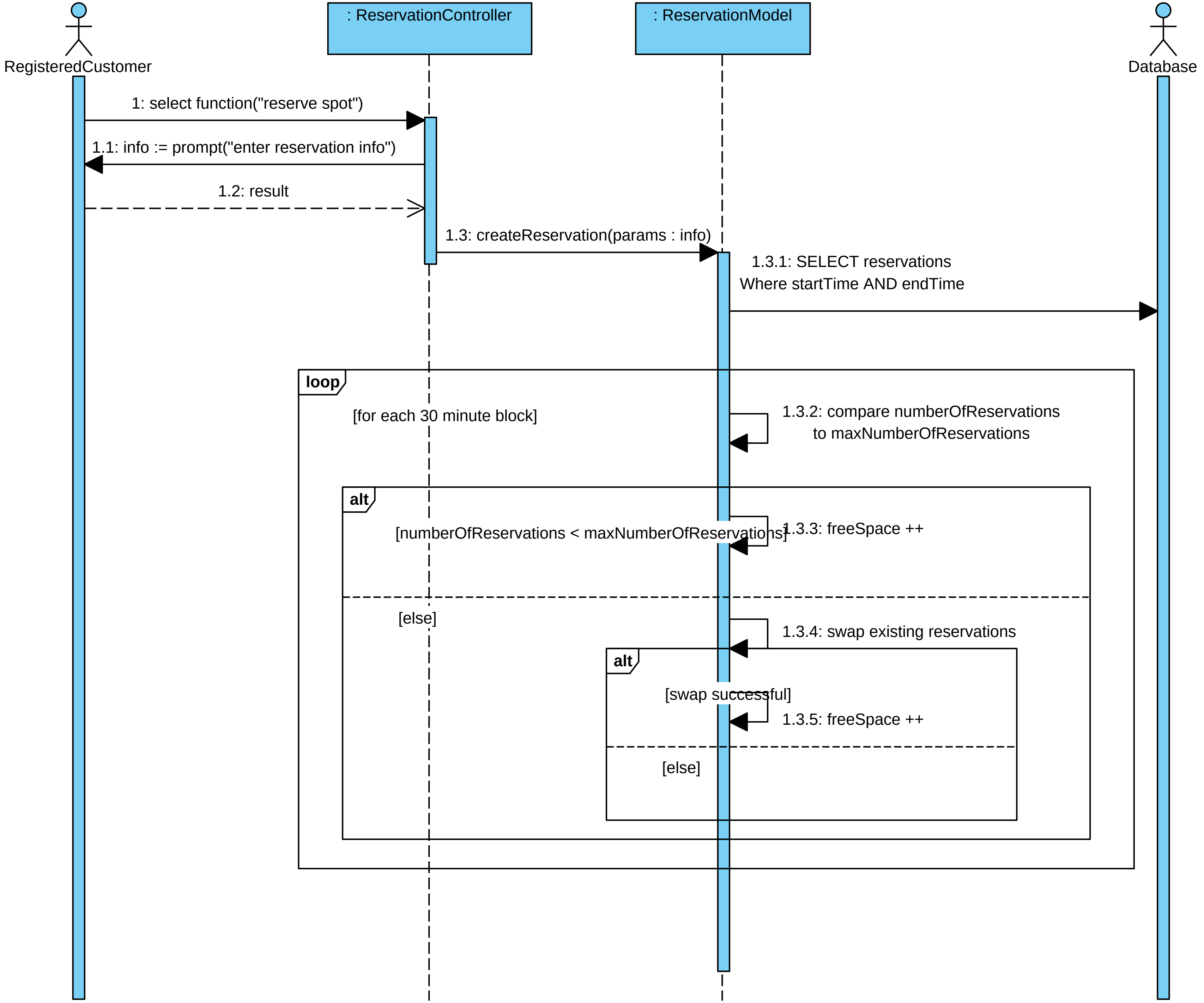
The following convention is used for database calls in the system interaction diagrams below. Although this is a not formal function definition, it will help the reader to understand what the query is trying to retrieve.

Function Call	Description
SQL Queries	Between Database Proxy and Database there are mock SQL queries, which although they are not complete, provide a rough idea of the query we would be using in our system.

UC-1: Reserve

Goals: To create a reservation and add it to the Database.

Process: The customer is prompted to input information about the reservation they wish to create, then the Controller passes the information to the Model, which determines if the reservation is valid and does not overbook the garage. The customer is then given either a success page or asked to re-enter the information with different values.



RegisteredCustomer
: ReservationController
: ReservationModel
Database
1: select function("reserve spot")
1.1: info := prompt("enter reservation info")
1.2: result
1.3: createReservation(params : info)
1.3.1: SELECT reservations
Where startTime AND endTime
loop
[for each 30 minute block]
1.3.2: compare numberOfReservations
to maxNumberOfReservations
alt
[numberOfReservations < maxNumberOfReservations]
1.3.3: freeSpace ++
[else]
1.3.4: swap existing reservations
alt
[swap successful]
1.3.5: freeSpace ++
[else]

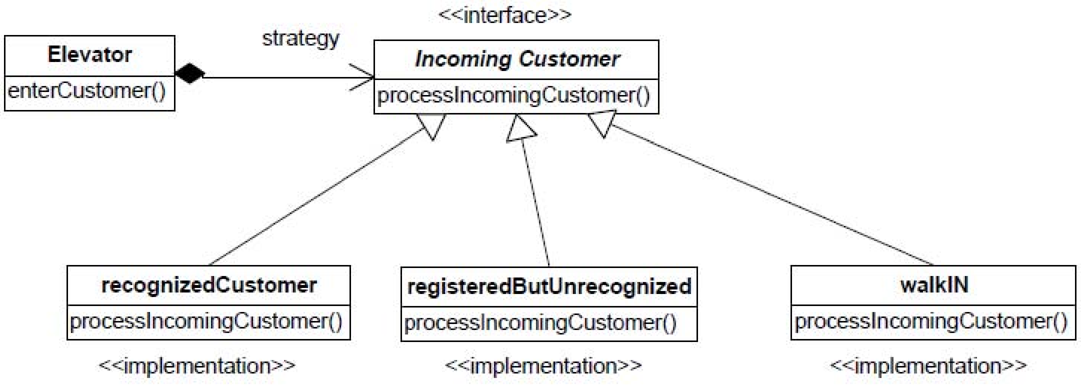
UC-2: Park

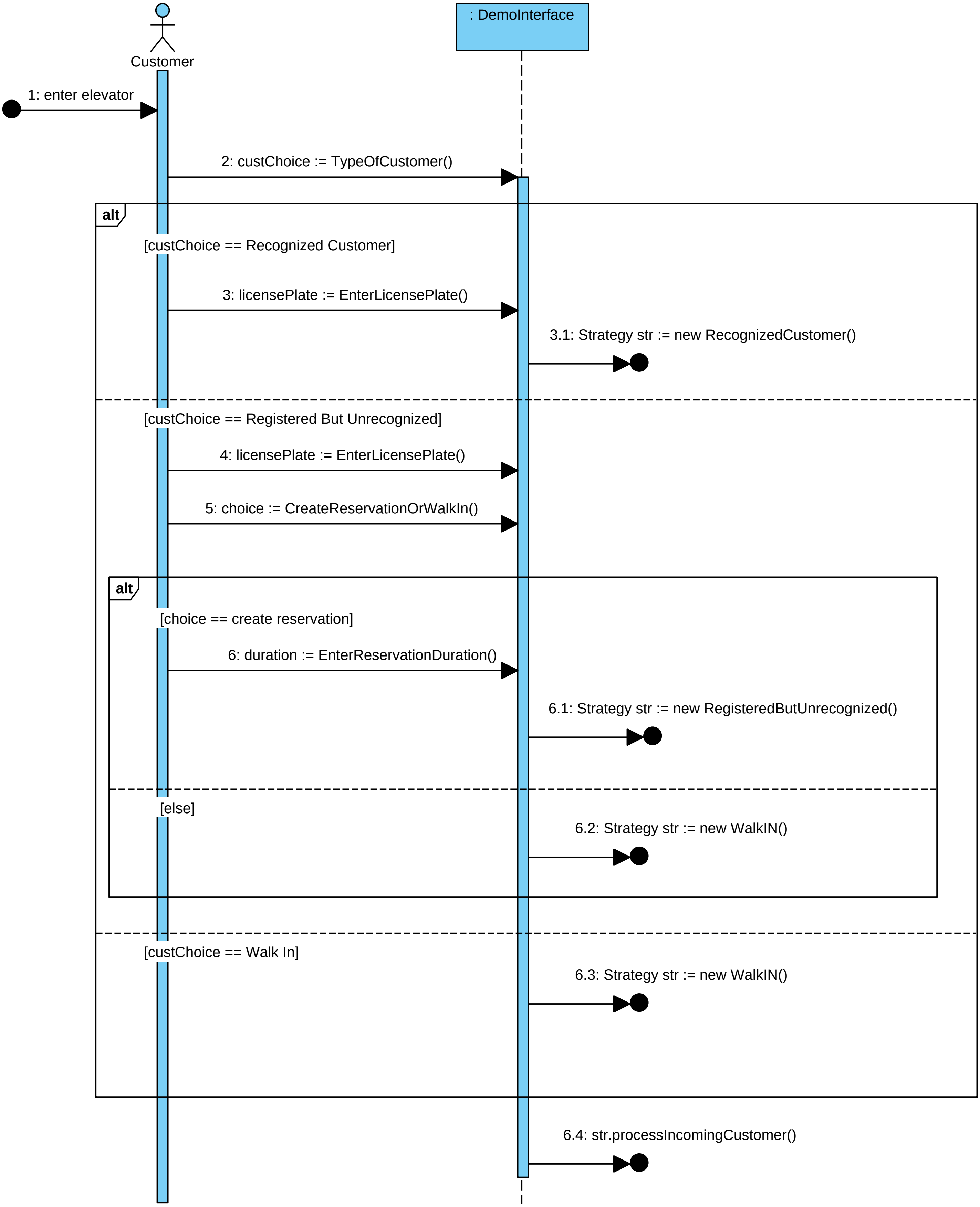
Goals: To park in the garage, either to fulfill a reservation or as a walk in.

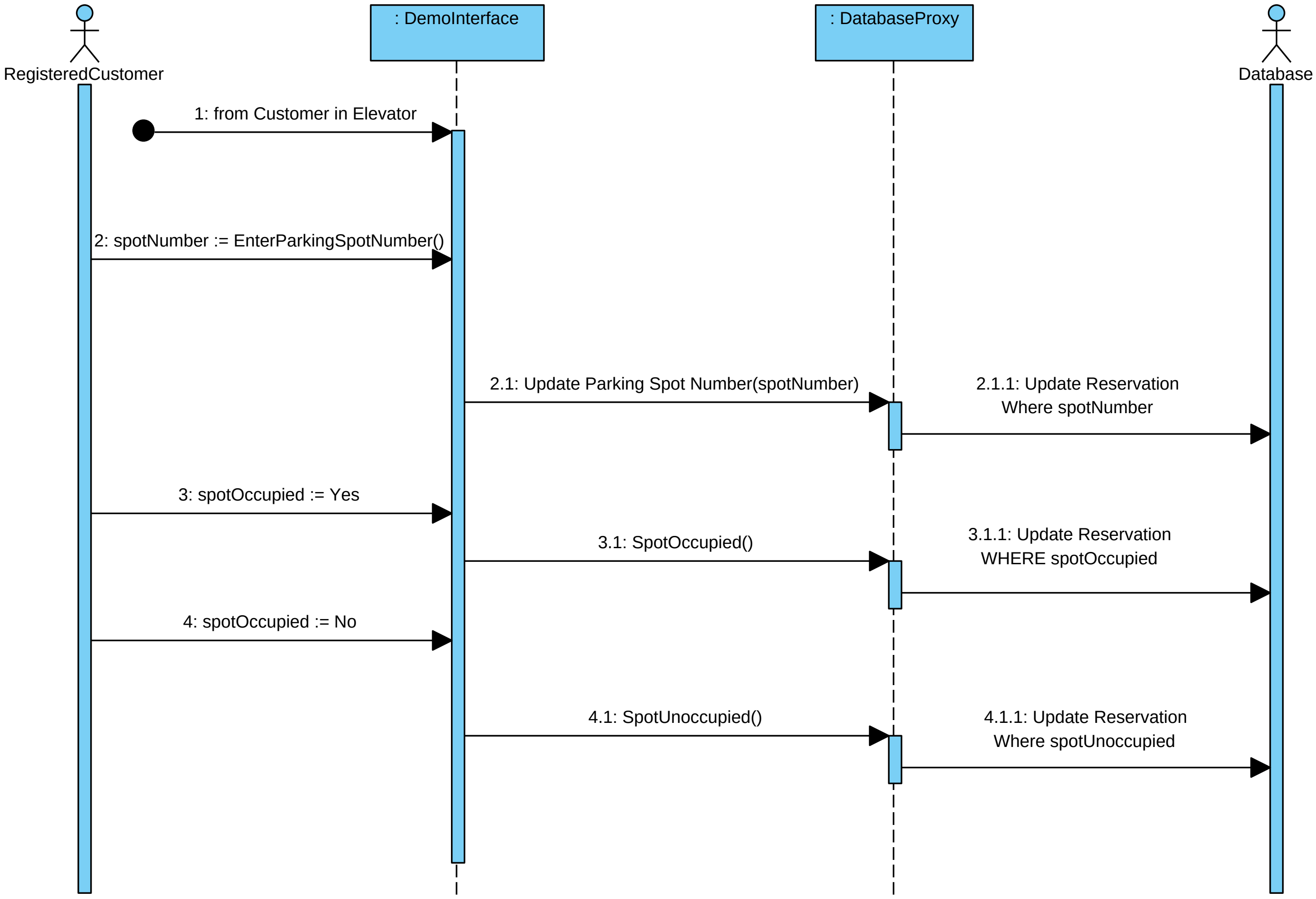
Process: A customer enters the garage and is prompted to select one of three options; Registered Customer with a reservation, Registered Customer without a reservation, or Walk In. Based on the selection the customer will be required to enter in his/her license plate number and then be told in what parking spot to park. The customer then has the option to leave the garage and is informed of how much the parked cost. This implementation is only for simulation purposes and is not intended to be used for an actual parking garage.

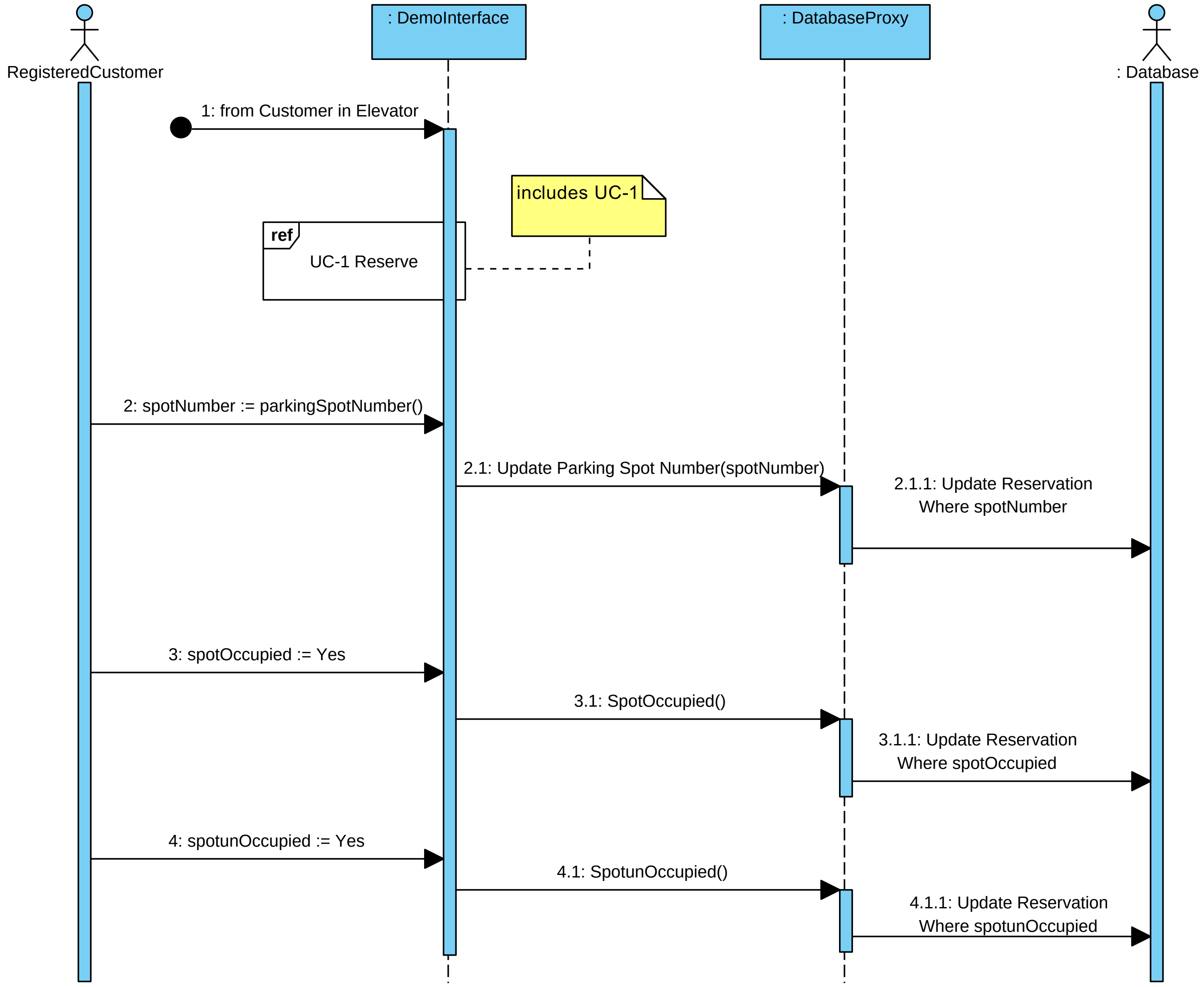
Design Pattern:
We have employed the strategy pattern in use case 2, Park. We decided to use this pattern because UC-2 has three different algorithms / options pertaining to the user's input and by employing this pattern we can simplify the logic involved. All three of the algorithms / options are chosen at run time so this patter was an obvious choice. The three algorithms we have employed are Recognized Customer, Registered But Unrecognized, and Walk In. All three strategies are inherited by the incoming customer interface. The diagram below shows the strategies and their inherited methods.

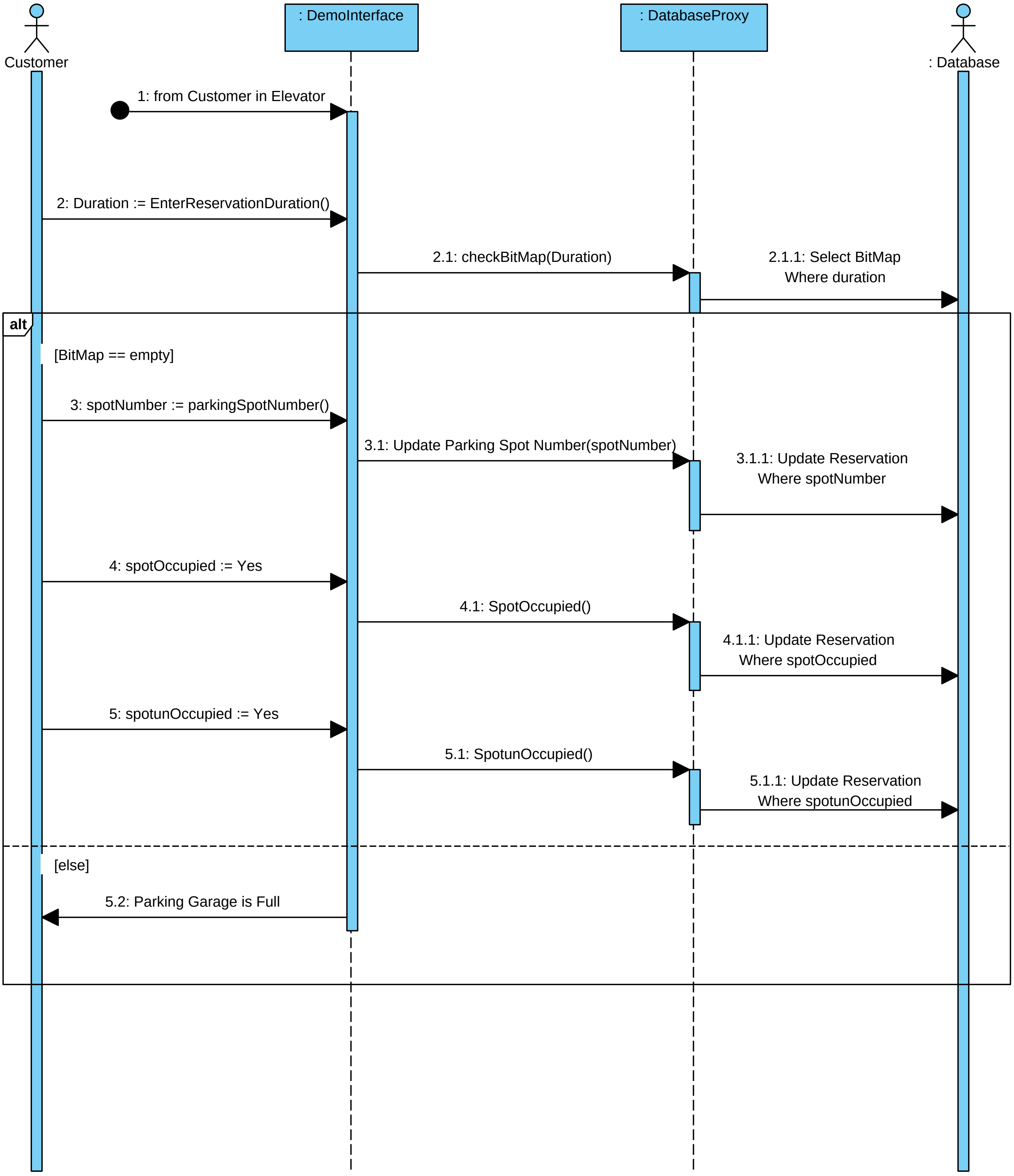
The advantaged to using the strategy pattern are that the strategies are encapsulated as an object and then made interchangeable. It would be easy in the future to add, remove or change any of the strategies because they are each a separate object called from their parent interface. Also, since all of the information relating to picking which strategy to use is entered during run time, the strategy patterns limits the amount of hard coding that is necessary to utilize each option. Each strategy is stored as a reference to the actual object which makes it easy to create and destroy.




Customer
: DemoInterface
1: enter elevator
2: custChoice := TypeOfCustomer()
alt
[custChoice == Recognized Customer]
3: licensePlate := EnterLicensePlate()
3.1: Strategy str := new RecognizedCustomer()
[custChoice == Registered But Unrecognized]
4: licensePlate := EnterLicensePlate()
5: choice := CreateReservationOrWalkIn()
alt
[choice == create reservation]
6: duration := EnterReservationDuration()
6.1: Strategy str := new RegisteredButUnrecognized()
[else]
6.2: Strategy str := new WalkIN()
[custChoice == Walk In]
6.3: Strategy str := new WalkIN()
6.4: str.processIncomingCustomer()


RegisteredCustomer
: DemoInterface
: DatabaseProxy
Database
1: from Customer in Elevator
2: spotNumber := EnterParkingSpotNumber()
2.1: Update Parking Spot Number(spotNumber)
2.1.1: Update Reservation
Where spotNumber
3: spotOccupied := Yes
3.1: SpotOccupied()
3.1.1: Update Reservation
WHERE spotOccupied
4: spotOccupied := No
4.1: SpotUnoccupied()
4.1.1: Update Reservation
Where spotUnoccupied

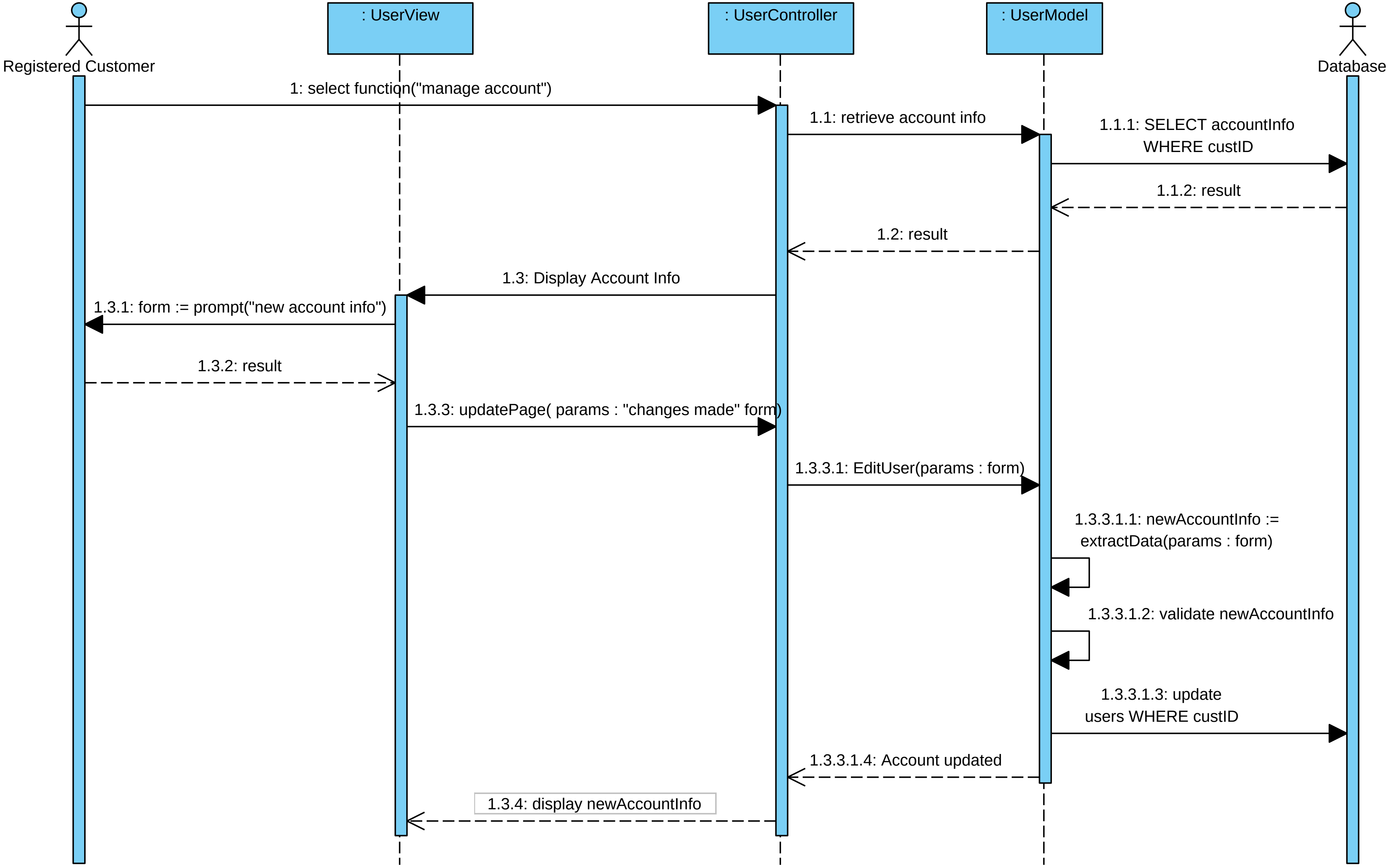

RegisteredCustomer
: DemoInterface
: DatabaseProxy
: Database
1: from Customer in Elevator
includes UC-1
ref
UC-1 Reserve
2: spotNumber := parkingSpotNumber()
2.1: Update Parking Spot Number(spotNumber)
2.1.1: Update Reservation
Where spotNumber
3: spotOccupied := Yes
3.1: SpotOccupied()
3.1.1: Update Reservation
Where spotOccupied
4: spotunOccupied := Yes
4.1: SpotunOccupied()
4.1.1: Update Reservation
Where spotunOccupied


Customer
: DemoInterface
: DatabaseProxy
: Database
1: from Customer in Elevator
2: Duration := EnterReservationDuration()
2.1: checkBitMap(Duration)
2.1.1: Select BitMap
Where duration
alt
[BitMap == empty]
3: spotNumber := parkingSpotNumber()
3.1: Update Parking Spot Number(spotNumber)
3.1.1: Update Reservation
Where spotNumber
4: spotOccupied := Yes
4.1: SpotOccupied()
4.1.1: Update Reservation
Where spotOccupied
5: spotunOccupied := Yes
5.1: SpotunOccupied()
5.1.1: Update Reservation
Where spotunOccupied
[else]
5.2: Parking Garage is Full

UC-3: Manage Account

Goals: To change account details for a registered customer.

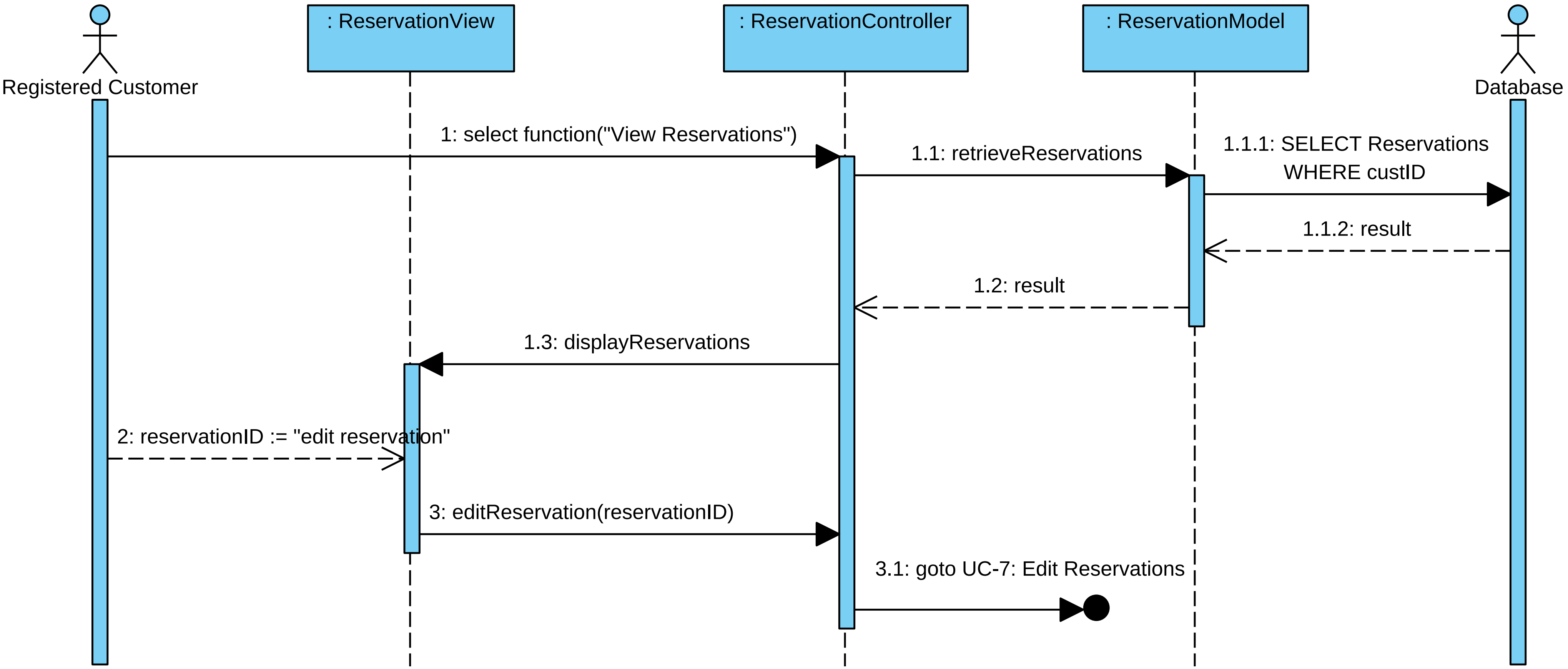
Process: The registered customer is prompted to make changes to their current account information. The controller passes the information to the Model, which updates the account in the database and then displays a page signifying the successful update.


Registered Customer
: UserView
: UserController
: UserModel
Database
1: select function("manage account")
1.1: retrieve account info
1.1.1: SELECT accountInfo WHERE custID
1.1.2: result
1.2: result
1.3: Display Account Info
1.3.1: form := prompt("new account info")
1.3.2: result
1.3.3: updatePage(params : "changes made" form)
1.3.3.1: EditUser(params : form)
1.3.3.1.1: newAccountInfo := extractData(params : form)
1.3.3.1.2: validate newAccountInfo
1.3.3.1.3: update users WHERE custID
1.3.3.1.4: Account updated
1.3.4: display newAccountInfo

UC-4: View Reservations

Goals: To view existing reservations for a registered customer.

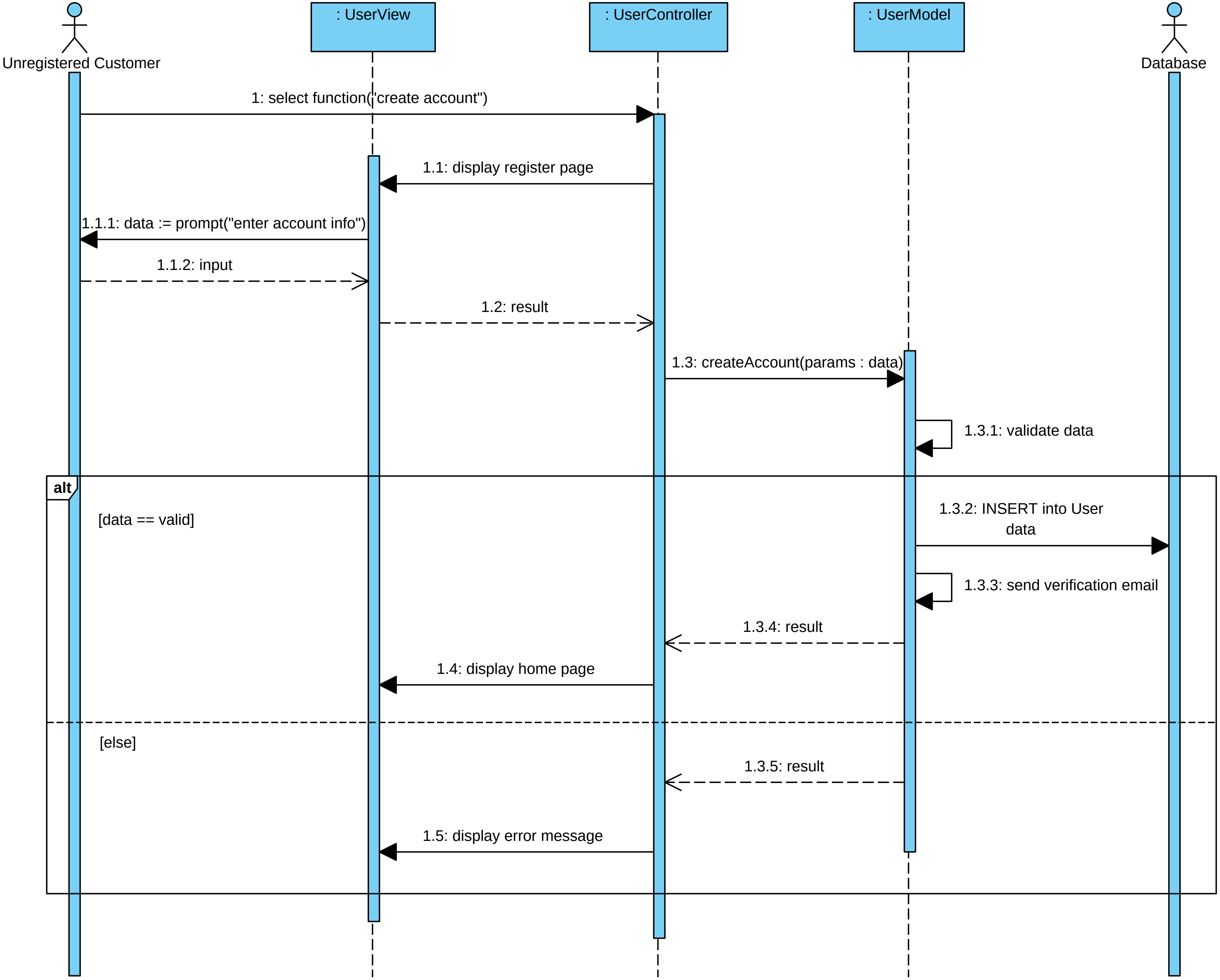
Process: The registered customer selects the option to view his or her existing reservations, both past, present, and future. The Controller receives this choice and displays that customer's reservations.


Registered Customer
: ReservationView
: ReservationController
: ReservationModel
Database
1: select function("View Reservations")
1.1: retrieveReservations
1.1.1: SELECT Reservations WHERE custID
1.1.2: result
1.2: result
1.3: displayReservations
2: reservationID := "edit reservation"
3: editReservation(reservationID)
3.1: goto UC-7: Edit Reservations

UC-5: Register

Goals: To become a registered customer and be stored in the database.

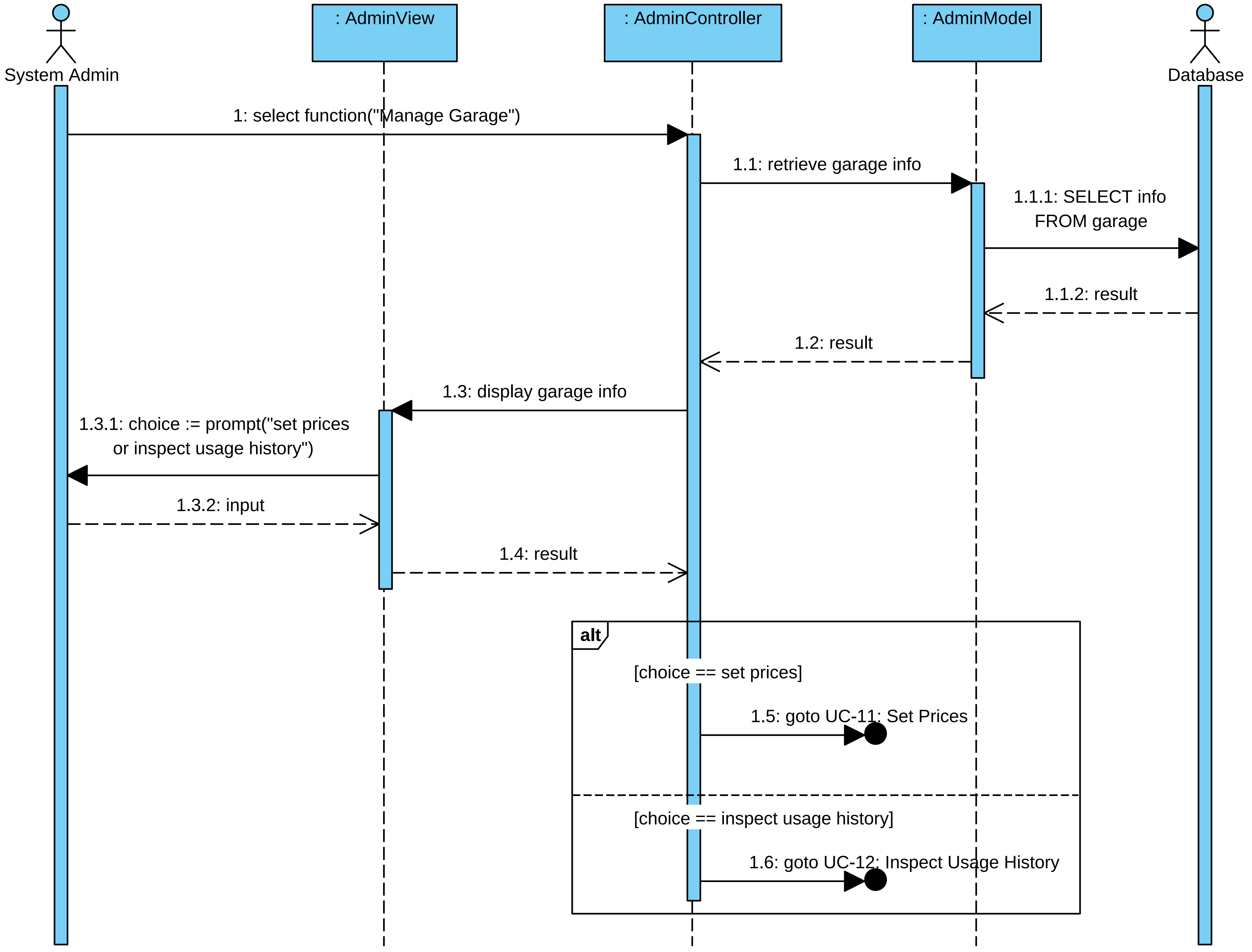
Process: The registered customer is prompted to input their account information. The Controller passes the account information to the Model which creates the account in the database. The Controller then calls the View to display a page signifying the success.


Unregistered Customer
: UserView
: UserController
: UserModel
Database
1: select function("create account")
1.1: display register page
1.1.1: data := prompt("enter account info")
1.1.2: input
1.2: result
1.3: createAccount(params : data)
1.3.1: validate data
alt
[data == valid]
1.3.2: INSERT into User data
1.3.3: send verification email
1.3.4: result
1.4: display home page
[else]
1.3.5: result
1.5: display error message

UC-6: Manage Garage

Goals: For a system admin to either set prices or view access history of a garage (these are both sub-use cases)..

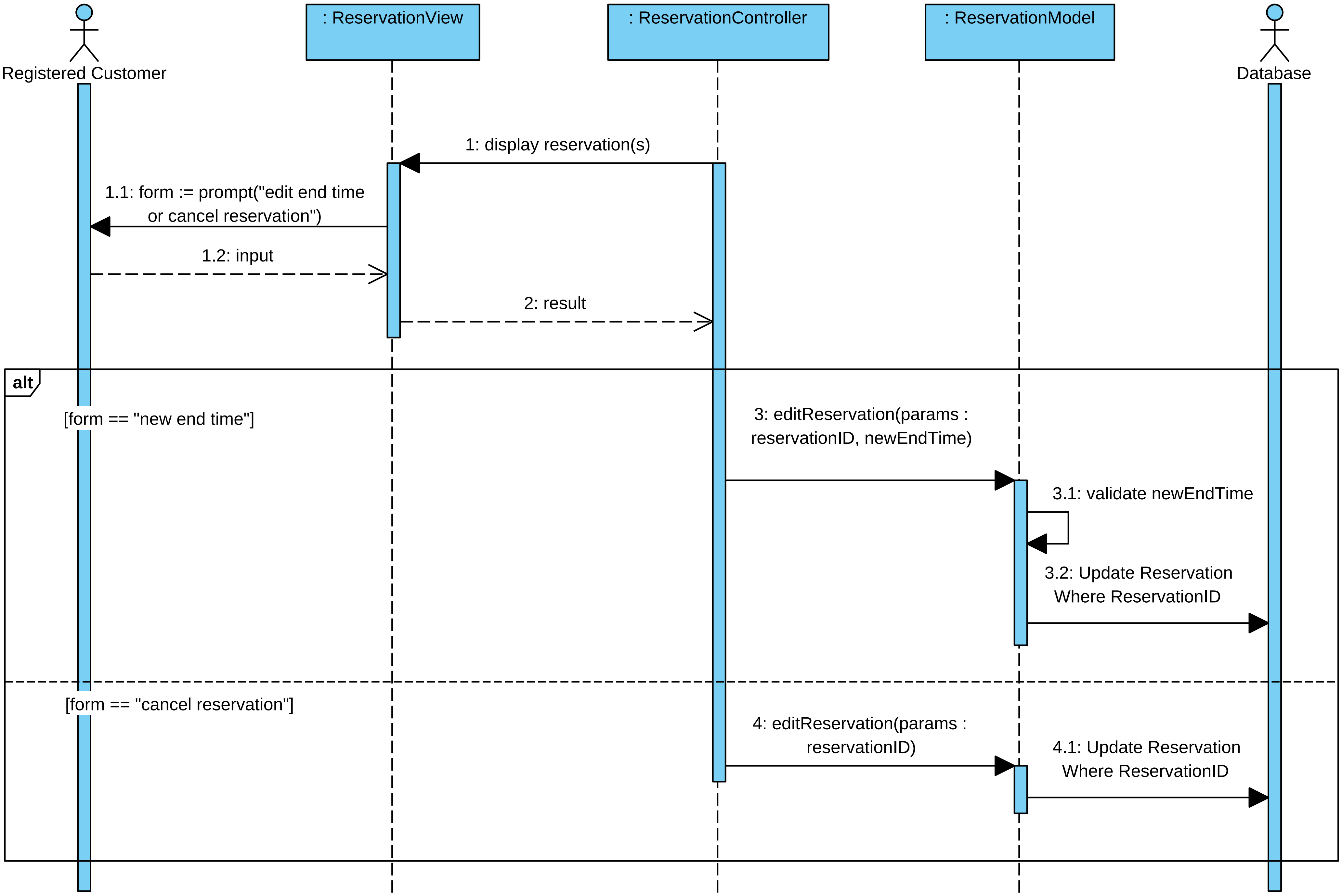
Process: The system admin is prompted to select whether to set prices or view access history for a garage. The Controller receives the choice and then passes it to to either system interaction diagram 11 or 12.


System Admin
: AdminView
: AdminController
: AdminModel
Database
1: select function("Manage Garage")
1.1: retrieve garage info
1.1.1: SELECT info FROM garage
1.1.2: result
1.2: result
1.3: display garage info
1.3.1: choice := prompt("set prices or inspect usage history")
1.3.2: input
1.4: result
alt
[choice == set prices]
1.5: goto UC-11: Set Prices
[choice == inspect usage history]
1.6: goto UC-12: Inspect Usage History

UC-7: Edit Reservation

Goals: To edit or cancel an existing reservation for a registered customer.

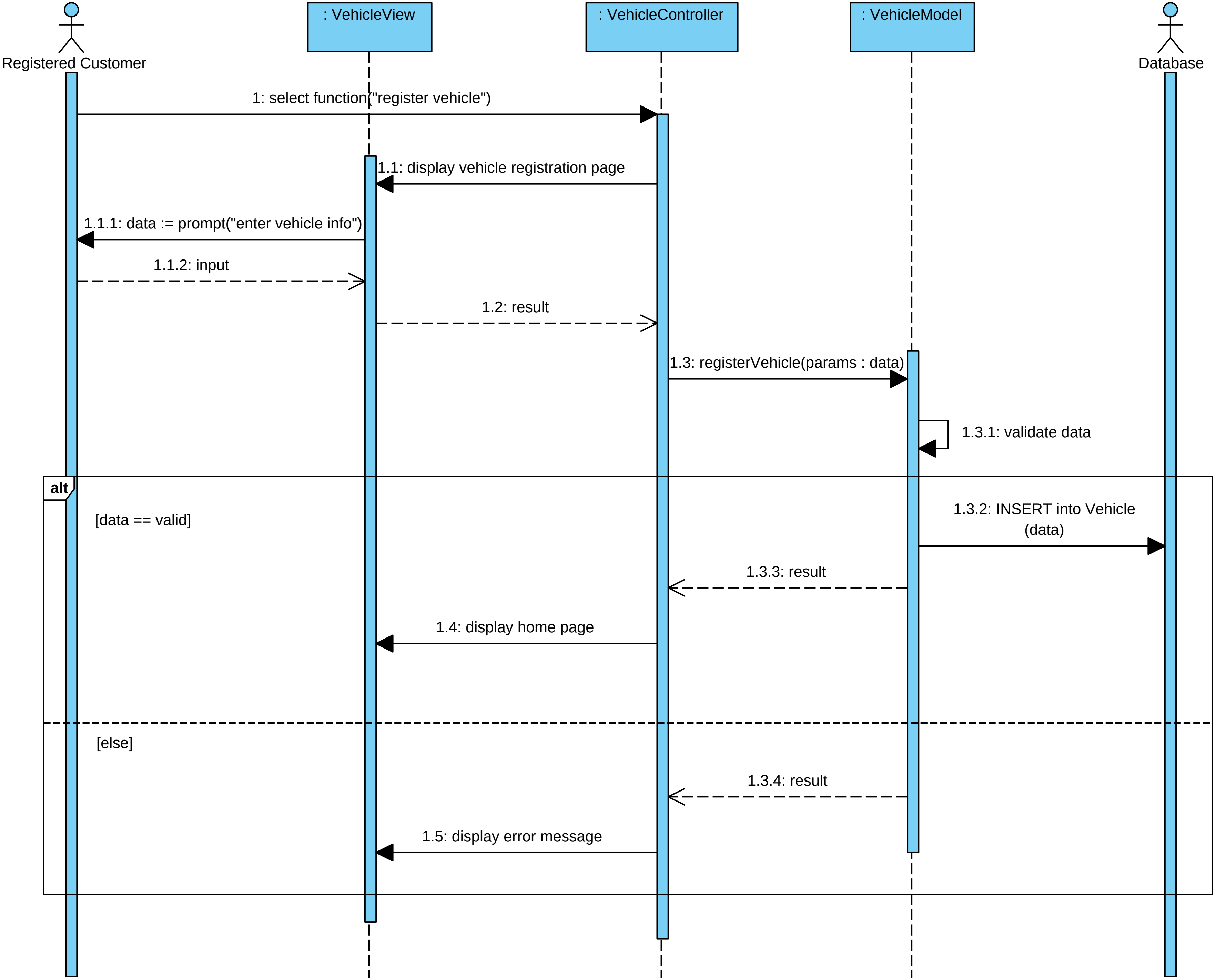
Process: The Registered customer either selects an change in the end time of the reservation or selects the cancel reservation option. The Controller receives this information and then passes it to the Model to validate and make the changes in the database. The View displays the successful change.


Registered Customer
: ReservationView
: ReservationController
: ReservationModel
Database
1: display reservation(s)
1.1: form := prompt("edit end time or cancel reservation")
1.2: input
2: result
alt
[form == "new end time"]
3: editReservation(params : reservationID, newEndTime)
3.1: validate newEndTime
3.2: Update Reservation Where ReservationID
[form == "cancel reservation"]
4: editReservation(params : reservationID)
4.1: Update Reservation Where ReservationID

UC-8: Register Vehicle

Goals: To create a vehicle for a Registered Customer’s account.

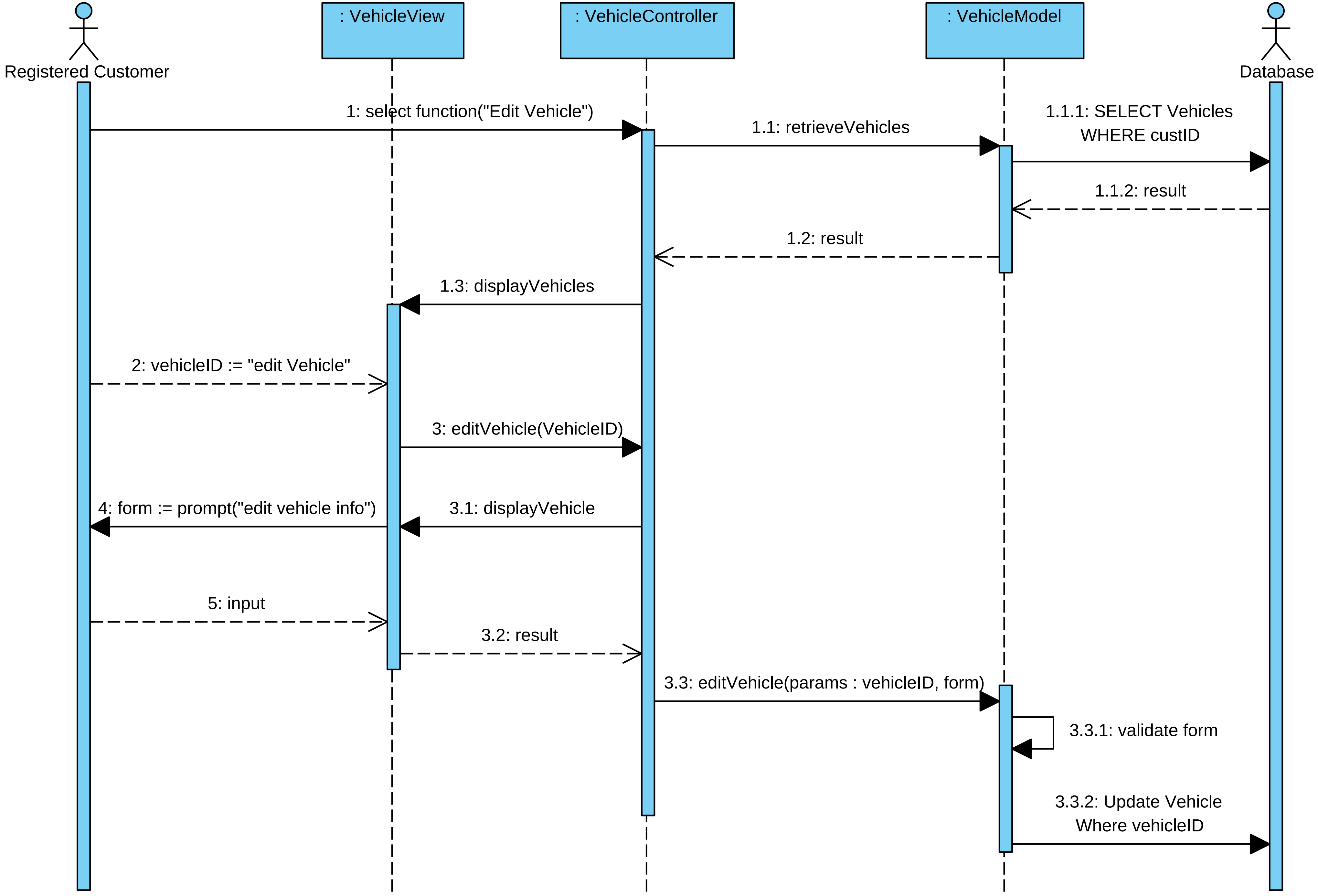
Process: The Registered Customer enters the vehicle’s license plate number and state. The Controller receives this information and then passes it to the Model which validates it and then makes the addition to the database. The View displays the successful creation of the vehicle.


Registered Customer
: VehicleView
: VehicleController
: VehicleModel
Database
1: select function("register vehicle")
1.1: display vehicle registration page
1.1.1: data := prompt("enter vehicle info")
1.1.2: input
1.2: result
1.3: registerVehicle(params : data)
1.3.1: validate data
alt
[data == valid]
1.3.2: INSERT into Vehicle (data)
1.3.3: result
1.4: display home page
[else]
1.3.4: result
1.5: display error message

UC-9: Edit Vehicle

Goals: To edit a registered vehicle's license plate number or state..

Process: The Registered Customer enters the vehicle's new license plate number and state. The Controller receives this information and then passes it to the Model which validates it and then makes the addition to the database. The View displays the successful change to the vehicle.


Registered Customer
: VehicleView
: VehicleController
: VehicleModel
Database
1: select function("Edit Vehicle")
1.1: retrieveVehicles
1.1.1: SELECT Vehicles WHERE custID
1.1.2: result
1.2: result
1.3: displayVehicles
2: vehicleID := "edit Vehicle"
3: editVehicle(VehicleID)
4: form := prompt("edit vehicle info")
3.1: displayVehicle
5: input
3.2: result
3.3: editVehicle(params : vehicleID, form)
3.3.1: validate form
3.3.2: Update Vehicle Where vehicleID

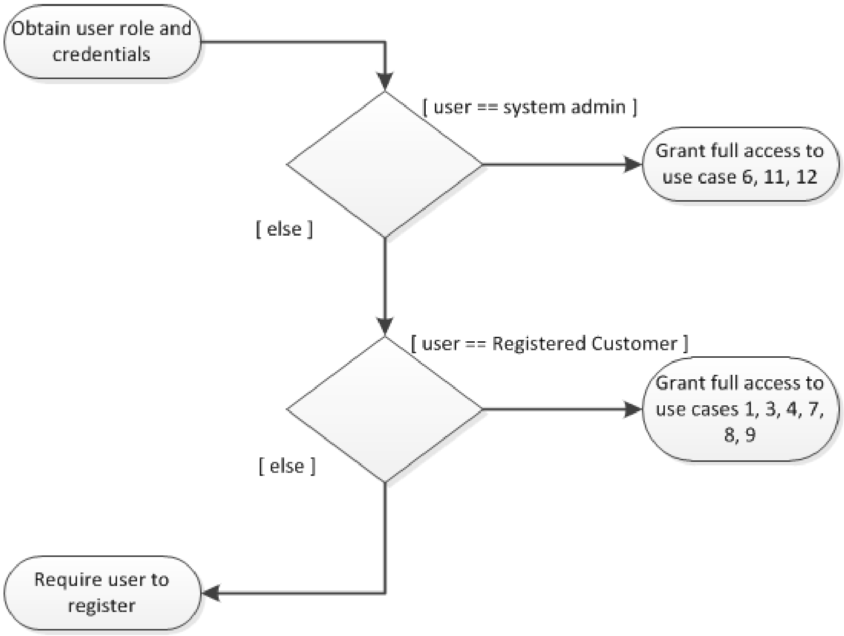
UC-10: Authenticate User

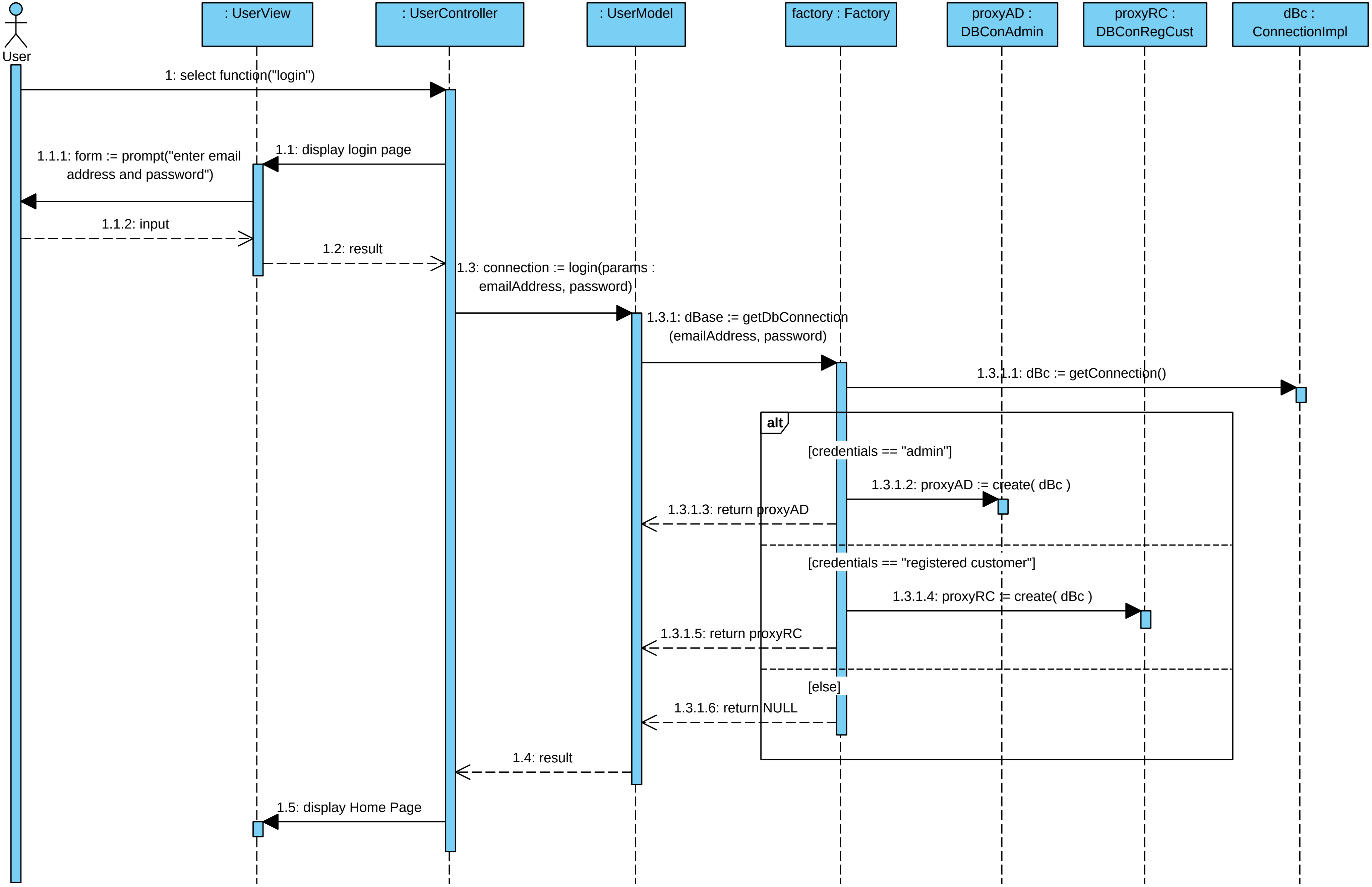
Goals: To determine if a user is registered with the system and has an account in the database.

Process: The user accesses the Interface Page via a web browser, selects log in, and enters their credentials (email and password). The Controller passes the information to the Model which creates a protection proxy based on the user's privileges.

Design Pattern:
We have employed the protection proxy pattern in use case 10, Authenticate User. We decided to use this pattern because different users with different roles log into the system and it would be helpful to assign each user their privileges through a protection proxy. The two roles for the protection proxy are system admin and Registered Customer. The diagram below shows which use cases the two different roles have a privilege to participate in.

The advantages to using the protection proxy are that we are able to take the logic for granting privileges away from the models and into a proxy. By doing this, it will be easy in the future to add additional roles and privileges. If we were to keep the system the way it was, then it would be a daunting task to make any changes because all of the complex logic and IF-THEN-ELSE statements involved. Also, the roles and privileges serve as a distraction from the main task of the client and server objects, so adding a protection proxy removes side responsibilities away from the objects and into their own proxy. The protection proxy is also able to protect an object from unauthorized access better than in out previous system because of its simplicity.

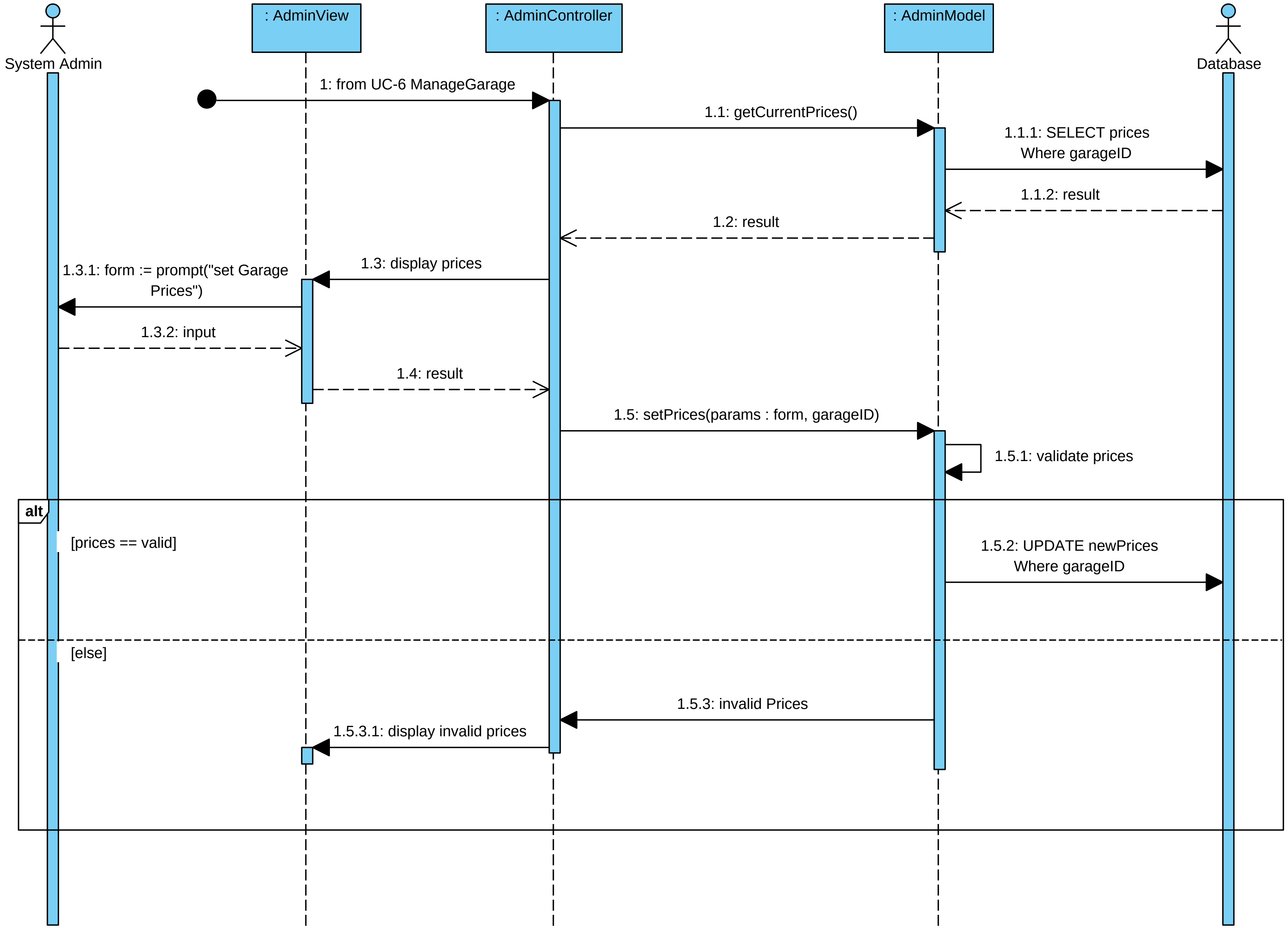

Obtain user role and credentials
[user == system admin]
Grant full access to use case 6, 11, 12
[else]
[user == Registered Customer]
Grant full access to use cases 1, 3, 4, 7, 8, 9
[else]
Require user to register


User
: UserView
: UserController
: UserModel
factory : Factory
proxyAD : DBConAdmin
proxyRC : DBConRegCust
dBc : ConnectionImpl
1: select function("login")
1.1: display login page
1.1.1: form := prompt("enter email address and password")
1.1.2: input
1.2: result
1.3: connection := login(params : emailAddress, password)
1.3.1: dBase := getDbConnection (emailAddress, password)
1.3.1.1: dBc := getConnection()
alt
[credentials == "admin"]
1.3.1.2: proxyAD := create(dBc)
1.3.1.3: return proxyAD
[credentials == "registered customer"]
1.3.1.4: proxyRC := create(dBc)
1.3.1.5: return proxyRC
[else]
1.3.1.6: return NULL
1.4: result
1.5: display Home Page

UC-11: Set Prices

Goal: To update the prices for parking at the garage.

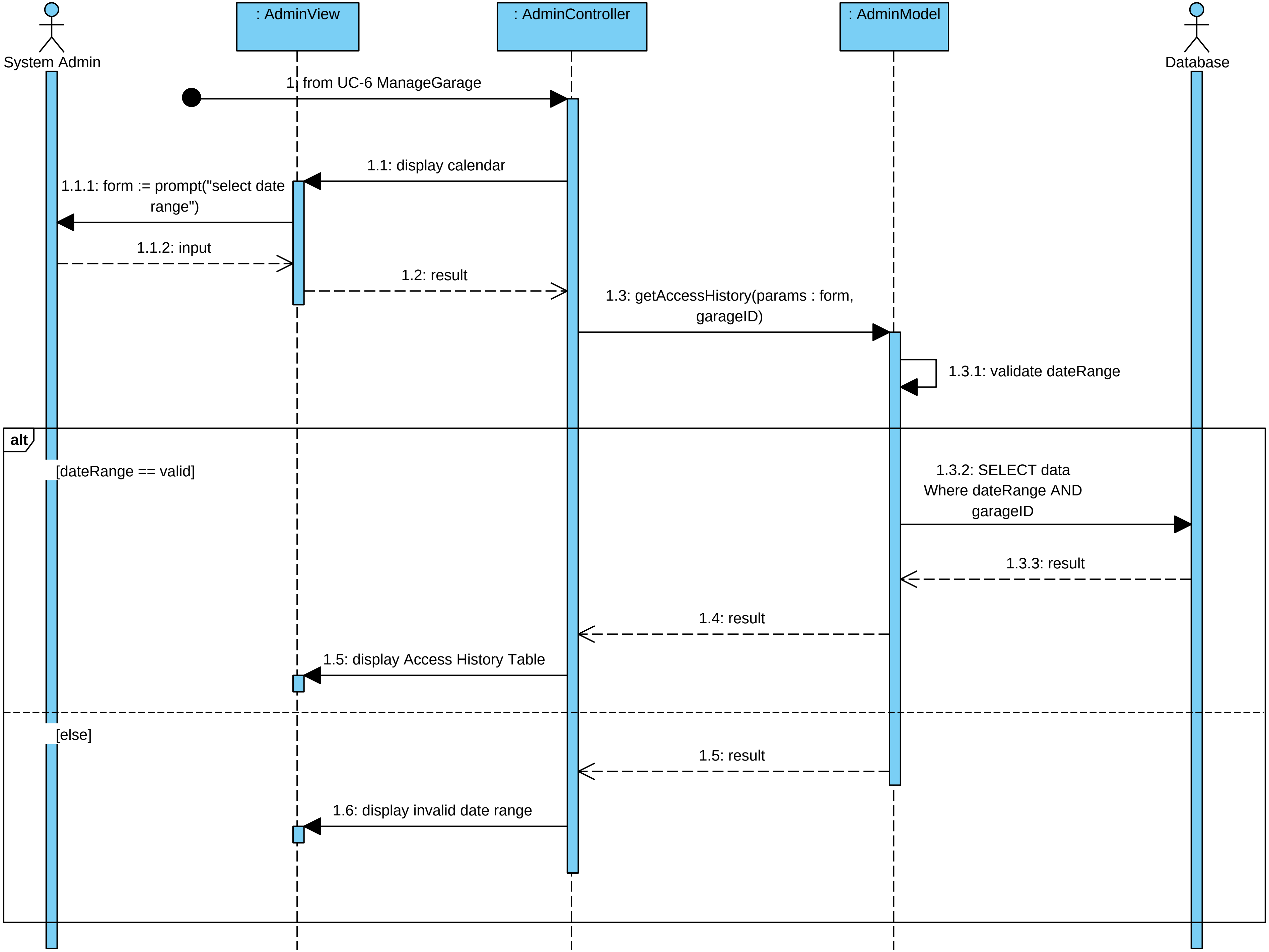
Process: The system admin requests via a web browser to change the prices for a garage, and is given a web page containing the old pricing information and a form to change to newer prices. Controller takes this form and passes it to the Model, which extracts the new info from it and updates the Database with that pricing data. The web page the system admin sees is then updated with the new prices and a confirmation of the changes.


System Admin
: AdminView
: AdminController
: AdminModel
Database
1: from UC-6 ManageGarage
1.1: getCurrentPrices()
1.1.1: SELECT prices Where garageID
1.1.2: result
1.2: result
1.3: display prices
1.3.1: form := prompt("set Garage Prices")
1.3.2: input
1.4: result
1.5: setPrices(params : form, garageID)
1.5.1: validate prices
alt
[prices == valid]
1.5.2: UPDATE newPrices Where garageID
[else]
1.5.3: invalid Prices
1.5.3.1: display invalid prices

UC-12: Inspect Usage History

Goal: To view the usage history of the parking garage.

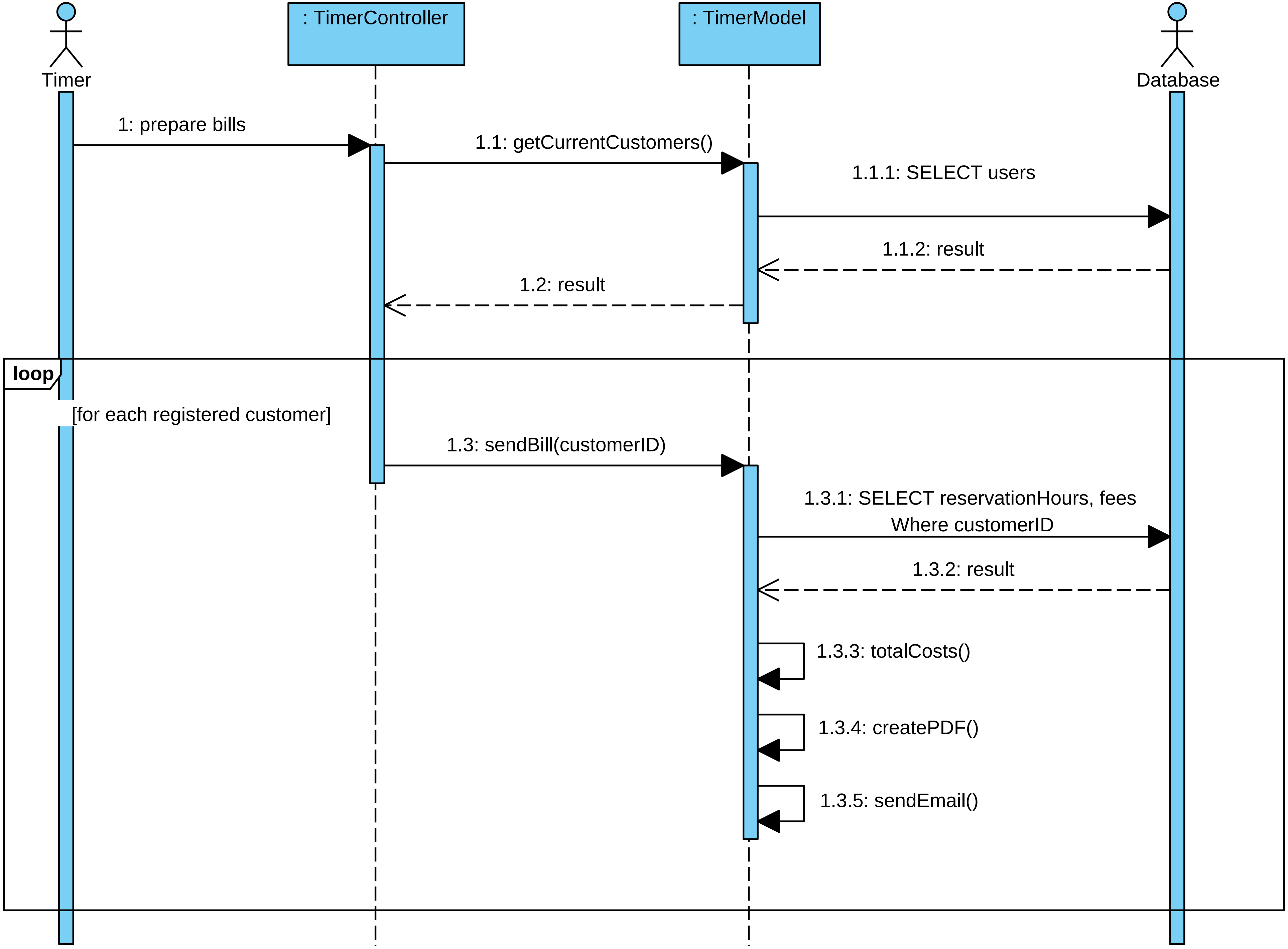
Process: The system admin requests to view the parking garage usage history, and the Controller requests from Model that the history be pulled from the Database for a certain range of dates input by the system admin. The Model retrieves this data and the View displays the statistics to the system admin for review.


System Admin
: AdminView
: AdminController
: AdminModel
Database
1: from UC-6 ManageGarage
1.1: display calendar
1.1.1: form := prompt("select date range")
1.1.2: input
1.2: result
1.3: getAccessHistory(params : form, garageID)
1.3.1: validate dateRange
alt
[dateRange == valid]
1.3.2: SELECT data Where dateRange AND garageID
1.3.3: result
1.4: result
1.5: display Access History Table
[else]
1.5: result
1.6: display invalid date range

UC-13: Monthly Billing

Goal: To generate and send a monthly bill to all registered customers.

Process: The Timer requests the user's billing information from the Model, which extracts the info and create s a bill in PDF form to be emailed to the registered customer.


Timer
: TimerController
: TimerModel
Database
1: prepare bills
1.1: getCurrentCustomers()
1.1.1: SELECT users
1.1.2: result
1.2: result
loop
[for each registered customer]
1.3: sendBill(customerID)
1.3.1: SELECT reservationHours, fees Where customerID
1.3.2: result
1.3.3: totalCosts()
1.3.4: createPDF()
1.3.5: sendEmail()

Class Diagram and Interface Specification

Class Diagrams

For this project, we plan to take advantage of a PHP framework known as Kohana [4]. Therefore, our class diagrams will be a composition between classes that come directly from tables in the Database (see next section, Data Types and Operation Signatures). In addition, we develop a class diagram from our system interaction diagrams. The marriage of these two will be the basis for our system.

The advantage to using the Kohana framework in this way is that we can now run methods against objects in our program rather than SQL queries against the database. From a programming perspective, this is ideal because it allows to always keep an object-oriented view about our program (treating tables in the database as objects in the program). However, it makes our system interaction diagrams slightly more difficult to translate into a class diagram.

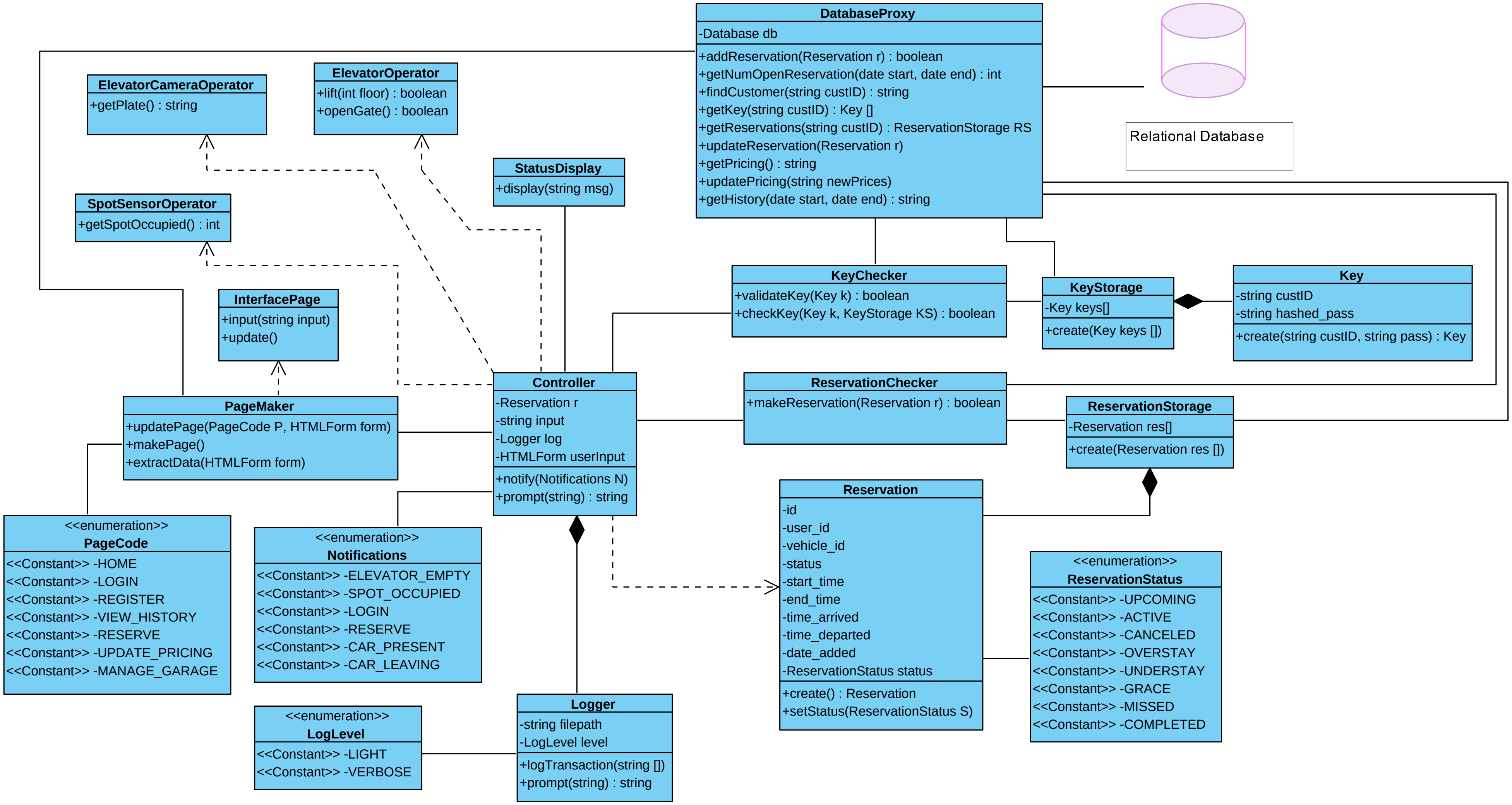
Therefore, we define both a class diagram from our system interaction diagram and also a class diagram for our Kohana framework mapped tables.

To alleviate a lot of the coding needed to implement our system, we will be using an existing, open source PHP framework, Kohana version 3.1. The Kohana framework works on the MVC (model, view, controller) architecture. The basic idea of this system architecture is that business logic is stored in models, customer-facing presentation is coded in views, and controller's handle the interaction between the views (customers) and the models (our system).

Within Kohana, there is exists an ORM (object relational mapping) library that abstracts a lot of the database queries as objects. Within the ORM there are a lot of common methods, such as `save()`, `create()`, and `edit()`. These stock methods will make it easy to implement our system.

Below is a sample of the PHP code inside the Kohana Framework that we might use in our design. It consists of an example model for reservations in the parking garage. This model directly maps to the reservation table in our database, and is abstracted using the ORM.

```
class Model_Reservation extends ORM
{
      protected $_belongs_to = array(
            'user' => array('model' => 'user'),
      );

      public function create_reservation(array $values)
      {
            // Will throw an exception if validation fails
            $this->values($values)->create();

            return TRUE;
      }
}

class Model_User extends ORM
{
      protected $_has_many = array(
            'reservations' => array('model' => 'reservation'),
      );

      public function add_reservation(array $values)
      {
            $reservation = new Model_Reservation;
            $reservation->user_id = $this->id;

            return $reservation->create_reservation($values);
      }
}
```



DatabaseProxy
-Database db
+addReservation(Reservation r) : boolean
+getNumOpenReservation(date start, date end) : int
+findCustomer(string custID) : string
+getKey(string custID) : Key []
+getReservations(string custID) : ReservationStorage RS
+updateReservation(Reservation r)
+getPricing() : string
+updatePricing(string newPrices)
+getHistory(date start, date end) : string

ElevatorCameraOperator
+getPlate() : string

ElevatorOperator
+lift(int floor) : boolean
+openGate() : boolean

StatusDisplay
+display(string msg)

SpotSensorOperator
+getSpotOccupied() : int

InterfacePage
+input(string input)
+update()

KeyChecker
+validateKey(Key k) : boolean
+checkKey(Key k, KeyStorage KS) : boolean

KeyStorage
-Key keys[]
+create(Key keys [])

Key
-string custID
-string hashed_pass
+create(string custID, string pass) : Key

PageMaker
+updatePage(PageCode P, HTMLForm form)
+makePage()
+extractData(HTMLForm form)

Controller
-Reservation r
-string input
-Logger log
-HTMLForm userInput
+notify(Notifications N)
+prompt(string) : string

ReservationChecker
+makeReservation(Reservation r) : boolean

ReservationStorage
-Reservation res[]
+create(Reservation res [])

<<enumeration>>
PageCode
<<Constant>> -HOME
<<Constant>> -LOGIN
<<Constant>> -REGISTER
<<Constant>> -VIEW_HISTORY
<<Constant>> -RESERVE
<<Constant>> -UPDATE_PRICING
<<Constant>> -MANAGE_GARAGE

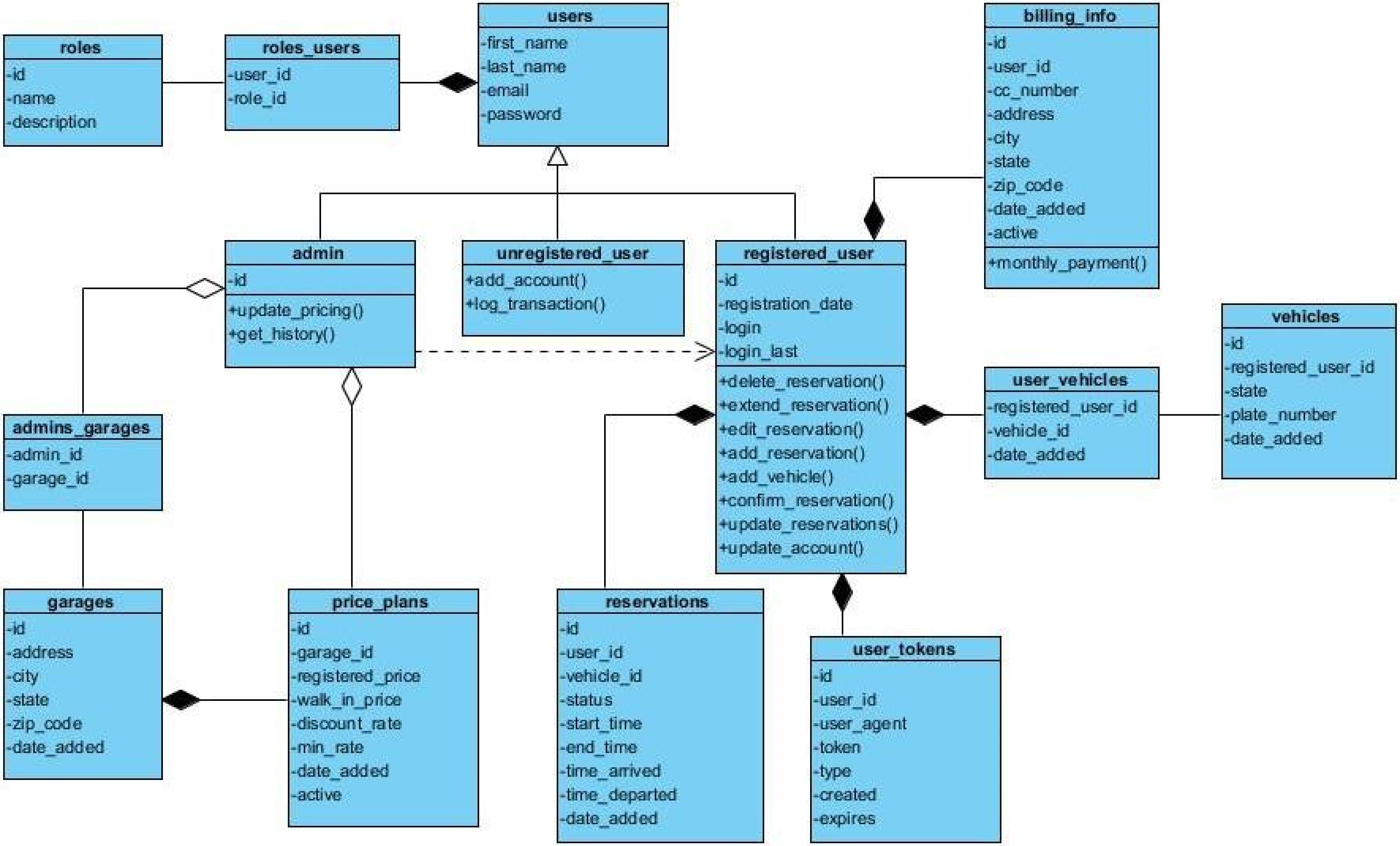
<<enumeration>>
Notifications
<<Constant>> -ELEVATOR_EMPTY
<<Constant>> -SPOT_OCCUPIED
<<Constant>> -LOGIN
<<Constant>> -RESERVE
<<Constant>> -CAR_PRESENT
<<Constant>> -CAR_LEAVING

Reservation
-id
-user_id
-vehicle_id
-status
-start_time
-end_time
-time_arrived
-time_departed
-date_added
-ReservationStatus status
+create() : Reservation
+setStatus(ReservationStatus S)

<<enumeration>>
ReservationStatus
<<Constant>> -UPCOMING
<<Constant>> -ACTIVE
<<Constant>> -CANCELED
<<Constant>> -OVERSTAY
<<Constant>> -UNDERSTAY
<<Constant>> -GRACE
<<Constant>> -MISSED
<<Constant>> -COMPLETED

<<enumeration>>
LogLevel
<<Constant>> -LIGHT
<<Constant>> -VERBOSE

Logger
-string filepath
-LogLevel level
+logTransaction(string [])
+prompt(string) : string


roles
-id
-name
-description
roles_users
-user_id
-role_id
users
-first_name
-last_name
-email
-password
billing_info
-id
-user_id
-cc_number
-address
-city
-state
-zip_code
-date_added
-active
+monthly_payment()
admin
-id
+update_pricing()
+get_history()
unregistered_user
+add_account()
+log_transaction()
registered_user
-id
-registration_date
-login
-login_last
+delete_reservation()
+extend_reservation()
+edit_reservation()
+add_reservation()
+add_vehicle()
+confirm_reservation()
+update_reservations()
+update_account()
vehicles
-id
-registered_user_id
-state
-plate_number
-date_added
user_vehicles
-registered_user_id
-vehicle_id
-date_added
admins_garages
-admin_id
-garage_id
garages
-id
-address
-city
-state
-zip_code
-date_added
price_plans
-id
-garage_id
-registered_price
-walk_in_price
-discount_rate
-min_rate
-date_added
-active
reservations
-id
-user_id
-vehicle_id
-status
-start_time
-end_time
-time_arrived
-time_departed
-date_added
user_tokens
-id
-user_id
-user_agent
-token
-type
-created
-expires

Data Types and Operation Signatures

The Kohana framework allows us to dynamically map tables from our database directly to object types in our system, which we define below. [4]

garage: (parking spots in our garage)

Attributes:

-int id
-int parking_id: accompanying parking record
-int level_num: the floor where the spot is
-int row_num: the row where the spot is
-int col_num: the column where the spot is
-string license_plate: info about the vehicle in the spot
-string state: info about the vehice in the spot
-bool open: whether the spot is open or taken

Operations:

+bool clear_spot(): clears the spot, for when a user exits the spot
+bool take_spot(values): marks the spot as taken

parking: (parking records)

Attributes:

-int id
-int user_id: user who parked here (NULL for walk-in, unregistered parking)
-int price_plan_id: the price plan that was active when this record was created
-int reservation_id: reservation that this record attaches to (NULL for walk-ins)
-int vehicle_id: the vehicle that the user arrived in (NULL for unregistered vehicles)
-int arrival_time: the time the user actually arrived
-int departure_time: the time the user actually departed

Operations:

+bool start_parking(values): opens a new parking record, starts the timer
+float stop_parking(): stops the timer, computes the amount due

price_plan: (current and previous price plans)

Attributes:

-int id
-float member_price
-float guest_price
-float discount_rate: rate at which members price drops
-float min_price: the minimum acceptable price, even with discounts applied
-int date_added

Operations:

+bool create_price_plan(array values)

+bool activate_price_plan(): activates this price plan, and deactivates all others
+price_plan static get_active_price_plan(): returns the active price plan object

reservation: (keeps records of upcoming/past reservations)
Attributes:
-int id
-int user_id
-int start_time
-int end_time
-bool recurring: whether this is a recurring reservation
-int previous_id: a reference to the reservation directly before this (if recurring)
-int date_added
-int last_edited
-active: whether this reservation is active/cancelled
Operations:
+bool create_reservation(array values): creates a new reservation record, and all others (if recurring)
+bool cancel_reservation(): deactivates (cancels) this reservation, and all that follow (if recurring)
+bool edit_reservation(array values): edits this reservation, and all that follow (if recurring)

roles: (different roles users can take)
Attributes:
-int id
-string name: one of either login, confirmed, or admin
-string description: a description describing the role in some detail
Operations:
N/A

roles_users: (joins users to roles, which is many to many)
Attributes:
-int user_id
-int role_id
Operations:
N/A

users:
Attributes:
-int id
-string first_name
-string last_name
-string email
-string password

-int registration_date

Operations:

+bool add_reservation(array values): creates a reservation, and binds it to this user

+bool add_vehicle(array values): creates a vehicle, and binds it to this user

user_token: (user “remember me” tokens)

Attributes:

-int id

-int user_id

-string user_agent

-string token

-string type

-int created

-int expires

Operations:

+bool create_token()

vehicle:

Attributes:

-int id

-int user_id

-string license_plate

-string state

-int date_added

Operations:

+bool create_vehicle(array values)

+bool remove_vehicle()

Design Patterns

Since our Interaction Diagrams make use of both the Strategy and Protection Proxy design patterns, and since the class diagram can be read almost directly from the Interaction Diagrams, the patterns applied there will carry over heavily into the class diagram.

To implement the Protection Proxy we would need to incorporate a Proxy object for each type of proxy we plan to have, which is:

1. Customer Proxy (proxyRC)
2. Adminstrator Proxy (proxyAD)

These proxies would represent an abstract layer between the database and the user, where each proxy has its own individualized set of commands it can operate on the database. For instance, the administrator proxy can alter the prices stored in the database, but the customer proxy cannot.

The strategy proxy we implement in UC-2 Park does not need any additional classes to function. Rather, it is implemented in the logic of the coding.

Object Constraint Language (OCL) Contracts

OCL Contracts are constraints on a class or operation that enables the users, implementers and extenders to share the same assumptions or rules about the class/operation. This kind of contracts determine certain constraints on the class state that must be valid (always unless there are exceptions) so that the operation can work properly. There are three types of OCL constraints: invariants, preconditions, and postconditions.

We will analyze our classes and show these three constrains for each class:

CLASS Garage

-*Invariants.*- Must have spots available

```
context Controller inv:
self.garage ~= FULL
```

-*Pre-conditions*.- Still has available spots, must not be full.

```
context Controller : Boolean pre
self.garagespotsavailable >0
```

-*Post-conditions*.- Is occupied/has less spots available. If full, do not allow further reservations/walk-ins.

CLASS Vehicle

-*Invariants*.- Must be a valid vehicle (have insurance, plates and be driven by a licensed user).

```
context Controller inv:
self.vehicle == valid
```

-*Pre-conditions*.-The vehicle has not been register under another customer or has been stolen

```
context Controller : Boolean pre:
if license_plate exists
then NULL;
else proceed;
```

-*Post-conditions*.-The vehicle enters the database and is a register vehicle in our system.

CLASS User

-*Invariants.*- Must be a registered user that confirmed its registration by email

```
context Controller inv:
email && password == valid;
```

-*Pre-conditions*.- Must have registered before.

```
context Controller : Boolean pre:
self.registered_customer ==valid;
```

-*Post-conditions*.- Adds information that was inputted by the user (add reservation or vehicle or updated information)

```
post: if add_reservation == valid || add_vehiclle ==valid
      then the reservation or vehicle gets bind to this customer
```

```
        else NULL;
```

CLASS Roles-Users

-*Invariants.*-Must be a registered user

```
    context Controller inv:
    email && password == valid;
```

-*Pre-conditions*.- N/A

-*Post-conditions*.- N/A

CLASS Parking

-*Invariants.*- A vehicle needs to park

```
    context Controller inv:
    self.parking == true;
```

-*Pre-conditions*.- The vehicle must be at the elevator where the spot will be assigned and park where assigned.

-*Post-conditions*.- The vehicle has left the spot and now the spot is empty and ready for someone else to use.

```
    post: start_parking =0;
          stop_parking =0;
```

CLASS Roles

-*Invariants.*- A user is trying to use the system

-*Pre-conditions*.- This user must be either logging in, a register user or an admin.

```
    context Controller : Boolean pre:
    email && password ==valid;
```

-*Post-conditions*.- N/A

CLASS User-Token

-*Invariants.*- Must be a valid registered user

```
    context Controller inv:
    email && password ==valid;
```

-*Pre-conditions*.- N/A

-*Post-conditions*.- N/A

CLASS Reservation

-*Invariants.*- Must be a valid registered user

```
    context Controller inv:
    email && password ==valid;
```

-*Pre-conditions*.- The garage cannot be full or must have space between the requested interval.

```
    context Controller (garagespotsavailable) Boolean pre:
    garagespotsavailable > 0;
```

-*Post-conditions*.- Garage has less space.

```
    post: garagespotsavailable = garagespotsavailable -1;
```

CLASS Price-Plan

-*Invariants.*- N/A

-*Pre-conditions*.- Must be an admin/manager

context Controller: Boolean **pre**:

email && password == valid admin;

-*Post-conditions*.- Updates the prices in the database and browser.

System Architecture and System Design

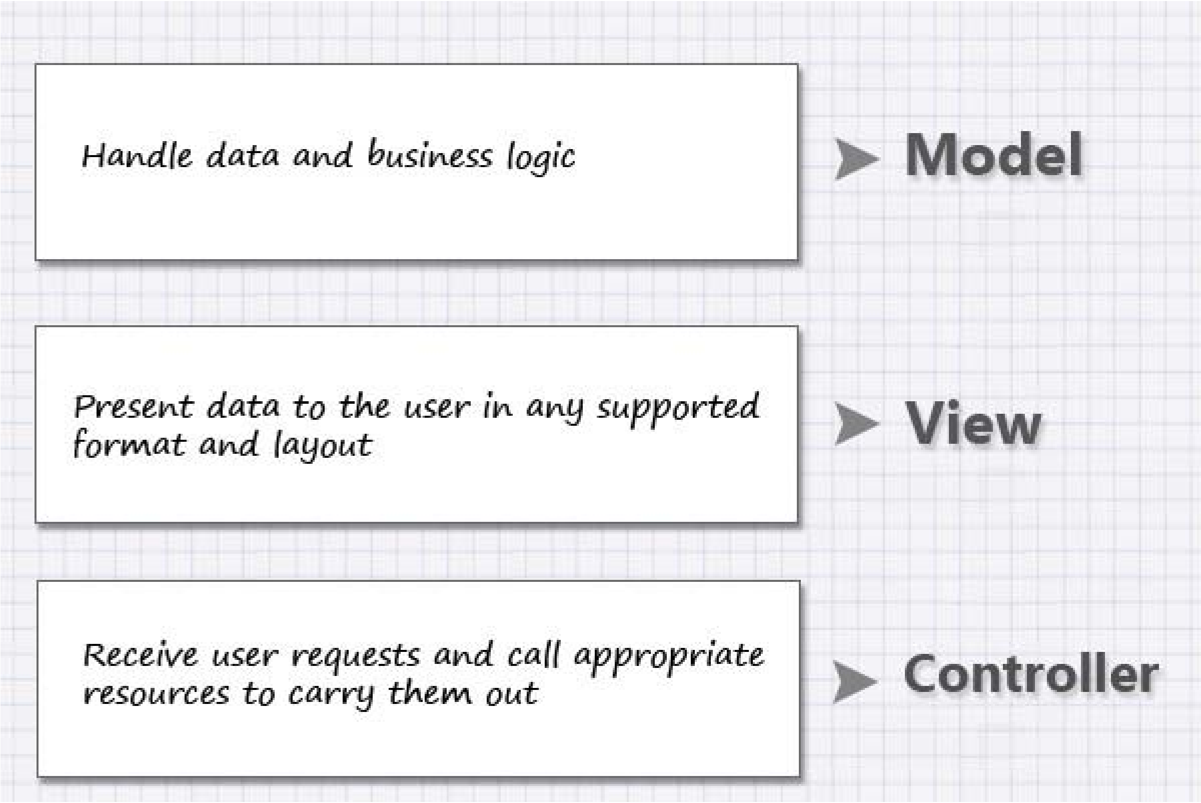
Architectural Styles

The structure used in our project resembles the Event-Driven architecture. The EDA is a pattern that promotes the production, detection, consumption of, and reaction to certain events.
Our software is based on the customers requesting to park their vehicles in our garage. Therefore our system architecture is based on the event of the customer wanting to reserve a parking space [1].

Different entities change state in our system. A reservation can have multiple states during its life cycle. When a customer cancels a reservation, its state changes from “active” to “canceled”. If a customer fails to leave the garage after the reservation time has ended, its state changes from “active” to “overstay”. The following table sums up the events and states for a reservation.

Events for Reservation	Pre-State	Post-State
Created	None	“upcoming”
Canceled	“upcoming”	“canceled”
Extended	“active”, “upcoming”	Same as Pre-State
Overstay	“active”	“overstay”
Understay	“active”	“understay”
Grace	“upcoming”	“grace”
Missed	“grace”	“missed”
Completed	“active”, “overstay”, “understay”, “missed”	“completed”

The framework we will be implementing in “Park-A-Lot” is a MVC (Model View Controller) framework. The main point in MVC is straight forward: the following responsibilities must be clearly separated.


Handle data and business logic
Model
Present data to the user in any supported format and layout
View
Receive user requests and call appropriate resources to carry them out
Controller

The controller deals with the user requests. It controls and coordinates the things needed in order to execute the user request. The model consists of the data and the rules or policies regarding the data. The view creates a way to represent the data obtained from the model [2].

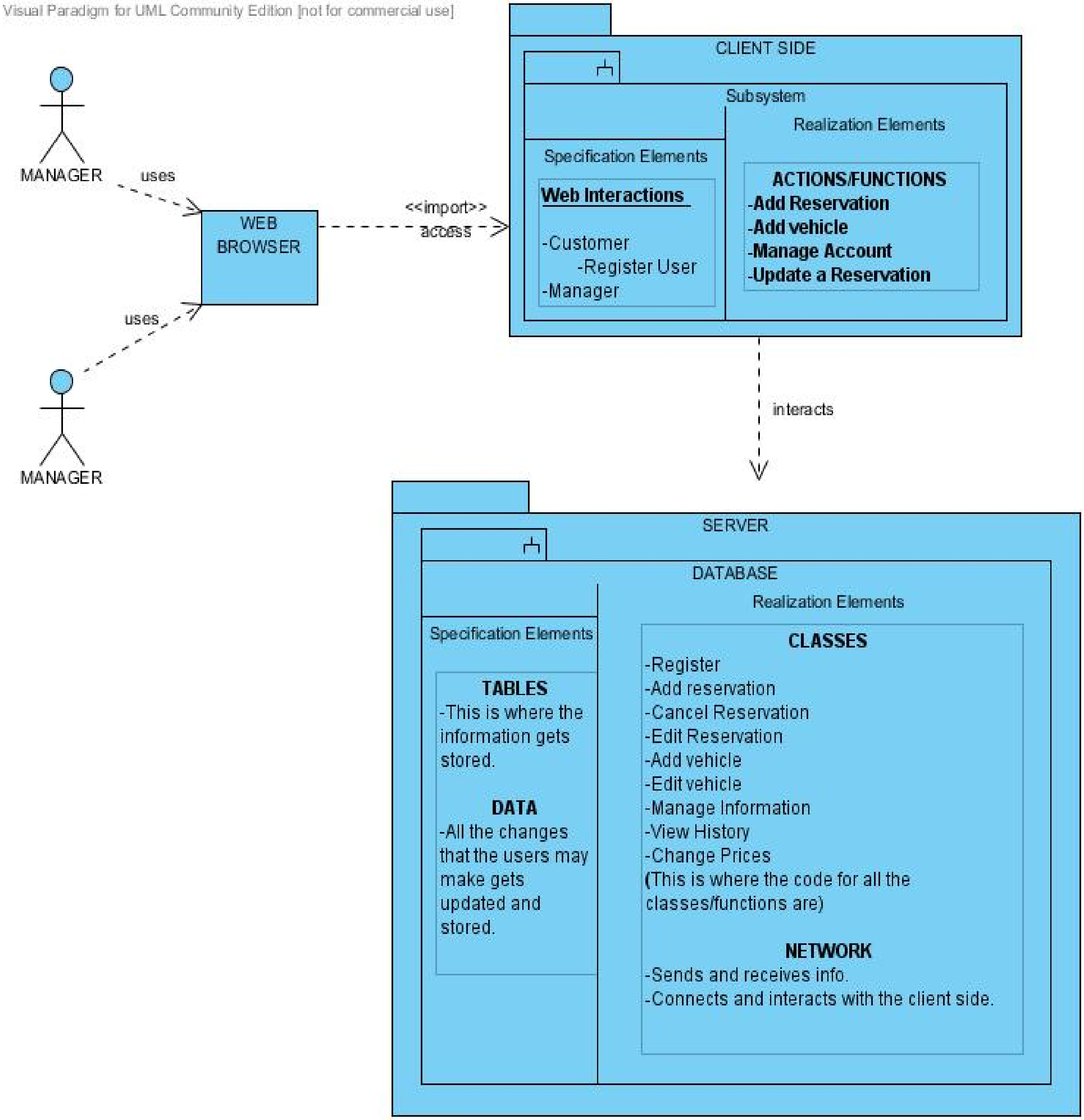
Identifying Subsystems

Our system will make use of many existing software packages and libraries. The following list describes in short the different packages we will use in implementing our system.

- Kohana PHP framework
- Kohana ORM module (object relational mapping)
- Kohana Validation module (validates a plethora of data, from emails to credit cards)
- PayPal, Google Checkout (accept payments from out customers)
- Kohana Auth module (used to login/logout and keep track of customers and administrators)
- jQuery Javascript library
- jQuery Calendar plugin (used for easy reservation scheduling)
- Google Maps plugin (used to locate nearby garages)
- Bluetrip CSS framework

Since our system is to going to be built as a website/web service, we will need to design both the server side and client side(s) that will make up our system. The end user will interact with the main server using any standard web browser using a standard HTTP connection. The client side will be implemented in HTML/CSS/JavaScript, while the server side will be implemented in PHP. If time permits, we also plan to add other clients, specifically as mobile applications on mobile phones.

The diagram below shows the UML package diagram for our system. It is divided into two sub-systems. The first one is the client side, which through the use of a browser chooses different actions and then this sub-system sends the info to the server. The server, which is the other sub-system, is where the code for all the classes is located. It also contains the database with all the info of the customer, rates, and network settings (with other garages).


Visual Paradigm for UML Community Edition [not for commercial use]
MANAGER
uses
MANAGER
uses
WEB BROWSER
<<import>>
access
CLIENT SIDE
Subsystem
Specification Elements
Web Interactions
-Customer
-Register User
-Manager
Realization Elements
ACTIONS/FUNCTIONS
-Add Reservation
-Add vehicle
-Manage Account
-Update a Reservation
interacts
SERVER
DATABASE
Specification Elements
TABLES
-This is where the information gets stored.
DATA
-All the changes that the users may make gets updated and stored.
Realization Elements
CLASSES
-Register
-Add reservation
-Cancel Reservation
-Edit Reservation
-Add vehicle
-Edit vehicle
-Manage Information
-View History
-Change Prices
(This is where the code for all the classes/functions are)
NETWORK
-Sends and receives info.
-Connects and interacts with the client side.

[3]

Mapping Subsystems to Hardware

The system can be broken into three main parts: a web server, a client terminal.

Web Server

The web server runs the majority of the code, maintains a relational database concerning details of the parking garage and its reservations, and accepts input from a client-side system that allows reservations to be created and user accounts to be managed.

The server stores three main types of data: garage information, reservations, and user accounts. The garage information are things such as garage capacity and pricing structure for the garage. The reservations is the reservation records made by customers, either in advance or on the spot as walk ins. Finally, the user account info contains usage history for each customer, and a log of user activity on the system, which can be used to determine discounts for particularly well behaved customers.

Client Side

The client side is much simpler. Basically, it consists of a web browser capable of executing HTML and some CSS. Most of the elements on the client side are HTML forms, which harvest data from the user and relay it to the web server which handles all processing and data manipulation.

In the future, it will be possible to have the client side also run asnative Android and iOS apps. That functionality will be similar to the web browser based system, just with a nicer interface on top. However, the functionality of harvesting data from a user and relaying it to the server will remain the same.

Persistent Data Storage

Our system requires us to keep track of several entities. We need to keep track of

- Customers
- Customer reservations
- Customer vehicles
- Customer billing information
- Garages (which implement our system)
- Garage price plans
- Garage administrators, and their privileges

We will be using a relational database(s), MySQL. We chose MySQL because of its reputation, its price, and because its open source software that we can alter to fit our needs if need be.

Table of contents

1 garage

Creation: May 06, 2011 at 10:03 AM

Field	Type	Attributes	Null	Default	Extra	Links to	Comments	MIME
id	int(10)	UNSIGNED	No		auto_increment			
parking_id	int(10)	UNSIGNED	Yes	NULL		parking -> id		
level_num	tinyint(3)	UNSIGNED	No					
row_num	tinyint(3)	UNSIGNED	No					
col_num	int(11)		No					
license_plate	varchar(10)		Yes	NULL				
state	char(2)		Yes	NULL				
open	tinyint(1)		No	1				

2 parking

Creation: May 06, 2011 at 10:05 AM

Field	Type	Attributes	Null	Default	Extra	Links to	Comments	MIME
id	int(10)	UNSIGNED	No		auto_increment			
user_id	int(10)	UNSIGNED	Yes	NULL		users -> id		
reservation_id	int(10)	UNSIGNED	Yes	NULL		reservations -> id		
price_plan_id	int(10)	UNSIGNED	No			price_plans -> id		
vehicle_id	int(10)	UNSIGNED	Yes	NULL		vehicles -> id		
arrival_time	int(10)	UNSIGNED	No					
departure_time	int(10)	UNSIGNED	Yes	NULL				

3 price_plans

Creation: Apr 28, 2011 at 07:31 PM

Field	Type	Attributes	Null	Default	Extra	Links to	Comments	MIME
id	int(10)	UNSIGNED	No		auto_increment			
member_price	float(4,2)	UNSIGNED	No				hourly price for registered customers	
guest_price	float(4,2)	UNSIGNED	No				hourly price for walk ins	
discount_rate	float(2,2)	UNSIGNED	No				discount rate given to members for good attendance	
min_price	float(4,2)	UNSIGNED	No				minimum allowable price, even with discounts	
date_added	int(10)	UNSIGNED	No					
active	tinyint(1)		No	0			only one price plan can be active at a time	

4 reservations

Creation: May 06, 2011 at 10:07 AM

Field	Type	Attributes	Null	Default	Extra	Links to	Comments	MIME
id	int(10)	UNSIGNED	No		auto_increment			
user_id	int(10)	UNSIGNED	No			users -> id		
start_time	int(10)	UNSIGNED	No					
end_time	int(10)	UNSIGNED	No					
recurring	tinyint(1)		No	0				
previous_id	int(10)	UNSIGNED	Yes	NULL		reservations -> id		
date_added	int(10)	UNSIGNED	No					
last_edited	int(10)	UNSIGNED	Yes	NULL				
active	tinyint(1)		No	1				

5 roles

Creation: Apr 28, 2011 at 07:31 PM

Field	Type	Attributes	Null	Default	Extra	Links to	Comments	MIME
id	int(11)	UNSIGNED	No		auto_increment			
name	varchar(32)		No					
description	varchar(255)		No					

6 roles_users

Creation: Apr 28, 2011 at 07:31 PM

Field	Type	Attributes	Null	Default	Extra	Links to	Comments	MIME
user_id	int(10)	UNSIGNED	No			users -> id		
role_id	int(10)	UNSIGNED	No			roles -> id		

7 user_tokens

Creation: Apr 28, 2011 at 07:31 PM

Field	Type	Attributes	Null	Default	Extra	Links to	Comments	MIME
id	int(11)	UNSIGNED	No		auto_increment			
user_id	int(11)	UNSIGNED	No			users -> id		
user_agent	varchar(40)		No					
token	varchar(40)		No					
type	varchar(100)		No					
created	int(10)	UNSIGNED	No					
expires	int(10)	UNSIGNED	No					

8 users

Creation: Apr 28, 2011 at 07:31 PM

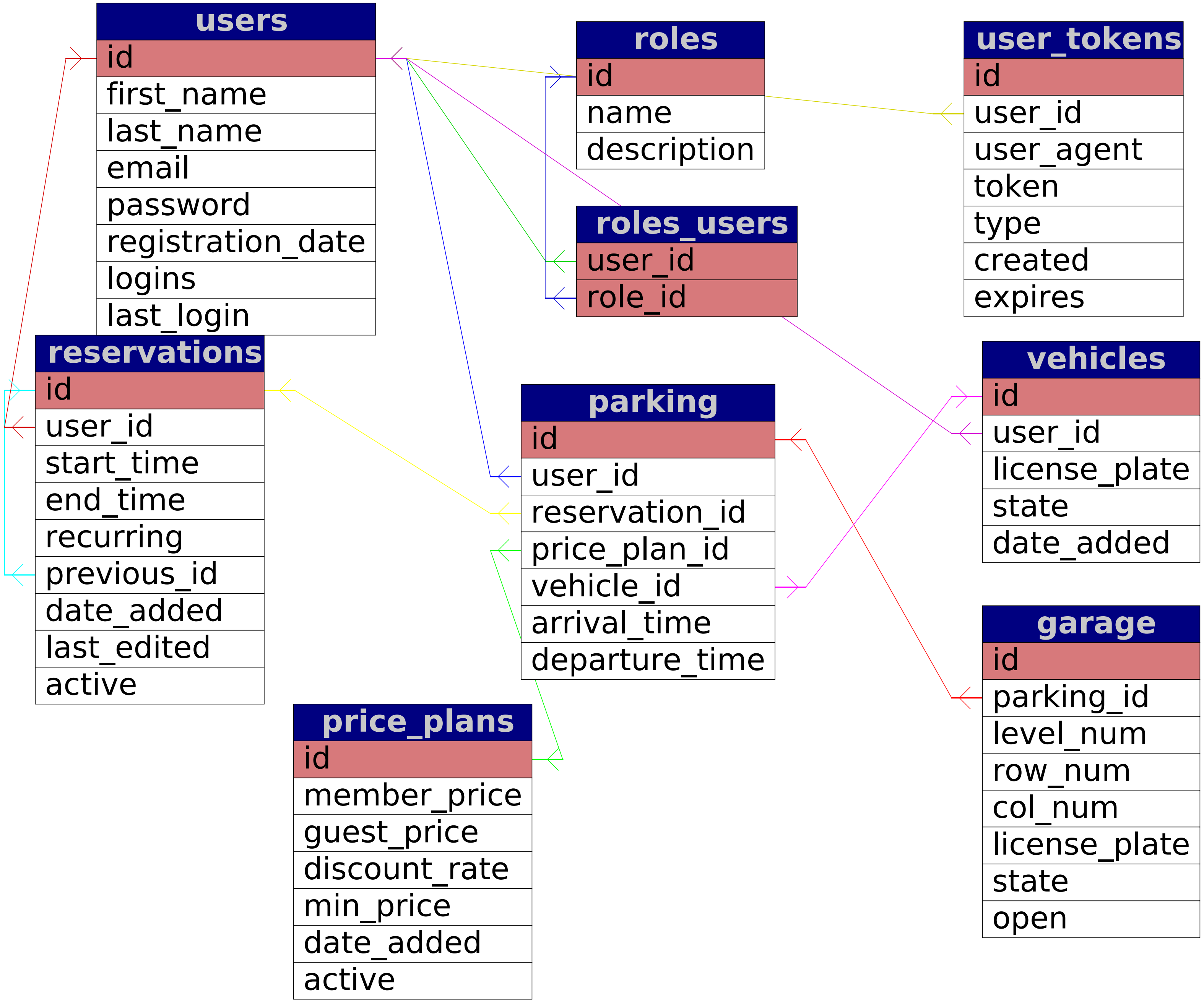
Field	Type	Attributes	Null	Default	Extra	Links to	Comments	MIME
id	int(10)	UNSIGNED	No		auto_increment			
first_name	varchar(30)		No					
last_name	varchar(40)		No					
email	varchar(127)		No					
password	char(64)		No					
registration_date	int(10)	UNSIGNED	No					
logins	int(10)		No	0				
last_login	int(10)		Yes	NULL				

9 vehicles

Creation: May 06, 2011 at 10:08 AM

Field	Type	Attributes	Null	Default	Extra	Links to	Comments	MIME
id	int(10)	UNSIGNED	No		auto_increment			
user_id	int(10)	UNSIGNED	No			users -> id		
license_plate	varchar(10)		No					
state	char(2)		No					
date_added	int(10)	UNSIGNED	No					

No description



Network Protocol

Since our system will be built as a website/web service, and thus built on a single server, there is no need for any communication protocols except the standard HTTP.

Some sensitive data is being passed back and forth (such as the date and time of a customer's reservation), and eventually it may become prudent to implement the option to allow customers to access the site through an SSL connection, in order to ensure total privacy.

Global Control Flow

- **Execution order -** Our software is event-driven so it depends on the customer and his purpose while interacting with the system.
- **Time Dependency -** The only timers in the system is the 15 min inactive log-in time allowed. If a customer logs into his/her account and remains inactive for 15 min then the system will logged him/her out.
- **Concurrency -** No.

Hardware Requirements

Our system requires the following:

- **Keypad -** For the customer to enter information at the elevator.
- **Screen Display** -Minimum resolution of 640 X480 pixels. This display will be implemented in the elevator.
- **Hard Disk/Server -** 50 GB of storage space would satisfy our needs to keep track of all user data, as well as garage and reservation data. With an expectation of supporting 1M users, we would expect to amass about 5GB of data at the end of each year.
- **Sensors -** In order to figure out if the parking space is available or occupied.
- **Cameras -** In order to identify the vehicles coming in the garage and leaving.
- **Network Bandwidth -** Since there is not a huge amount of information being transferred, the amount required should be 56 kbps to access the system successfully.
- **Web browser -** In order for the client to access the software, he/she must use a semi-advance browser (i.e Mozilla Firefox 4.0 or up, Google Chrome, Internet Explorer 8 or up).

Aside from the hardware which is integrated into our system (keypad, display, etc.), the only hardware we require is disk space for our database(s). Looking at our database schema, its evident that our largest table will be the reservations table. The reservation table consists of 11 fields, 10 of which are int fields (4 bytes), and one of which is an enum field (1 byte). Each record in the table will take up 41 bytes. The amount of disk space will depend on the number of customers we intend to support, and the average rate of reservations we think each user will make and how long we will keep old records.

For example: Lets say we intend on supporting up to 100 customers, who make reservations at a rate of 1 per month. That's 1200 reservations a year. That's 1200 * 41 = 49,200 bytes, or 49.2 Kbytes a year. If we intend on holding reservation information for the past 3 years, that's at least 150 Kbytes before we dump out old data. Obviously, we will support a lot more customers and customers will make a lot more reservations, and we will keep data a lot longer... this is just an example. Obviously, other tables will take up data as well, but nothing is expected to grow as fast as the reservations table, so this is what we should be looking to support (I would think). In any case, disk space is cheap, bandwidth is not... so depending on the number of hits we intend on getting a day, we may need heavy processors, fast ram, etc. But, for this project, I don't think bandwidth will be much of an issue.

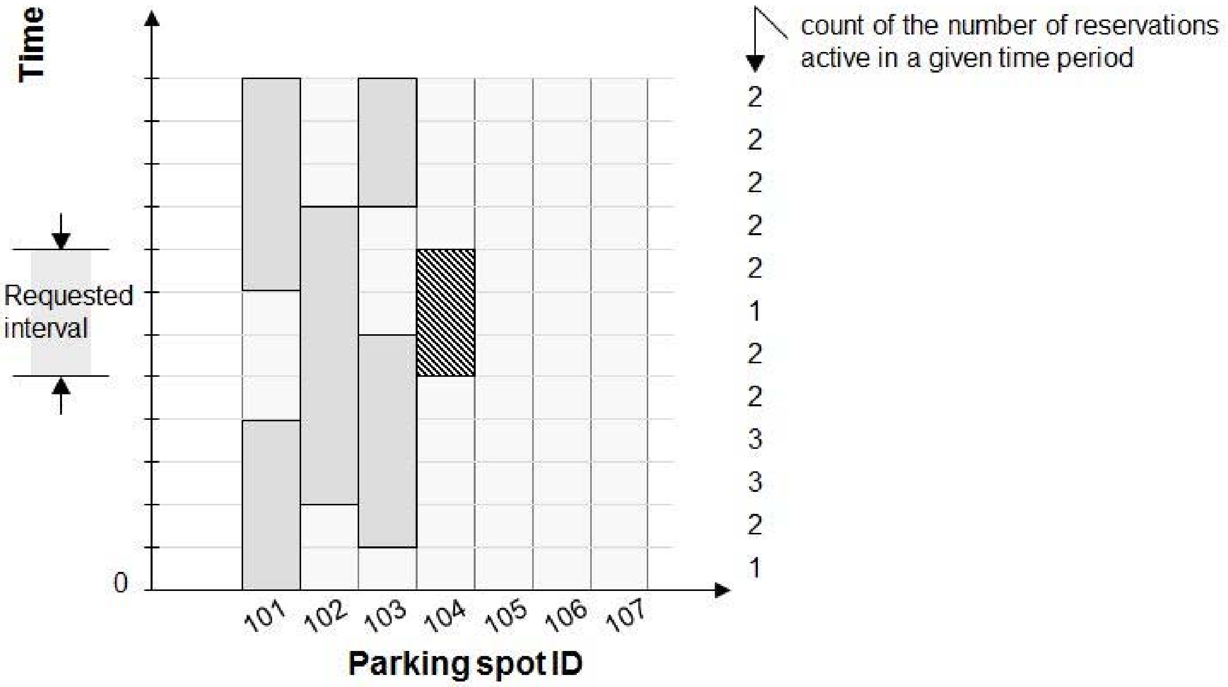
Data Structures and Algorithms

Data Structures

Our system does not use many complicated structures (hash tables, linked lists or trees) but we do use arrays in order to quickly analyze and refer to the data in our database. The reason why we chose arrays over any other method is because it is simple to use and linked lists are not implicitly supported by PHP, the language we will be coding our server side application in.

The main source and storage of all our data will be done in relational databases. Using a technology such as MySQL allows quick access to all of our data, and complicated algorithms can be simplified with the powerful Structured Query Language (SQL).

A main component of our parking reservation system will be the parking lot bitmap, which is simply a two-dimensional map which encodes information about what parking spots are filled at all times. The bitmap will be as follows,



This data can be encoded in a table in the database.
It is important to note that the size of this bitmap can become quite large, especially when coded into a table. Suppose, for a garage with 300 spaces, that we keep track of the next 3 months

worth of reservations. If we consider 30 minute increments of time, this parking lot bitmap table will contain

(30 days/month)*(3 months)*(24 hours/day)*(2 half-hours/hour) x 300 parking spots

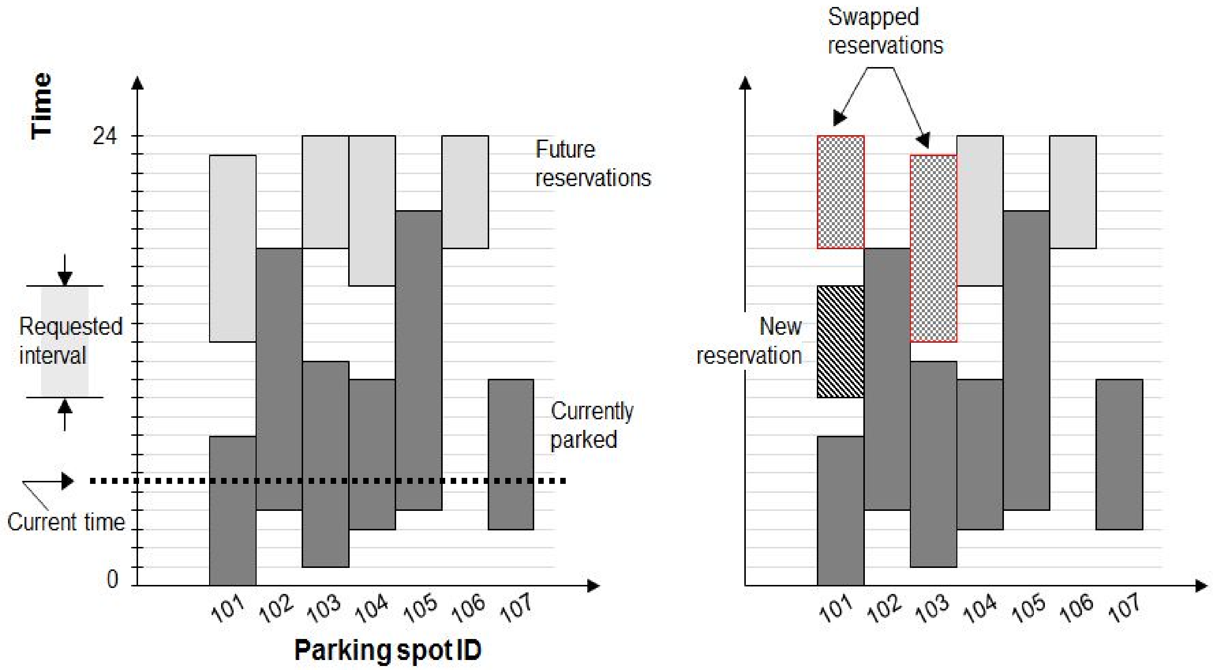
or

4320 rows x 300 columns

Which is an enormous table that would take a very long time to search through. Therefore, it might be better to find a different type of data structure that could store this parking lot bitmap. However, using PHP and Kohana there does not appear to be a more efficient way.

Algorithms

Scheduling Reservations - A problem may arise where a customer wishes to schedule a reservation but is unable to do so because there are no blocks of time available in which to schedule the full reservation. This problem can be represented by considering the following case for our parking lot bitmap:

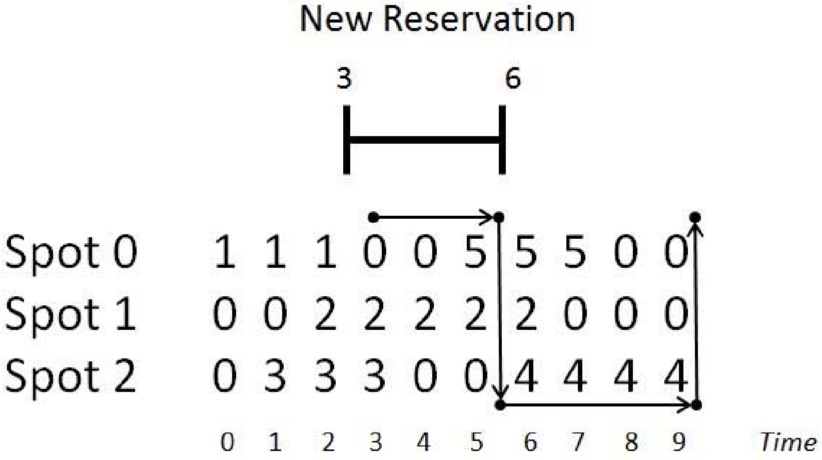


Note that for the original parking lot bitmap it is impossible to schedule the reservation for a future time, because of how we currently have future reservations distributed. However, we can implement a `swap()` function that pairwise compares future reservations to determine if two can be swapped that would allow the new reservation to be created. [5]

The simplest implementation of this algorithm is as follows. The input access the existing parking lot bitmap, encoded as a matrix M. The rows of M are the parking spots in the garage, and the columns of M are the time blocks (could be in 30 minutes increments, 1 minute increments, whatever the system calls for). The reservation R has a start and end time, and a unique reservation number to distinguish it from other reservations.

```
Input:  Parking lot bitmap (as matrix M[spot, time block]), Reservation R, Max number of
        parking spots max_spots
Output:  Parking lot bitmap with new reservation added, if possible

Assign-Reservation(M, R, max_spots)
assigned = false;
for each parking spot m in M until assigned = true
    if m is free from R.start until R.end then
        Assign R to m;
        assigned = true;
if assigned = false then
    if size M = max_spots
        return Swap-Reservation(M, R);
else
    return M;

Swap-Reservation(M, R)
for each parking spot m in M
    if m is free at R.start but not R.end then
        R1 = m.nextReservation;
        for each parking spot n not previously considered in M
            R2 = n.nextReservation;
            if n is free from R1.start until R.end and
              m is free from R1.end until R2.end then
                Swap R1 with R2;
                Assign R to parking spot m;
```

The Swap-Reservation() function simply checks if two reservations can be swapped so that the new reservations can be included in the parking lot bitmap. If we consider the matrix M below with pre-defined reservations in it, and attempt to insert a new reservation (Number: 6, Start: 3, End: 6), the following happens.



Spot 0	1	1	1	0	0	5	5	5	0	0
Spot 1	0	0	2	2	2	2	2	0	0	0
Spot 2	0	3	3	3	0	0	4	4	4	4
	0	1	2	3	4	5	6	7	8	9

The general idea here is to start by searching for a parking spot that can accommodate the start time of the reservation, then seeing which reservation can be swapped with the conflicting reservation in that spot. If a swap is possible, we make it and assign the new reservation. If not, we move on to the next spot in the bitmap, and repeat, this time comparing against all spots (except the first spot, which we have already paired with the second, so no need to repeat that).

The appendix to this report contains a C++ implementation of the algorithm.

As for applying this algorithm to our system, looking at the database schema there are two tables of which to take note: Parking and Reservations. Parking contains all of the information about cars currently parked in the garage - spot number filled, start time, end time, etc. Reservations contains the information about reservations in the system, both past, current, and future. Feeding this information into the algorithm, there are two steps to building and updating the parking bitmap.

1. Look in the Parking table to determine the durations of those parked in currently assigned spots, all of which are "active" and therefore untouchable, i.e. unable to be swapped with other reservations in an attempt to make room for a future reservations. These current parkings will be the basis of our parking bitmap.
2. Second, look in the Reservations table (up until some point in time in the future) and try to place as many reservations into spaces as possible, using the algorithm to assign them. When a new reservation gets added simply try to add it, and if it doesn't fit call the swap() function to see if it can be swapped to make room. If not, it cannot be added.

Each time either Parking or Reservations gets updated, we will need to update the parking bitmap, which is stored as another table inside the database.

The point in the future we consider in Step 1 would probably be either 24 or 48 hours, wherein we consider intervals of 1 minute. Therefore, a table storing this information would have 24*60=1440 rows or 2880 rows, which means the parking bitmap would have a resolution of 1440/2880 time increments, multiplied across the number of spots in the garage.

Best Alternative Reservation - This algorithm will determine the largest contiguous block of time contained in the period between a customer's desired reservation start and end times, in the event that a reservation for the entire block of time is unavailable.

For example, if a customer wants a confirmed reservation from 9am until 5pm and the creation of that reservation would overbook the garage for any amount of time within that interval, then the customer would be offered the largest contiguous block of time within that 9-5 window that would not create an overbooked garage.

```
Input:  A reservation R, Parking Lot Bitmap M
    Output:  Largest contiguous reservation possible, G

    Find-Best-Alternative(M, R)
    maxBlock = 0;
    G = NULL;
    for each parking spot m in M
        conflict = m.nextReservation;
        if (R.start - conflict.start) > maxBlock then
            maxBlock = R.start - conflict.start;
            G = new Reservation starting at R.start and ending at conflict.start;
```

Mathematically this is a very simple algorithm. We look through the parking lot bitmap to determine which spot offers the most amount of time matching the given start and end times of a desired reservation. Then return a reservation with those characteristics, which the customer can choose to reserve instead.

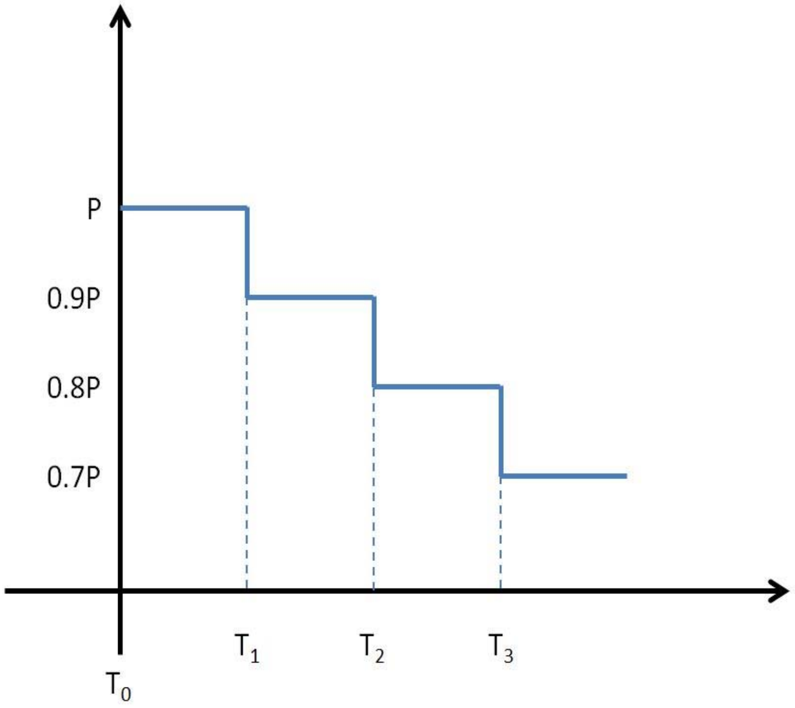
This algorithm will help the customer select a reservation. The customer may not know that their requested reservation of 9am until 5pm is unmanageable, but perhaps a 9am-4:30pm reservation is possible.

Discount Price - This algorithm will determine the discounted price a registered customer may pay for parking. After a registered customer has completed some amount of “perfect reservations” for a given time period, i.e. reservations for which the customer both arrived on time and departed on time the pricing for each upcoming reservation will be offered at a discounted price per hour.

This is actually a very simple algorithm. It is a tiered pricing structure which moves the pricing per hour for a registered customer into a lower pricing tier if the customer has a solid record of arriving a departing on time. We will call this type of a customer a “good” customer.

Registered customer’s normally pay a fixed price $P per hour. The price will drop as a percentage of $P as the customer’s performance record over the time window W reaches certain thresholds, T[i] where i = 0, 1, 2...

The following step function illustrates this idea.



Note how for different thresholds of T[] the pricing decreases This price structure is completely customizable per each garage, and different thresholds can be set for customer performance. Each value of T[] must be a statistical percentage of reservations completed on time, compared with all reservations, some of which may have been either overstays or understays.

Moreover, a customer must have at least 10 reservations completed for these discounts to take affect. For example, a customer with only 1 reservation completed on-time will not be eligible for the 100% completed discount rate. Rather, the rates will only apply after a customer has accrued more than 10 completed reservations to ensure fairness.

As an example, the following values for T[] and P are possible.

P = $20, 0.9P = $18, 0.8P = $16, 0.7P = $14
T[0] = 0%, T[1] = 80%, T[2] = 90%, T[3] = 95%.

So a customer who arrived on time and departed on time 95% of the time would only pay $14 per hour, as opposed to the regular rate of $20 per hour.

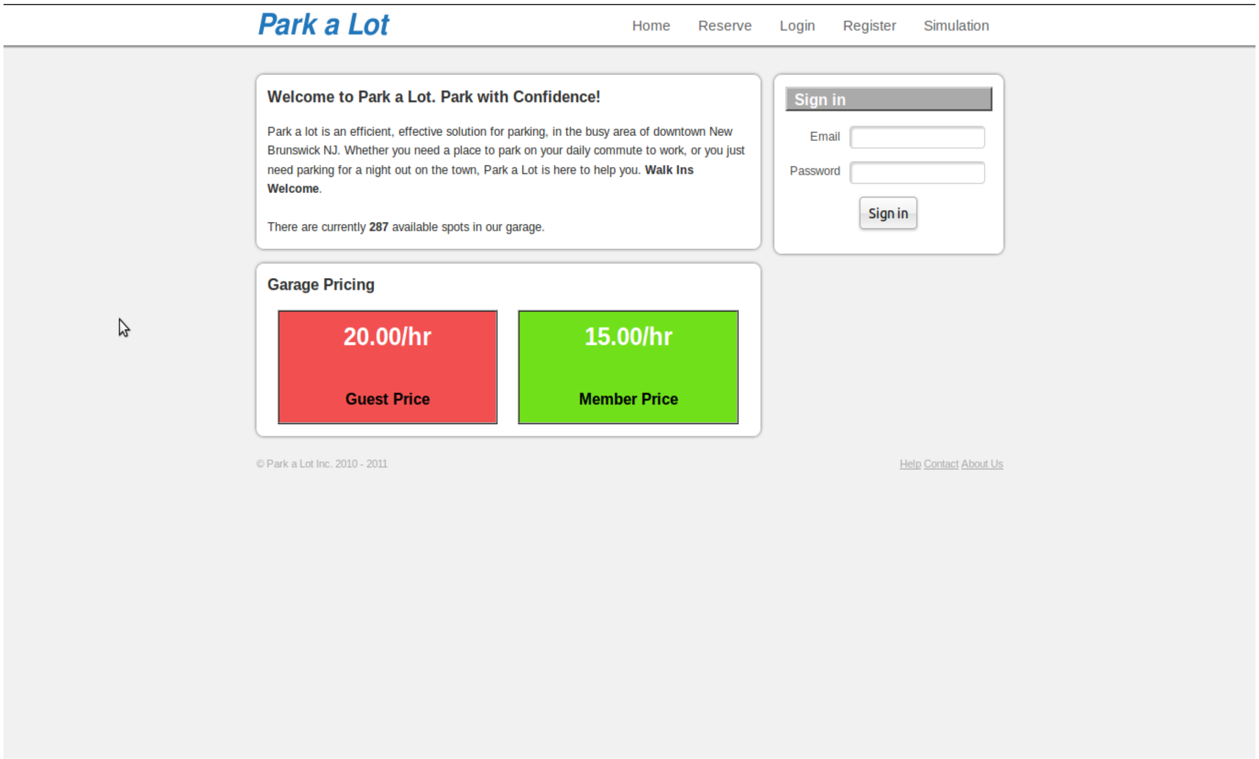
User Interface Design and Implementation

A few significant additions/edits were made to the user interface to increase the user experience. All mock-ups can be seen live at http://www.park-a-lot.vacau.com/. The design is meant to reduce user effort to a minimum, by providing a sleek and reduced graphical interface that is simple to understand (no screen clutter or extraneous information).

The differences between our previous design and our current design go as follows:

Home Page

Since we did not actually implement the multi-garage feature, we've removed the Google Map from our homepage. We've also added a price breakdown on our home page so customers can easily see our current prices for members and guests alike.



Welcome to Park a Lot. Park with Confidence!

Park a lot is an efficient, effective solution for parking, in the busy area of downtown New Brunswick NJ. Whether you need a place to park on your daily commute to work, or you just need parking for a night out on the town, Park a Lot is here to help you. **Walk Ins Welcome**.

There are currently **287** available spots in our garage.

Garage Pricing

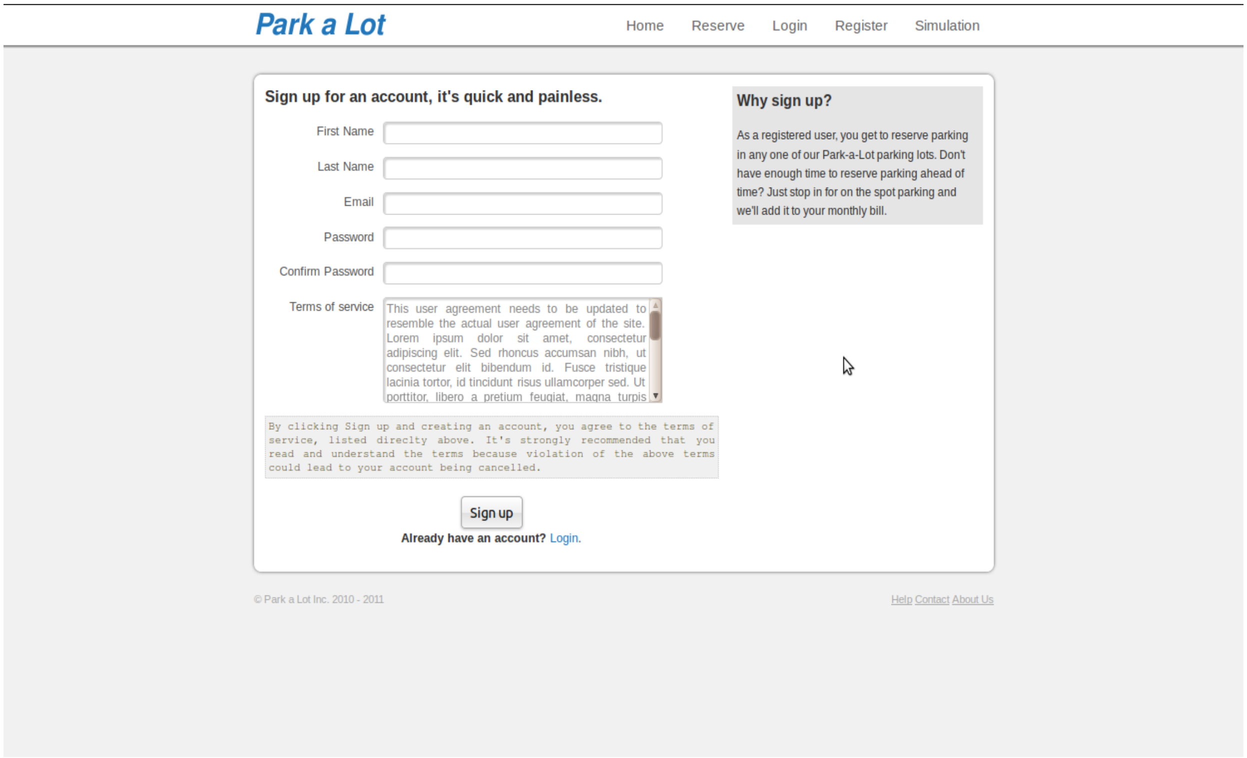
🟥	🟩
20.00/hr	15.00/hr
Guest Price	Member Price

© Park a Lot Inc. 2010 - 2011

Help Contact About Us

User Registration

We've eased up on our user registration form, making it easier and quicker for users to sign up. We no longer require any credit card information at sign up.



Sign up for an account, it's quick and painless.

First Name

Last Name

Email

Password

Confirm Password

Terms of service

This user agreement needs to be updated to resemble the actual user agreement of the site. Lorem ipsum dolor sit amet, consectetur adipiscing elit. Sed rhoncus accumsan nibh, ut consectetur elit bibendum id. Fusce tristique lacinia tortor, id tincidunt risus ullamcorper sed. Ut porttitor, libero a pretium feugiat, magna turpis

```
By clicking Sign up and creating an account, you agree to the terms of
service, listed direclty above. It's strongly recommended that you
read and understand the terms because violation of the above terms
could lead to your account being cancelled.
```

Already have an account? Login.

Why sign up?

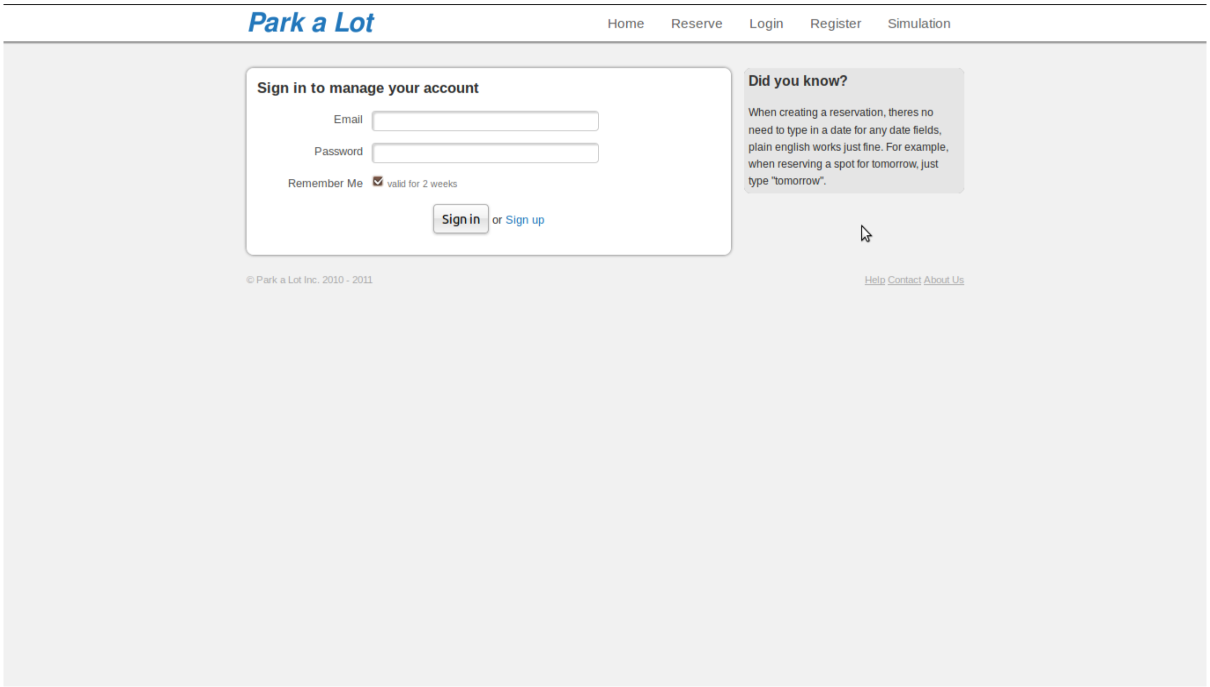
As a registered user, you get to reserve parking in any one of our Park-a-Lot parking lots. Don't have enough time to reserve parking ahead of time? Just stop in for on the spot parking and we'll add it to your monthly bill.

© Park a Lot Inc. 2010 - 2011

Help Contact About Us

User Login

We did not change much with user login, except added a few site-wide tips to our login screen. The login screen as seen from our home page remains the same, but we added a few helpful tips and FAQ to our main login screen.



Sign in to manage your account

Email

Password

Remember Me ☑ valid for 2 weeks

or Sign up

Did you know?

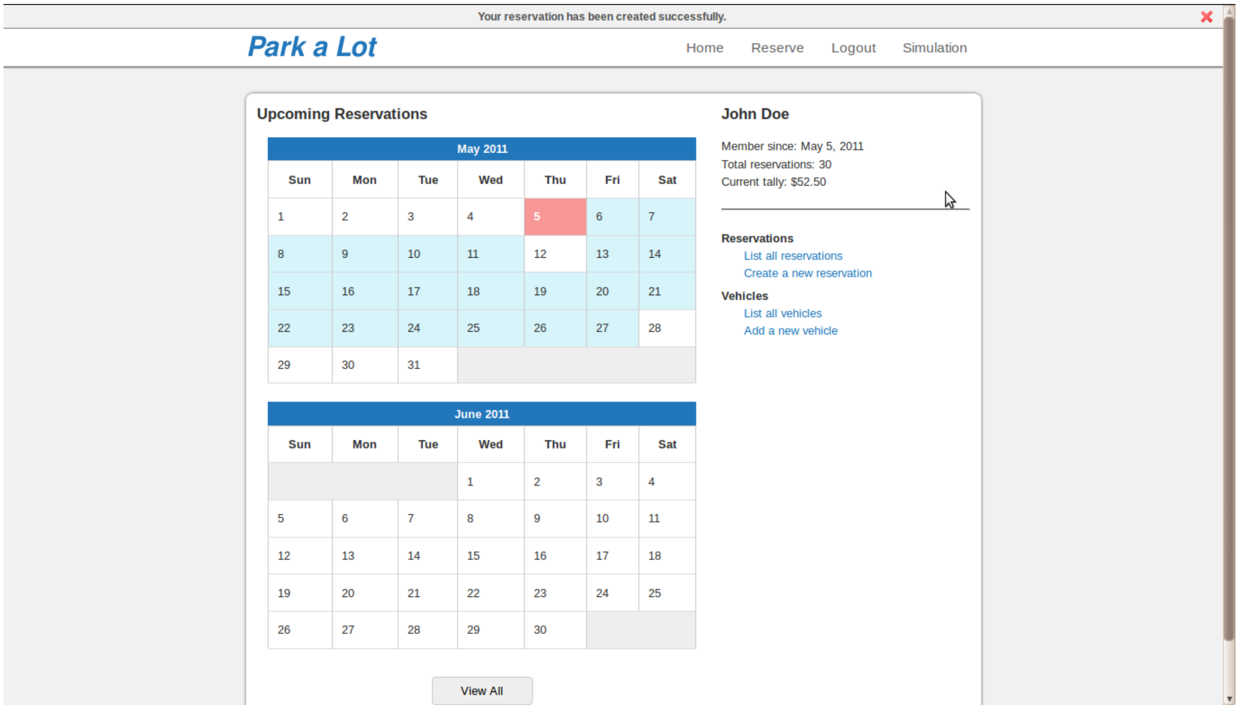
When creating a reservation, theres no need to type in a date for any date fields, plain english works just fine. For example, when reserving a spot for tomorrow, just type "tomorrow".

© Park a Lot Inc. 2010 - 2011

Help Contact About Us

User Profile

The user profile page contains links to all important pages from a users standpoint. From there, they can go to the create reservation page, create vehicle page, list reservations page, list vehicles page, as well as view any reservations for a specific day by clicking on a date from the calendars shown, which summarize which days the user has a reservation in the next 2 months. Also, from the home page, the user can see how much they have on their current monthly bill to date. Every action that the user takes, whether it be add a new reservation, cancel an existing one, etc. they are redirected back to their profile and a popup notification is shown at the very top of the page to provide feedback.



Your reservation has been created successfully.

Upcoming Reservations

May 2011

Sun	Mon	Tue	Wed	Thu	Fri	Sat
1	2	3	4	5	6	7
8	9	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28
29	30	31				

June 2011

Sun	Mon	Tue	Wed	Thu	Fri	Sat
			1	2	3	4
5	6	7	8	9	10	11
12	13	14	15	16	17	18
19	20	21	22	23	24	25
26	27	28	29	30		

John Doe

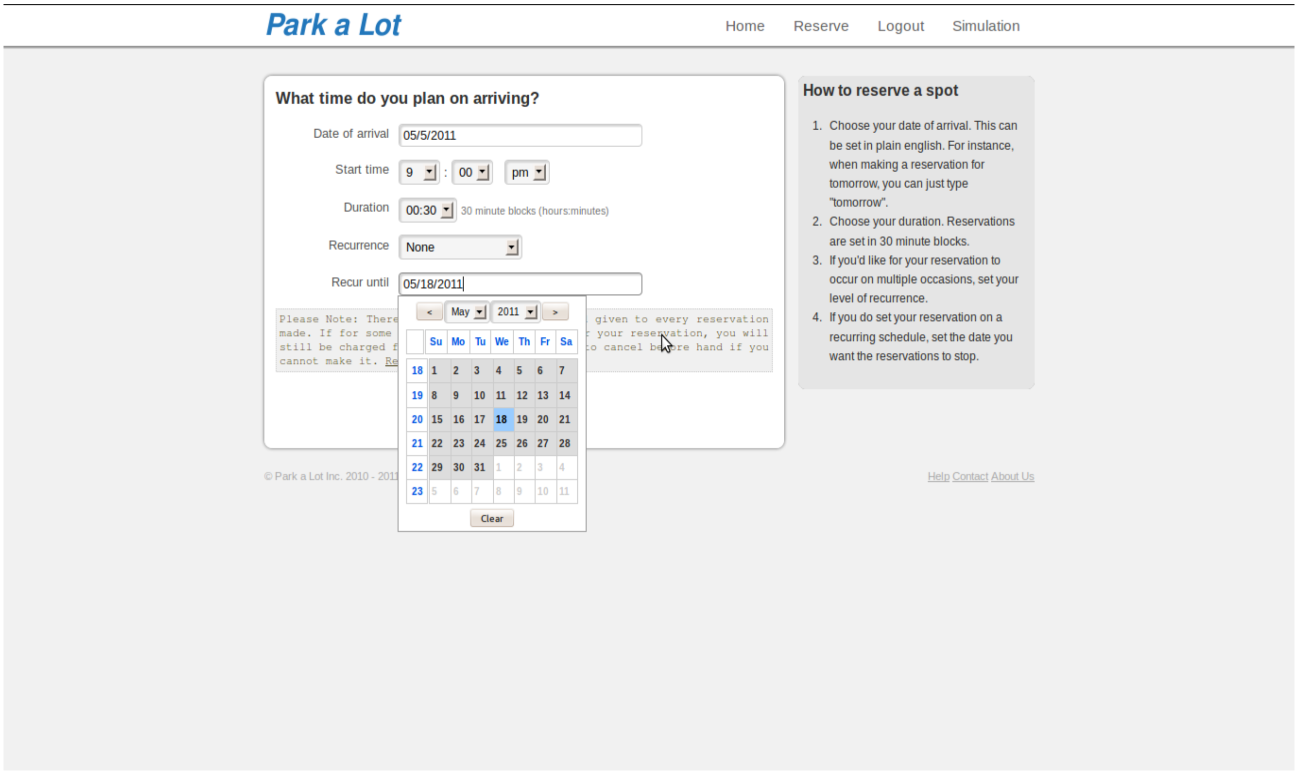
Member since: May 5, 2011
Total reservations: 30
Current tally: $52.50

Reservations

Vehicles

Create Reservation

The create reservation page was updated to add popup calendars to each field that takes a date input to make it easier for the user.



What time do you plan on arriving?

Date of arrival: 05/5/2011

Start time: 9 : 00 pm

Duration: 00:30 — 30 minute blocks (hours:minutes)

Recurrence: None

Recur until: 05/18/2011

	Su	Mo	Tu	We	Th	Fr	Sa
18	1	2	3	4	5	6	7
19	8	9	10	11	12	13	14
20	15	16	17	18	19	20	21
21	22	23	24	25	26	27	28
22	29	30	31	1	2	3	4
23	5	6	7	8	9	10	11

Please Note: There … given to every reservation made. If for some … your reservation, you will still be charged f… to cancel before hand if you cannot make it. Re…

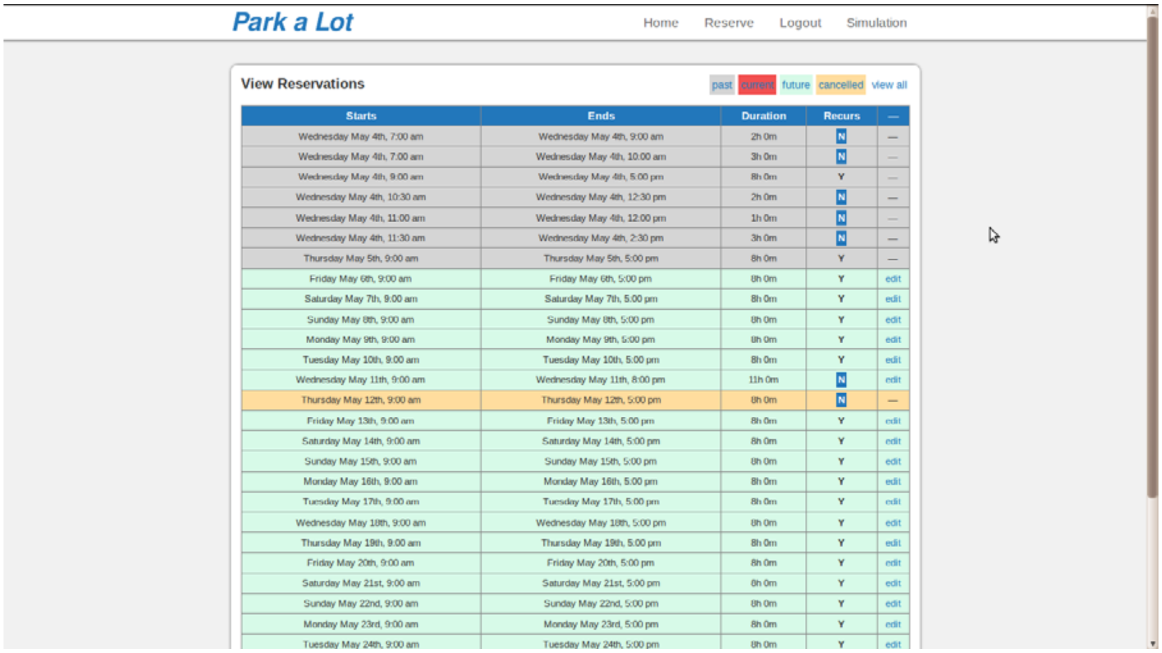
How to reserve a spot

1. Choose your date of arrival. This can be set in plain english. For instance, when making a reservation for tomorrow, you can just type "tomorrow".
2. Choose your duration. Reservations are set in 30 minute blocks.
3. If you'd like for your reservation to occur on multiple occasions, set your level of recurrence.
4. If you do set your reservation on a recurring schedule, set the date you want the reservations to stop.

© Park a Lot Inc. 2010 - 2011

List Reservations

The list reservations page lists in more detail all of the users reservations, past, present, and future. From here, they can choose to edit/cancel any reservation which can still be edited/updated. each reservation is color coded to easily distinguish between past/present/future/cancelled reservations.

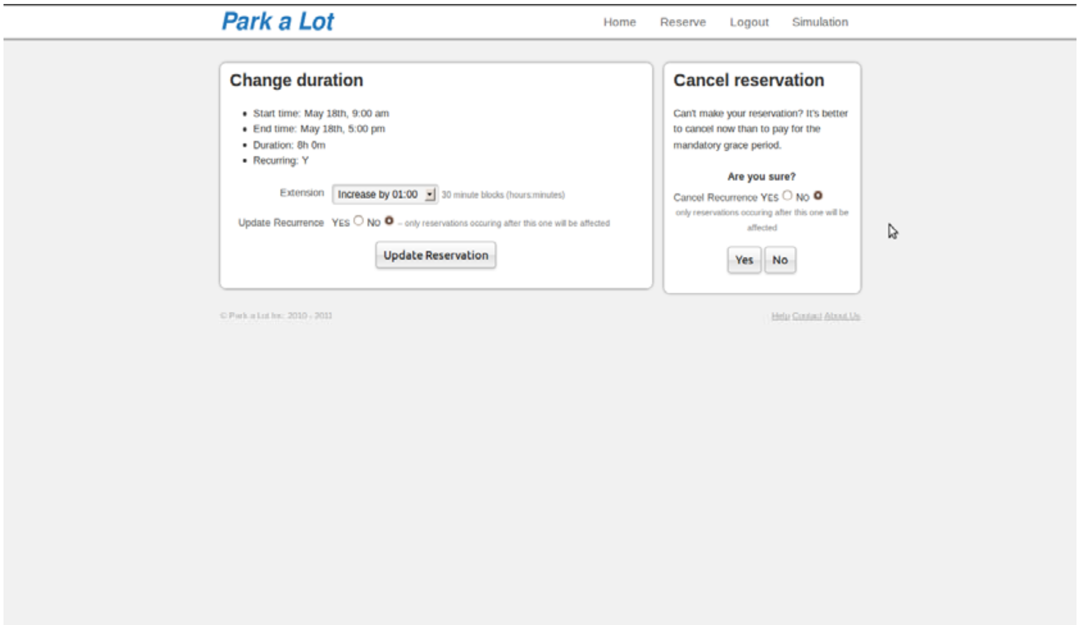


View Reservations

Starts	Ends	Duration	Recurs	—
Wednesday May 4th, 7:00 am	Wednesday May 4th, 9:00 am	2h 0m	N	—
Wednesday May 4th, 7:00 am	Wednesday May 4th, 10:00 am	3h 0m	N	—
Wednesday May 4th, 9:00 am	Wednesday May 4th, 5:00 pm	8h 0m	Y	—
Wednesday May 4th, 10:30 am	Wednesday May 4th, 12:30 pm	2h 0m	N	—
Wednesday May 4th, 11:00 am	Wednesday May 4th, 12:00 pm	1h 0m	N	—
Wednesday May 4th, 11:30 am	Wednesday May 4th, 2:30 pm	3h 0m	N	—
Thursday May 5th, 9:00 am	Thursday May 5th, 5:00 pm	8h 0m	Y	—
Friday May 6th, 9:00 am	Friday May 6th, 5:00 pm	8h 0m	Y	edit
Saturday May 7th, 9:00 am	Saturday May 7th, 5:00 pm	8h 0m	Y	edit
Sunday May 8th, 9:00 am	Sunday May 8th, 5:00 pm	8h 0m	Y	edit
Monday May 9th, 9:00 am	Monday May 9th, 5:00 pm	8h 0m	Y	edit
Tuesday May 10th, 9:00 am	Tuesday May 10th, 5:00 pm	8h 0m	Y	edit
Wednesday May 11th, 9:00 am	Wednesday May 11th, 8:00 pm	11h 0m	N	edit
Thursday May 12th, 9:00 am	Thursday May 12th, 5:00 pm	8h 0m	N	—
Friday May 13th, 9:00 am	Friday May 13th, 5:00 pm	8h 0m	Y	edit
Saturday May 14th, 9:00 am	Saturday May 14th, 5:00 pm	8h 0m	Y	edit
Sunday May 15th, 9:00 am	Sunday May 15th, 5:00 pm	8h 0m	Y	edit
Monday May 16th, 9:00 am	Monday May 16th, 5:00 pm	8h 0m	Y	edit
Tuesday May 17th, 9:00 am	Tuesday May 17th, 5:00 pm	8h 0m	Y	edit
Wednesday May 18th, 9:00 am	Wednesday May 18th, 5:00 pm	8h 0m	Y	edit
Thursday May 19th, 9:00 am	Thursday May 19th, 5:00 pm	8h 0m	Y	edit
Friday May 20th, 9:00 am	Friday May 20th, 5:00 pm	8h 0m	Y	edit
Saturday May 21st, 9:00 am	Saturday May 21st, 5:00 pm	8h 0m	Y	edit
Sunday May 22nd, 9:00 am	Sunday May 22nd, 5:00 pm	8h 0m	Y	edit
Monday May 23rd, 9:00 am	Monday May 23rd, 5:00 pm	8h 0m	Y	edit
Tuesday May 24th, 9:00 am	Tuesday May 24th, 5:00 pm	8h 0m	Y	edit

Edit/Cancel Reservation

From this page, users can edit or cancel a reservation. Reservations can only be cancelled if done so at least 30 minutes prior to the beginning of the reservation, where as reservations can be edited if done so at least 30 minutes before they end. If a user comes to this page after the allowable cancellation period, the cancel action will not be shown. We also added options to edit/cancel all reservations following the one being edited/cancelled if the one being edited/cancelled is a recurring reservation.



Change duration

- Start time: May 18th, 9:00 am
- End time: May 18th, 5:00 pm
- Duration: 8h 0m
- Recurring: Y

Extension: Increase by 01:00 — 30 minute blocks (hours:minutes)

Update Recurrence: YES ○ NO ● – only reservations occuring after this one will be affected

Cancel reservation

Can't make your reservation? It's better to cancel now than to pay for the mandatory grace period.

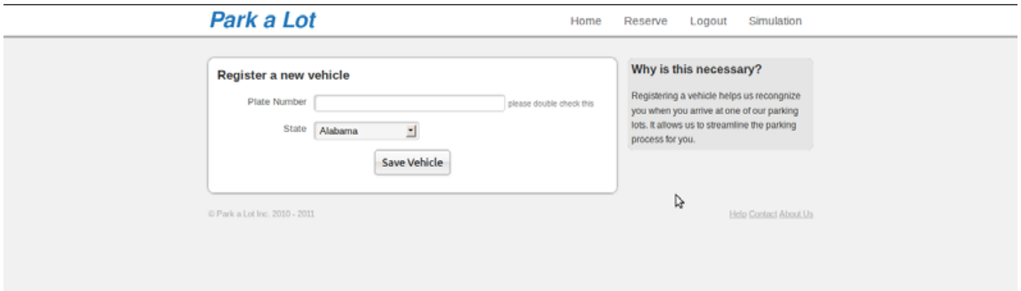
Are you sure?

Cancel Recurrence: YES ○ NO ●

only reservations occuring after this one will be affected

Add Vehicle

The add vehicle page is where the user goes to register a new vehicle with their account. These vehicles are recognized when the user arrives for quick and easy entrance into the garage.



Register a new vehicle

Plate Number: please double check this

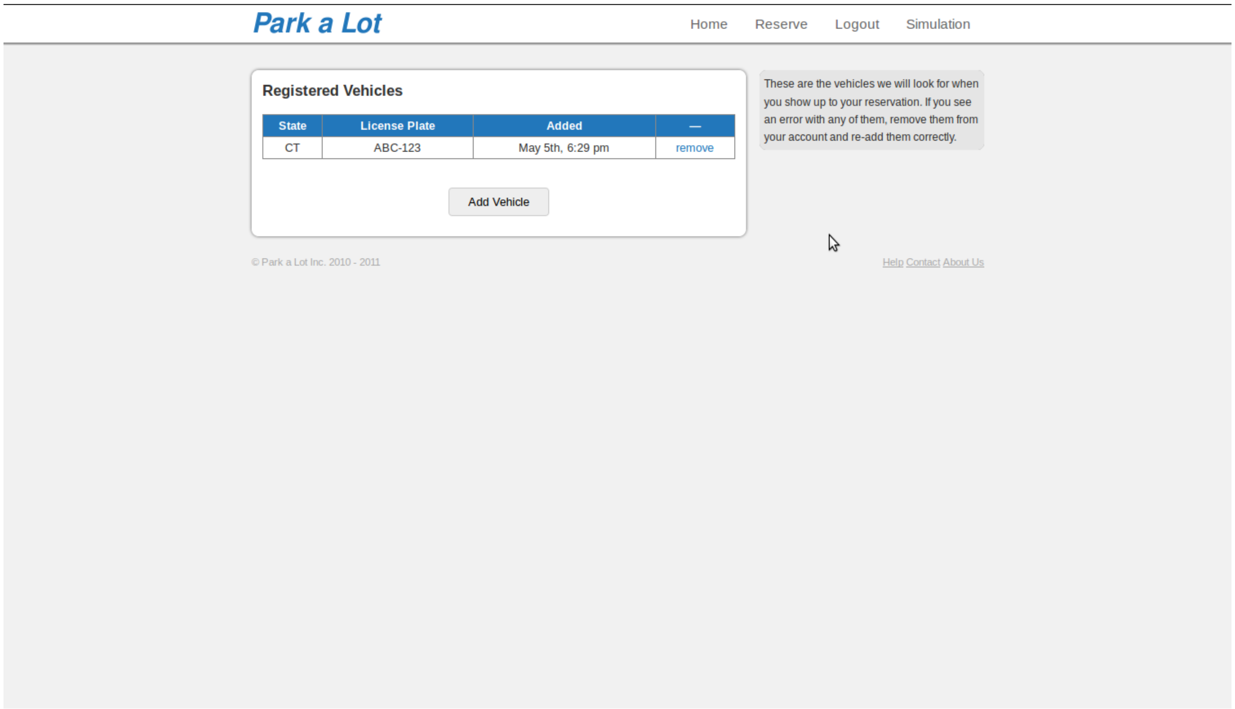
State: Alabama

Why is this necessary?

Registering a vehicle helps us recongnize you when you arrive at one of our parking lots. It allows us to streamline the parking process for you.

List Vehicles

The list vehicles lists all vehicles currently registered with the users account.



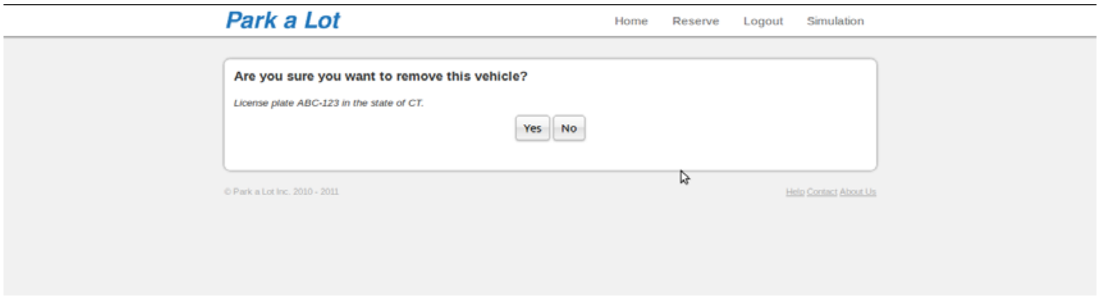
Registered Vehicles

State	License Plate	Added	—
CT	ABC-123	May 5th, 6:29 pm	remove

These are the vehicles we will look for when you show up to your reservation. If you see an error with any of them, remove them from your account and re-add them correctly.

Remove Vehicle

From this page, users can de-register any of their vehicles. This page just asks for confirmation, as well as shows information about the vehicle being removed.

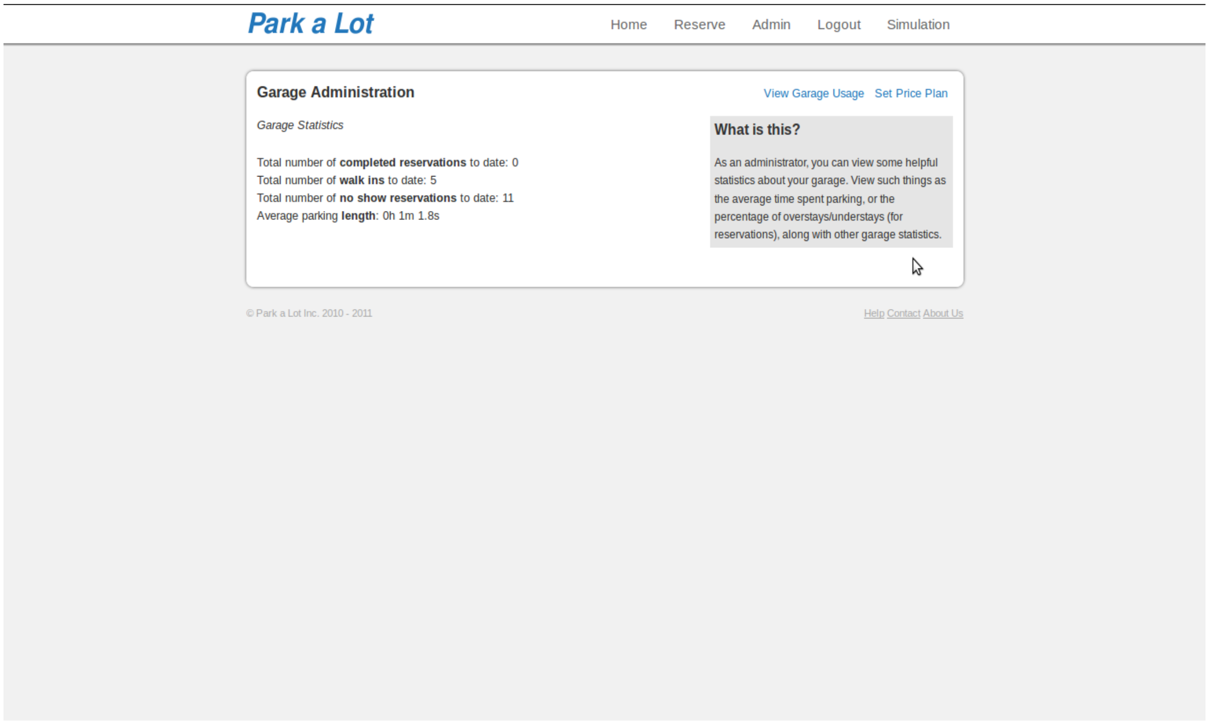


Are you sure you want to remove this vehicle?

License plate ABC-123 in the state of CT.

View Garage Usage

From this page, **administrators** can view a few statistics about the garage, such as the average time spent parking, the percentage of overstays/understays, the number of no-shows, etc.



Garage Administration

Garage Statistics

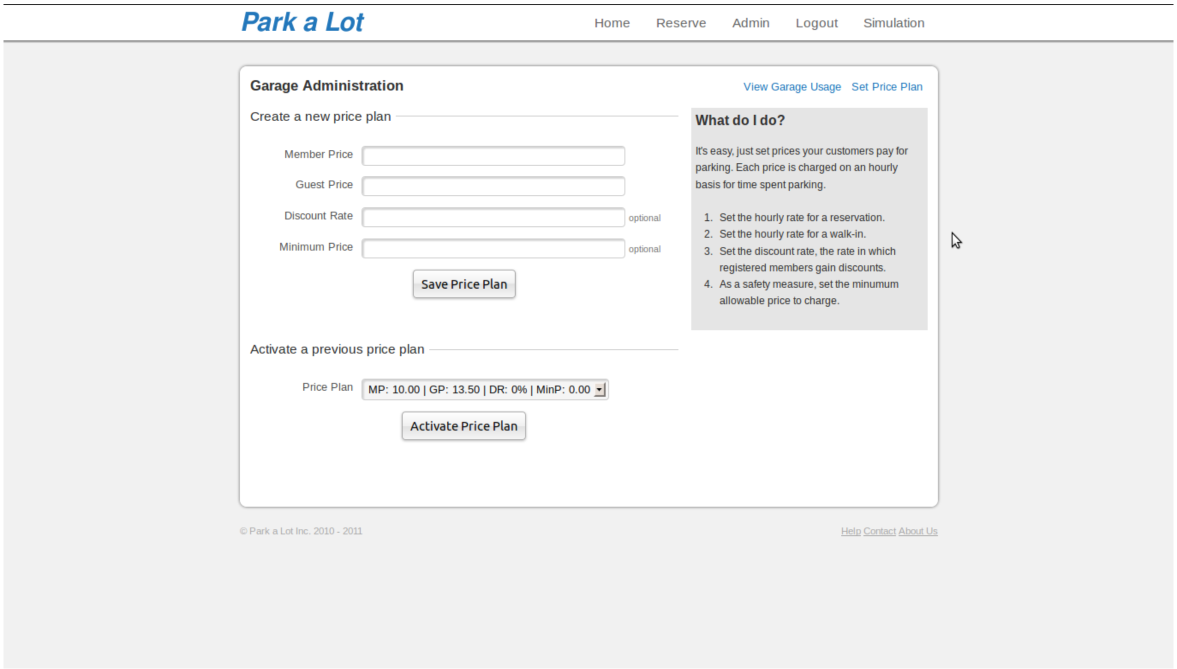
Total number of **completed reservations** to date: 0
Total number of **walk ins** to date: 5
Total number of **no show reservations** to date: 11
Average parking **length**: 0h 1m 1.8s

What is this?

As an administrator, you can view some helpful statistics about your garage. View such things as the average time spent parking, or the percentage of overstays/understays (for reservations), along with other garage statistics.

Set Price Plan

From this page, **administrators** can set new price plans, as well as activate old price plans.



Garage Administration

Create a new price plan

Member Price
Guest Price
Discount Rate optional
Minimum Price optional

Activate a previous price plan

Price Plan: MP: 10.00 | GP: 13.50 | DR: 0% | MinP: 0.00

What do I do?

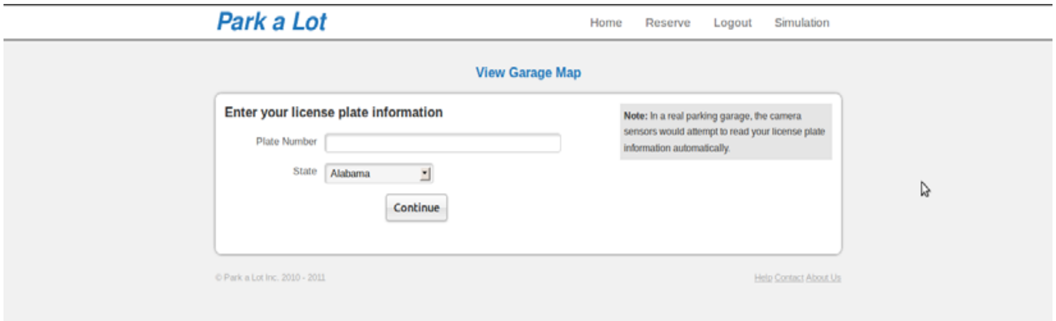
It's easy, just set prices your customers pay for parking. Each price is charged on an hourly basis for time spent parking.

1. Set the hourly rate for a reservation.
2. Set the hourly rate for a walk-in.
3. Set the discount rate, the rate in which registered members gain discounts.
4. As a safety measure, set the minumum allowable price to charge.

Simulation

To test our system, we implemented a simulation test bench. The simulation is to emulate as close as possible the process of physically parking in our garage. Below are a few screenshots of our simulation.

Welcome Screen



Enter your license plate information

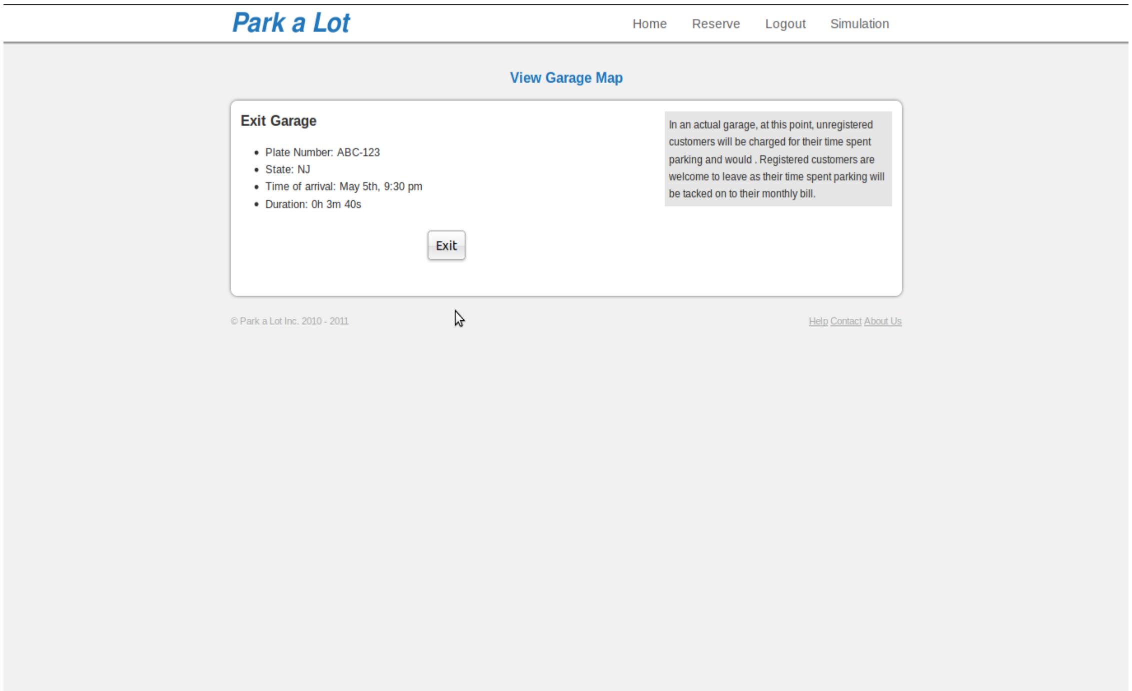
Plate Number

State: Alabama

Note: In a real parking garage, the camera sensors would attempt to read your license plate information automatically.

© Park a Lot Inc. 2010 - 2011

Exit Screen



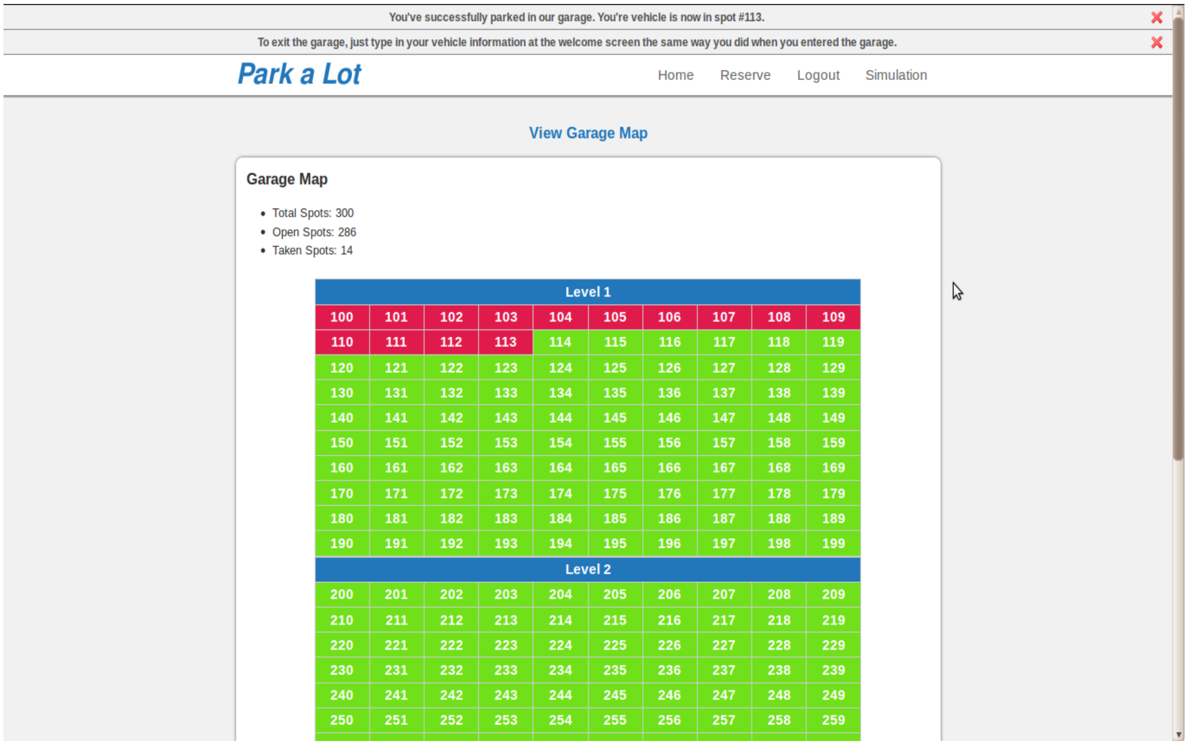
Exit Garage

- Plate Number: ABC-123
- State: NJ
- Time of arrival: May 5th, 9:30 pm
- Duration: 0h 3m 40s

In an actual garage, at this point, unregistered customers will be charged for their time spent parking and would . Registered customers are welcome to leave as their time spent parking will be tacked on to their monthly bill.

© Park a Lot Inc. 2010 - 2011

Garage Map



You've successfully parked in our garage. You're vehicle is now in spot #113.

To exit the garage, just type in your vehicle information at the welcome screen the same way you did when you entered the garage.

Garage Map

- Total Spots: 300
- Open Spots: 286
- Taken Spots: 14

Level 1									
100	101	102	103	104	105	106	107	108	109
110	111	112	113	114	115	116	117	118	119
120	121	122	123	124	125	126	127	128	129
130	131	132	133	134	135	136	137	138	139
140	141	142	143	144	145	146	147	148	149
150	151	152	153	154	155	156	157	158	159
160	161	162	163	164	165	166	167	168	169
170	171	172	173	174	175	176	177	178	179
180	181	182	183	184	185	186	187	188	189
190	191	192	193	194	195	196	197	198	199
Level 2									
200	201	202	203	204	205	206	207	208	209
210	211	212	213	214	215	216	217	218	219
220	221	222	223	224	225	226	227	228	229
230	231	232	233	234	235	236	237	238	239
240	241	242	243	244	245	246	247	248	249
250	251	252	253	254	255	256	257	258	259

Our entire user interface is aimed at an ease-of-use for the user. We've accomplished this by limiting the number of options on each page to the minimal, only the necessary actions to be taken are shown on a page. Our longest form is our user registration form, which only requires 5 input text fields. Our average form only requires 3 input fields, making form completion very simple for the user. Our most complicated form, our create reservation form, is already filled in with the basic information, such as the start date and start time (which we take as starting 3 hours from the current time).

History of Work and Current Implementation Status

Since the last report, there have been many changes to the design and implementation of our project. First, we re-evaluated the Use Cases that we used in Report 1. We broke down some of the requirements so that each requirement will apply to at least one Use Case as opposed to our first report where some business requirements applied to several use cases, and some business requirements applied to none. After re-organizing the Use Cases, we updated our UML diagrams and class diagrams accordingly. Since the previous report and demo, there were many small details that we had to figure out, such as:

- How do we bill the customer?
- How do we ensure maximum usage of the parking lot?
- How do we give discounts?
- How should recurrence reservations work? limits?
- Do we want to let the user register multiple cars? If so how many is allowed?
- How much access do we give the garage administrator?

These key questions were related to the use cases we had yet to implement.

Our progress developed as follows:

Completion of First Report	February 16th, 2011
Completion of Second Report	March 10th, 2011
Completion of First Demo	March 27th, 2011
Major Revision of First Report	April 4th, 2011
Major Revision of Second Report	April 15th, 2011
Implementation	April 28th, 2011
Unit and Integration Testing	May 1st, 2011
Completion of Second Demo	May 1st, 2011
Completion of Third Report	May 6th, 2011

Our milestones in this project were:

- Determine the Use Cases
- Determine our System Design
- Determine our Database Design
- Finalized the Software Design
- Implementation of the Code

Some our key accomplishments are:

- We were able to have concurrent reservations.
- Came up with an algorithm that provides fair discounts to customers.
- Make reservations as fast and easy as possible.
- Efficient reservation swapping algorithm to maximize garage usage.
- Create simulation tab to simulate customer entering and exiting garage.
- Implemented all Use Cases we designed into the website.

Overall, the project is now somewhat usable as a real-world system. It contains all of the major functionality described by our use cases, and has been tested to work properly in a limited but still comprehensive set of tests. Although certainly not ready for a production environment, these system could be used as a great learning tool, and with a little bit more coding and testing (perhaps another month of work or so) all of the bugs could be ironed out and this system could be placed into production.

The one major piece we are missing is the implementation of UC-2 Park, since we did not have any sensors or a parking garage to integrate into our project. Without that, we did not code anything to handle retrieving input from the parking garage sensors, which would be integral to this project functioning successfully. Therefore, any move of this product to production status would require some heavy lifting in that area.

In this project, we have accomplished all of our goals we set out with. In the next section we will take some time to wrap it all up, and then discuss what additional features we thought would work well with our system, yet did not have the time to implement.

Conclusion and Future Work

This project was difficult from the beginning, mainly because none of our group members had ever worked on a full-scale software project like this before. We had all coded applications, but the planning aspect of it and the formulation of such length reports proved somewhat a challenge.

However, at the conclusion of the semester we now have a functioning software system in place, and a long list of documentation and design reports supporting it. Although it was not simple or easy to put together, our careful design process and extremely clean coding style helped to create a powerful and easy to understand piece of code.

In sum, this project is not ready for prime-time. It certainly looks sharp and can handle all of the use cases we have designed for in this report, however some bugs do still exist that would require time to debug and refactor. The reports, on the other hand, should be nearly bulletproof by this point. We have implemented all of our ideas, and even elaborated on the process used to determine that those ideas were the best option for our project, in these reports. With our work as a baseline, it should be simple to translate our work on these pages into a powerful piece of software.

We would like to thank our professor, Ivan Marsic, for all of the help he gave us during this project and over the course of the semester.

Below, we discuss future work that could be integrated into the Park-A-Lot system.

Directed Advertising for Online Reservations

Goal: To offer targeted advertising to customers who create reservations using the online system which is relevant to both the time and location of the currently created reservation.

Design: Suppose that a customer decides to make a reservation for a Park-A-Lot garage in New York City. The garage she parks at is within walking distance of the nearby theatre district. The Park-A-Lot system, in an effort to improve revenue, could target advertising to this customer for the nearby theatre. If the customer makes her reservation for 7pm, the system could show advertisements and recommendations for Broadway shows starting at 8pm, and possibly nearby places to grab a bite to eat.

To start out, this advertising system would be available to nearby retailers and services surrounding each Park-A-Lot garage. A garage owner might sell advertising space to these merchants with the promise that customers parking nearby would be notified of offers, discounts, and deals available to Park-A-Lot customers.

As time goes on, the advertising system could grow to learn what a customer enjoys doing in a nearby area (perhaps present the customer with the option to take a quick survey to more effectively target advertising), and the results from that survey could be employed to provide more granular and refined results when making reservations. A customer's preferences would be stored alongside her account information and be used to target advertising to her when making future reservations.

In an even more advance scenario, it may be possible to book parking at 7pm, and from the Park-A-Lot website also buy Broadway show tickets at 8pm, simply by clicking through a targeted advertising link. The possibilities are endless.

Integration: The main goal of integrating this type of advertising into the Park-A-Lot system should be simplicity. The website was designed with the goal of simplicity in mind, and that goal should be carried through any changes that are made to the system. Therefore, it might be better to simply notify the customer that, when making a reservation for a particular location and time, that offers exist in the area. If the customer so chooses, she could select to view these promotions and could take advantage of them as a Park-A-Lot customer. Another tab could be added to the website which would store information about existing offers for each reservation the customer has made into the future.

The goal of implementing this system should not be driven purely by profit, but rather by the desire to help a customer get the most out of using our system. If we can offer the convenience of a nearby deli to grab a sandwich in, then we have increased our customers' happiness and also increased the chances they will continue to use our service into the future. At no point should this system become like those of other "ad-tracking" companies, who seek to gather information on users and then sell it to the highest bidder. This ideal should be a principle tenant of any advertising system designed for Park-A-Lot.

Multiple Garage Integration

Goal: To allow multiple Park-A-Lots to be interfaced together and allow the sharing of information under one System Administrator account to facilitate parking and convenience for the customer.

Design: Often times, a parking garage may become full and have no more capacity to handle walk-ins or reservations. At this point, it would be advantageous to redirect that customer's business to another nearby parking garage, or risk losing it entirely. A customer may be put off by being told "No Vacancy for You", but might find warmth in the response "We found you something else."

The greatest advantage that could be gained from interfacing multiple Park-A-Lots is the ability to share traffic between them. If they are close, location-wise, this would be almost as if there were one large, combined garage the customer could park in. If a customer coming from out of town cannot find a spot at the garage on Street A, then perhaps a spot in the garage on Street B just one block away might suffice. This convenience would ensure customer loyalty and increase revenue.

The question remains: does two garages need to be owned by the same person or corporation in order to interface them?

Certainly, if two garages are owned by one company or person it would be highly advantageous to interface them, and allow them to share reservations between themselves. Of course, this sort of bond is only as strong as the distance between the two interfaced garages - if they are geographically far apart, the benefits become smaller. In that case, it may be only a mild convenience to know that it is possible for a customer to find a spot in the garage across town.

If the two garages are not owned by the same interest, it may still be an advantage to have a connection between them. Directing your customer to the competition is by no means a way to turn a profit, but letting a customer know you car about them by offering to find them a spot in another garage goes a long way in the line of customer service, and eventually customer loyalty which can turn to easy profit.

Integration: Clearly, this implementation wouldn't require much of a change to our existing database and system structure. Rather, it would build on top of it a new system for sharing information between separate Park-A-Lot systems.

To effectively integrate a multi-garage system composed of several smaller, individual garages we would need to implement a new web interface for administrating the connection between each garage in the system. Technically, it would be an extension of the current System Administrator role.

Currently, we have System Admins who can edit pricing and view parking history at the garages. If we decorated System Admins with the ability to access multiple garages from the same account, and also granted them access to a special web interface for viewing the entire garage system at once, then we could easily accomplish the administration of the multi-garage system without modifying our existing design too much. Since we have already implemented a Protection Proxy for users accessing the system, this change is a simple as adding one additional proxy to our current system.

Of course, having Park-A-Lot suggest to customers a different garage would require some changes to our current system. Each garage would need to be aware of it’s sister garages and their current status, so that when a customer was redirected it would be to a garage that could actually handle the additional traffic.

Customer Parks in Unassigned Spot

Goal: To account and rearrange our system’s database when a customer does not park in his/her assigned spot.

Design: This is a question that is hard to account for because the sensors we have available only track if the spot is occupied or empty, it does not tell us which car or plate is on the parking spot. We understand a customer may be in a rush and parked in the first spot he saw after he/she got out of the elevator, however, what if the system assigns another customer right behind (or later) him/her to that specific spot because it thinks it;s empty? Would the database handle it correctly? what if there is a reservation already assigned to that spot in 30 min after, would it affect the customer that registered already?

The answer is no because, although the customer parked in the incorrect spot, there are still the same amount of spots free in that particular floor. The design of the Park-A-Lot garage is made so that no car can go from one floor to another unless it is through the elevator. Therefore if the elevator drops you off in floor 2, for example, there is no way you can park in a spot on floor 1 or 3 so the same open spots (each floor has 100 spots) will be there. Whether the spot free is 201, 205, 207 or 207,208 and 209, there are still 3 open spots so it would not affect reservations that were assigned, however it does affect the view of the garage map. We would like to have an accurate description of what spots are available and let the system know that someone has mistakenly parked in the wrong spot so that the system readjusts accordingly.

Implementation: One way to implement this is by constantly receive information from the sensors to the database. If we allowed a certain period of time to go by between levels of parking then we will be able to assure that a certain customer parked incorrectly. For example, if customer A parks incorrectly in 205 and then the customer B right behind it was assigned 205, how does the system know that if was the customer A that parked in spot 205 instead of customer B. In order to avoid this problem we would have to make the system assigned parking in different levels. What this means is that if you have a line of 5 cars waiting to receive a parking slot, the system will assigned the first level to the first car, then the second level for the second car and so on, this way we will have enough time to see if the car in level one parked incorrectly before the next customer arrives at level 1 and tries to park. Therefore writing a few restrictions on the way the system determines what spot to assign to the customers, we can control where they park and accurately determine if someone parks in the wrong spots and update our database accordingly.

New Sensors That Track Vehicles

Goal: To implement new sensors, as they become available, into our system to increase the performance.

Design: As time goes on, technology advances. There are new sensors that are coming out that track vehicles as they move through the garage, which will help us to determine where they park exactly. The biggest question is: how would our system handle that? Would we have to change the structure or design of our software to adapt these new sensors?

Certainly not, our architectural design is able to handle these new sensors and replace the old sensors that just detected if the spot was empty or occupied. The main difference would be the amount of space need to accommodate all the data sent by the sensors. These new sensors will be constantly communicating with the server and depending how many sensors we have then it might decrease the performance of the server due to the amount of information being send/received.

Implementation: The way to fix this issue is by increasing the network bandwidth and also the size of the disk so that it allows for more data to be stored. Although some of the classes and functions would have to change slightly, especially those regarding parking it would be of great benefit to have this type of sensors in our system and would help us solve the problem when a customer parks in the wrong spot.

Display Interactive Garage Map

Goal: To have an interactive map in which the user (admin) can select a time of the day and see what spots are occupied, reserved and their information.

Design: In order for the manager to have an accurate idea of how the garage is functioning, it is of great help for him to have a garage map that updates and lets him/her interact with it. Right now, the admin can see a map of the garage with the spots that are full and empty. However, what if the manager wants to know what car or what customer is parked in spot 117 or what time did the vehicle arrived at that spot? Then we need to consider updating the map that we have and let the user input a time of the day that he/she wishes to see the map of the garage. This will help the manager make better decision regarding operating hours and the effect it will have on the business.

Implementation: To implement this interactive map, we will need to modify the current display that we have now. Since what we have now is pretty simple and just color coded. We would need to change each spot to handle to store information (vehicle parked, time of arrival, expected time of departure and customer info) in order to have an interactive map. Another way to let the admin check the information of each spot is by creating a table with all the spots of the garage and just simply allowing the admin to have access to the information stored. For example, when the system gives the parking spot, it will need to store all the customer's info to that parking spot's data. Then simply we can create an interface (or another tab in the admin's profile) to let him enter a specific spot (or click from the map) and obtain all the information needed at any time of the day on any spot in the garage.

Code Appendix

```
/**
*  Parking Spot Assignment
*  by Matt Edwards, Rutgers University 2011
*
*  Program assigns parking reservations to spots (machines) using a simple
linear programming algorithm.  It is constrained by a maximum
*  number of spots (machines).  When the maximum is reached, the program
attemtps to swap existing reservations so that new reservations
*  can be fit within the existing time slots.
*/

#include <iostream>
#include <vector>
#include "Machine.h"

using namespace std;

bool assignJobs(int [], int [], int [], int, int, vector<Machine> & );
bool assignJob(int, int, int, int, vector<Machine> &);
bool trySwap(int, int, int, vector<Machine> &);
void makeReservation(int, vector<Machine> &);
void showReservations(vector<Machine> &);
int setNewMax(int, vector<Machine> &);
void deleteReservation(vector<Machine> &);

/**
*  Program starts with a max of 3 machines.  More can be added.  Each
reservation is assigned to a machine,
*  if free space is available.  Menu options allow for making, deleting and
displaying reservations.
*/
int main()
{
      int maxMachines = 3;
      vector<Machine> machines;

      int input = 0;
      do
      {
            cout << "Reservation Assigner" << endl;
            cout << "-------------------" << endl;
            cout << "1) Make Reservation" << endl;
            cout << "2) Delete Reservation" << endl;
            cout << "3) Show Reservation Table" << endl;
            cout << "4) Set Max Machines" << endl;
            cout << "5) Restart" << endl;
            cout << "6) Quit" << endl;
            cout << endl;
            cout << "Choice:  ";
            cin >> input;
            cout << endl << endl;

            switch (input)
            {
            case 1:
                  makeReservation(maxMachines, machines);

                  break;
            case 2:
                  deleteReservation(machines);

                  break;
            case 3:
                  showReservations(machines);

                  break;
            case 4:
                  maxMachines = setNewMax(maxMachines, machines);

                  break;
            case 5:
                  machines.clear();
                  cout << "Reservations cleared." << endl;
                  cout << endl << endl;

                  break;
            case 6:
                  return 0;

                  break;
            default:
                  cout << "Not a valid menu choice!" << endl;
                  cout << endl << endl;

                  break;
            }
      } while (input != 4);

      system("PAUSE");
      return 0;
}

/**
*  Assigns multiple jobs to the machines vector.
```

```
*
*  @param job                 Array of job numbers.
*  @param r                   Array of job start times.
*  @param d                   Array of job end times.
*  @param n                   Number of jobs, start times, and end times in
each array.
*  @param maxMachines   Maximum number of machines to assign to machines
vector.
*  @param machines            The machines vector, which stores the machines
that jobs are assigned to.
*
*  @return bool True if all jobs assigned, false if not.
*/
bool assignJobs(int job[], int r[], int d[], int n, int maxMachines,
vector<Machine> &machines)
{
      int success = true;
      for (int j = 0; j < n; j++)
      {
            success = assignJob(job[j], r[j], d[j], maxMachines, machines) &&
success;
      }

      return success;
}

/**
*  Assigns a single job to the machines in the machines vector.  Attempts to
assign job without moving any
*  existing reservations, then will try to swap other reservations to make
room for new reservation.
*
*  @param job                 The job number to be assigned.
*  @param r                   Job start time.
*  @param d                   Job end time.
*  @param maxMachines   Maximum number of machines to assign jobs to.
*  @param machines            Vector which stores all machines being assigned
jobs.
*
*  @return bool True if job assigned, false if not.
*/
bool assignJob(int job, int r, int d, int maxMachines, vector<Machine>
&machines)
{
      // Determine number of machines in vector, if any
      int i = 0;
      if (machines.size() == 0)
      {
            Machine m(i);
            machines.push_back(m);
      } else {
            i = machines.size() - 1;
      }

      int assigned_flag = false;  // Set to true when job is assigned
      int k = 0;
      while(!assigned_flag && k < machines.size())
      {
            if (machines[k].isFree(r, d))
            {
                  machines[k].assign(job, r, d);
                  assigned_flag = true;
            }
            k++;
      }  // Loop until job is assigned or we've hit the end of the machines
vector

      // Either add new machine or swap jobs
      if (!assigned_flag)
      {
            if (i+1 == maxMachines)  // Can't add any more machines
            {
                  // Try to swap two reservations to make room for the new
one
                  bool swap_flag = trySwap(job, r, d, machines);
                  return swap_flag;
            } else {
                  // Can add another machine, do so
                  i++;
                  Machine newMachine(i);
                  newMachine.assign(job, r, d);
                  machines.push_back(newMachine);
            }
      }

      return true;
}

/**
*  Attempt to swap two reservations so that a third (new) reservation can be
added.
*  This function considers only pairwise swapping (will not shift more than 2
jobs to make room for a third).
*
*  @param job           Job number to be added.
*  @param start         New job start time.
*  @param end           New job end time.
*  @param machines      Vector of machines.
```

```
*
*  @return bool True if swap successful, false if no swap possible.
*/
bool trySwap(int job, int start, int end, vector<Machine> &machines)
{
      for (int i = 0; i < machines.size(); i++)
      {
            // Determine why we can't assign a job to a machine
            int conflict = machines[i].findConflict(start, end);  // Returns
index of first intersection between new job and existing job already on
machine
            int conflictingJob = machines[i].findNextJob(conflict);
            int conflictingJobStart =
machines[i].findJobStart(conflictingJob);
            int conflictingJobEnd = machines[i].findJobEnd(conflictingJob);

            // Determine if there is any open time on machine, starting from
new job start time
            if (conflict > start)
            {
                  int diff = end - conflict;  // How much more time we need
                  for (int j = i+1; j < machines.size(); j++)  // Loop
through remaining machines
                  {
                        // Is there enough open time on machine j to
complement time we have on machine i?  e.g. Can we make up diff
                        if (machines[j].isFree(conflict, end))
                        {
                              // If so, determine next job on machine j that
we would need to swap with conflicting job on machine i
                              int nextJob =
machines[j].findNextJob(conflict);
                              int nextJobStart =
machines[j].findJobStart(nextJob);
                              int nextJobEnd =
machines[j].findJobEnd(nextJob);

                              // Determine if swap possible, so that job on
machine j swapped to machine i will not cause any conflicts, and vice-versa
                              if (machines[i].isFree(conflictingJobEnd,
nextJobEnd) && machines[j].isFree(conflictingJobStart, nextJobStart))
                              {
                                    // Then swap, start by deleting jobs on
machines i and j
                                    machines[i].deleteJob(conflictingJob);
                                    machines[j].deleteJob(nextJob);

                                    // Then reassigning those jobs, to effect
a swap
```

```
                                    machines[i].assign(nextJob, nextJobStart,
nextJobEnd);
                                    machines[j].assign(conflictingJob,
conflictingJobStart, conflictingJobEnd);

                                    // Finally, assign new job
                                    machines[i].assign(job, start, end);

                                    return true;
                              }
                        }
                  }
            }
      }

      return false;
}

/**
*  Helper function that handles user input for making a reservation.
*
*  @param maxMachines   Maximum number of machines to assign.
*  @param machines            Vector of machines.
*/
void makeReservation(int maxMachines, vector<Machine> &machines)
{
      int rnum = Machine::getLastJobNum();
      int start, end;

      cout << "Start Time: ";
      cin >> start;
      cout << "End Time: ";
      cin >> end;

      bool success = assignJob(rnum, start, end, maxMachines, machines);
      if (!success)
      {
            cout << "Reservation not made, cannot exceed max machines" <<
endl;
      }
}

/**
*  Helper function for displaying reservations to the console.
*
*  @param machines Machines vector to display.
*/
void showReservations(vector<Machine> &machines)
{
	for (int j = 0; j < machines.size(); j++)
	{
		Machine hold = machines.at(j);
		cout << "Machine " << j << "  ";
		hold.printTasks();
	}

	cout << endl << endl;
}

/**
*  Helper function to handle user input when setting a new max number of
machines.
*  Maximum number of machines must be greater than 0, and max cannot be set
smaller than
*  current number of machines assigned jobs in machines vector.
*
*  @param oldMax  The old max number of machines.
*  @param machines      Vector of machines.
*
*  @return int New maximum number of machines.
*/
int setNewMax(int oldMax, vector<Machine> &machines)
{
	int newMax = oldMax;
	cout << "Max Machines (currently " << oldMax << "): ";
	cin >> newMax;
	newMax = (newMax > 0) ? newMax : oldMax;

	if (newMax < machines.size())
	{
		newMax = oldMax;
		cout << "Max Machines cannot be less than size of machines
vector." << endl;
	} else {
		cout << "New Max Machines:  " << newMax << endl;
	}

	cout << endl << endl;

	return newMax;
}

/**
*  Helper function to handle user input when deleting a reservation by
number.
*
*  @param machines Machines vector.
*/
```

```
void deleteReservation(vector<Machine> &machines)
{
      showReservations(machines);

      int rnum = 0;
      cout << "Delete which reservation number: ";
      cin >> rnum;
      cout << endl;

      for (int i = 0; i < machines.size(); i++)
      {
            machines[i].deleteJob(rnum);
      }

      showReservations(machines);
}
```

```
#pragma once
class Machine
{
      private:
            int task[10];
            int number;
            static int lastJobNum;
            void clearFuture();

      public:
            Machine(int);
            void printTasks(void);
            bool isFree(int, int);
            int findConflict(int, int);
            int findNextJob(int);
            int findJobStart(int);
            int findJobEnd(int);
            bool isIdle();
            void assign(int, int, int);
            void advanceTime(int);
            void deleteJob(int);
            static int getLastJobNum();
};
```

```
/**
*  Class Machine
*  by Matt Edwards, Rutgers University 2011
*
*  A Machine can store jobs starting and ending at a specific time.  Jobs
have a number which differentiates them.
```

```
*/

#include <iostream>
#include "Machine.h"

using namespace std;

// Static variable for keeping track of the last job number assigned.
Incremented when new job is assigned.
int Machine::lastJobNum = 1;

Machine::Machine(int num)
{
      number = num;
      for (int i = 0; i < 10; i++)
            task[i] = 0;
}

void Machine::printTasks()
{
      for (int i = 0; i < 10; i++)
            cout << task[i] << ' ';
      cout << endl;
}

bool Machine::isFree(int start, int end)
{
      for (int i = start; i < end; i++)
            if (task[i] > 0) return false;
      return true;
}

int Machine::findConflict(int start, int end)
{
      for (int i = start; i < end; i++)
            if (task[i] > 0) return i;
      return -1;
}

int Machine::findNextJob(int start)
{
      for (int i = start; i < 10; i++)
            if (task[i] > 0) return task[i];
      return -1;
}

int Machine::findJobStart(int jobNum)
{
      for (int i = 0; i < 10; i++)
```

```
            if (task[i] == jobNum) return i;
      return -1;
}

int Machine::findJobEnd(int jobNum)
{
      for (int i = 9; i >= 0; i--)
            if (task[i] == jobNum) return i+1;
      return -1;
}

bool Machine::isIdle()
{
      if (Machine::isFree(0,10))
            return true;
      return false;
}

void Machine::assign(int job, int start, int end)
{
      for (int i = start; i < end; i++)
            task[i] = job;
      lastJobNum++;
}

void Machine::advanceTime(int shift)
{
      int temp[10] = {0};
      for (int i = 0; i < 10-shift; i++)
            temp[i] = task[i+shift];

      for (int i = 0; i < 10; i++)
            task[i] = temp[i];

      //Machine::clearFuture();
}

void Machine::clearFuture()
{
      for (int i = 0; i < 10-1; i++)
      {
            if (task[i] != task[i+1])
            {
                  for (int j = i+1; j < 10; j++)
                        task[j] = 0;
            }
      }
}
```

```
void Machine::deleteJob(int job)
{
      for (int i = 0; i < 10; i++)
      {
            if (task[i] == job)
                  task[i] = 0;
      }
}

int Machine::getLastJobNum()
{
      return lastJobNum;
}
```

References

[1] “Event Driven Architecture”, Wikipedia, http://en.wikipedia.org/wiki/Event-driven_architecture

[2] Model-View-Controller Tutorial, http://net.tutsplus.com/tutorials/other/mvc-for-noobs/

[3] Introduction to UML 2 Package Diagrams, http://www.agilemodeling.com/artifacts/packageDiagram.htm

[4] The Kohana Framework, http://kohanaframework.org/

[5] “Interval Scheduling, Reservations and Timetabling”, Tim Nieberg, http://www.or.uni-bonn.de/cards/home/nieberg/sched08/ivrtSCHED.pdf